Specifying Overlaps of Heterogeneous Models for Global Consistency Checking

Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki
Univesity of Waterloo
Waterloo, ON, Canada
{zdiskin, yingfei, kczarnec}@gsdlab.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 *mutlimodel*, 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 a formal semantics for it based on category theory.

Categories and Subject Descriptors

D.2.12 [**Software Engineering**]: Interoperability

General Terms

Design, Languages, Theory, Verification.

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 [25] or a pair of model [9]. 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 (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.

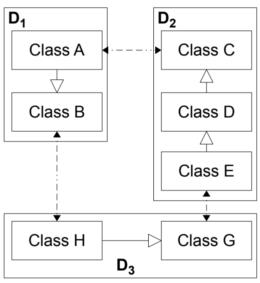

Figure 1: Three globally inconsistent models

The example shows two issues 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 even though they have different names. Second, when we have an overlap specification, we need an approach to check global consistency.

Sabezadeh et al.[22] proposed to check global consistency of homogeneous models by their merging. First, 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 of local models declared the same in the correspondence diagram become one element. Finally, consistency of the merged model is checked. Thus, verifying global consistency amounts to checking consistency of a single model. However, the approach was developed for the case of homogeneous models only.

The goal of the paper is to adopt the *consistency-checking-by-merging (CCM) idea* for the heterogeneous situation. A straightforward solution is to first merge all involved metamodels so that all local models become instances of the same global metamodel; then we can merge them 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 the homogeneous set of projections.

MDI'2010, October 5, 2010, Oslo, Norway

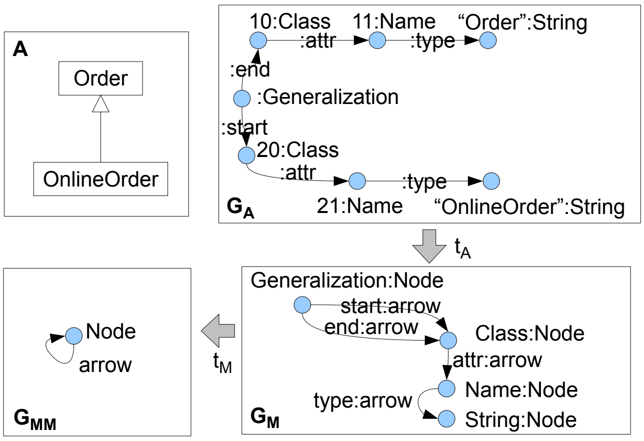

Figure 2: Graph Representation

We formulate the framework in a general way based on category theory. This makes it applicable to a wide class of models and metamodels, whose carrier structures are graphs, attributed graphs, or general *graph-like structures*. By the latter we mean systems of sets (nodes, arrows, arrows between arrows...) interrelated by (source and target) functions.

Realization of the approach requires several challenging issues to be solved: type-safe model matching, specification of indirect overlap between metamodels, and inter-metamodel constraints. We will discuss these issues in more detail in Section 3 after we briefly outline the basics of CCM-approach in Section 2.

The rest of the paper is structured as follows. Section 4 describes our main techniques with simple examples. Section 5 presents general definitions and constructions in a semi-formal way. Relation to other works is discussed in Section 6. Section 7 concludes.

2. BACKGROUND: HOMOGENEOUS OVERLAP AND CONSISTENCY

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

2.1 Software models are typed graphs

We consider metamodels as pairs $M = (G_M, C_M)$ 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 $A = (G_A, t_A)$ with G_A a graph (typically much bigger than G_M) and $t_A \colon G_A \to G_M$ 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. 2 shows how to represent a UML class diagram A as a typed graph. G_M is the graph representing the metamodel of UML class diagrams; G_A is the graph representing the diagram A; and t_A is the type mapping. UML classes, attributes, primitive values and generalization relations are represented as nodes; their relationships are captured by arrows. The value of mapping t_A at an element e is given after colon, e.g., expression "10:Class" means $t_A(10)$=Class for node 10. Identifiers of some elements are omitted, e.g., for all arrows. To refer to the elements, we will use the following notation: if N is the name of an element e, let &N be the slot (owned by e) where the name is held, and &&N be e itself. For example, &'Order'=11 and &&'Order'=10. In its turn, graph G_M is typed over the metametamodel graph G_{MM}.

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. For example, (C1) a class has only one name, or (C2) a class has only one parent class (we assume that multiple inheritance is prohibited), or (C3) classes with stereotype 'singleton' cannot be instantiated with more than one object. Note that constraints can either be imposed by a particular metamodeling technique, e.g., constraints (C1) and (C2), or can be user-defined, e.g., (C3), in a suitable language like OCL. In this paper we do not distinguish these two types and consider them abstractly as constraints over graphs.

2.2 Matching models via spans

Suppose two business analysts independently build two UML diagrams, A_1 and A_2 in Figure 3. To check their global consistency, we first need to specify overlap between the diagrams. Suppose we know that class 'OnlineOrder' in diagram A_1 and class 'Order' in A_2 refer to the same class, and their 'price' attributes refer to the same attribute. We could write the following two informal equations

$$\text{OnlineOrder@}A_1 \;=\; \text{Order@}A_2$$
$$\text{price@}A_1 \;=\; \text{price@}A_2.$$

Note that 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 A_0 shown in the middle of Fig. 3. The question mark indicates that the name of the class is unknown and the corresponding slot is empty. That is, the slot node (:Name) in the graph representing model A_0 does not have any arrow (:type) adjoint to it (see the auxiliary top-rightmost box in the figure). Nevertheless, it is convenient to denote the slot and its owner by &'?' and &&'?' like if '?' were a name.

Since elements of model A_0 represent pairs of elements (e_1, e_2) with $e_i \in A_i$, $i = 1, 2$, we have two inter-model mappings $f_i \colon A_0 \to A_i$. Formally, these mappings are functions between the corresponding graphs, e.g., f_1 acts on G_{A_0}'s nodes as follows:

f_1(&&`'?'`) = &&`'OnlineOrder'`, f_1(&&`'price'`) = &&`'price'`,
f_1(&`'?'`) = &`'OnlineOrder'`, f_1(&`'price'`) = &`'price'`,
f_1(`'price'`) = `'price'`.

Its action on arrows is evident. Mapping f_2 is defined similarly. Importantly, both mappings preserve the types of elements, i.e., commute with the typing mappings of the corresponding graphs. In Fig. 3 we specify mappings in a shortened way, but precise formal specifications like above will be needed when we consider merging.

We call a pair of mappings with a common source a *(binary) span*. The source (model A_0) is called the *head* of the span, mappings f_i are *legs* and their targets (models A_i) are *feet*. Thus, an overlap of two homogeneous models is specified by a *correspondence* span over the same metamodel. An overlap of n models is described by an n-ary span with n legs and feet.

2.3 Merging and conflicts

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

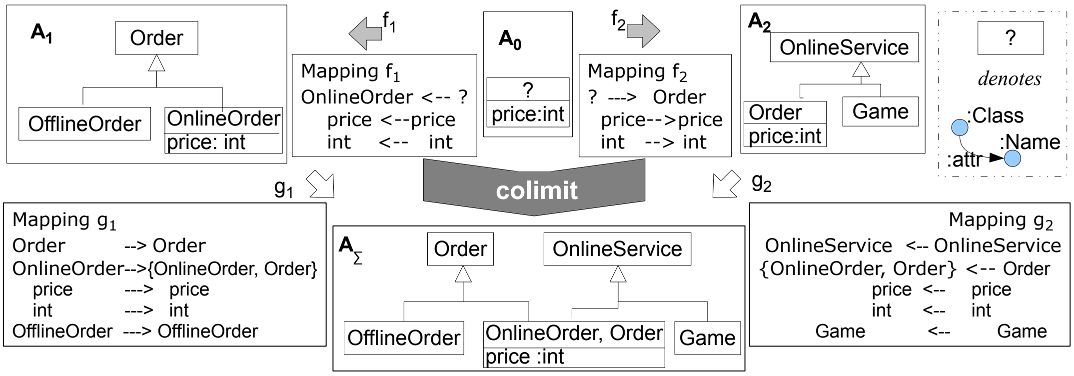

Figure 3: Homogeneous Model Matching

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 A_Σ in Figure 3, in which the merged graph has five rather than six class nodes because of gluing. Class &&`{OnlineOrder,Order}` has one name slot because the two local name slots were also glued, but this slot holds two names since they are not (and cannot be) equated in the head. (A precise formal specification of the mechanism can be found in [6]). Besides graph A_Σ, merging also produces two graph mappings $g_i\colon A_i \to A_\Sigma$ 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 [21] or [6]. It follows from general properties of colimit that the merged graph G_{A_Σ} is correctly typed over graph G_M (with M denoting the metamodel of class diagrams).

After we have built the merged graph, we can check whether it satisfies all constraints defined in the metamodel (say, with a checking tool). In our example, we find two violations: class `{OnlineOrder, Order}` has (i) two names and (ii) two parent classes.

3. FROM HOMO- TO HETEROGENEOUS MULTIMODELING: THE PROBLEMS

Existing CCM-approaches [22] handle the homogeneous case well, but in practice software models are often heterogeneous. Business analysts, database experts, and object-oriented software designers all work with different models in different languages, say, BPMN, ER, UML.

For instance, Fig. 4 presents three different UML models of a system developed independently by three different teams: a class diagram `cd`, a statechart `sc`, and a sequence diagram `sd`, whose simplified metamodels are shown in the right half of the figure.

Since the models are developed independently, synonymy and homonymy of names, and other similarities and discrepancies between models are quite possible. For example, classes `Order` in the class and the sequence diagram may refer to the same or different classes of the system. If they refer to the same class, we need to check whether message `settled@sd` refers to operation `setSettled@cd`. If it is the case, we have a naming conflict (synonymy) between the

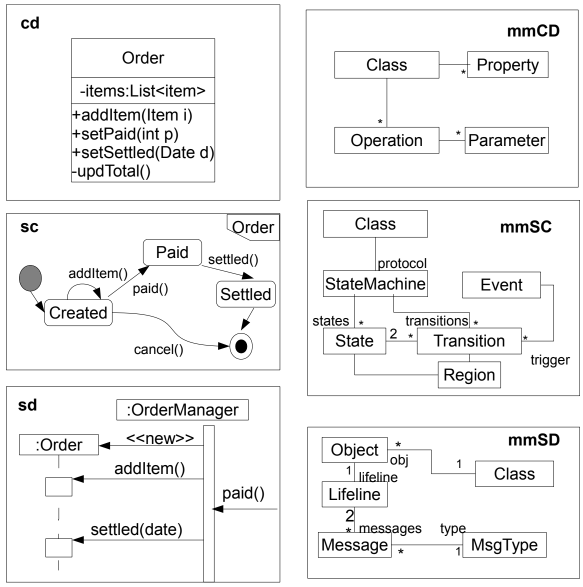

Figure 4: Motivating Example

models; in addition, parameters of the message and the operation it refers to are named differently (homonymy): 'd' in `cd` and 'date' in `sd`. Such conflicts are fixable by renaming, but we also need to take into account the statechart.

There may be more serious discrepancies between the models. Suppose, for example, that the sequence diagram states that parameter 'date' is of type `String` while class diagram declares a different type for the same parameter. This discrepancy violates the condition that an operation parameter has a single type. This condition is stated in both metamodels (of class and sequence diagrams), but message `settled` does not belong to a class diagram and operation `setSettled` is not in a sequence diagram. There are also semantically motivated constraints that directly regulate interaction between models defined in different metamodels. For example, we may require that the interaction described by the sequence diagram is to be allowed by the statechart's state machine. Thus, specifying overlap and checking global consistency of heterogeneous models gives rise to several specific problems caused by heterogeneity.

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.

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 (like conformance of traces to statecharts) are important for heterogeneous multimodeling. These 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 become crucial as soon as we consider type-safety as a fundamental requirement. The latter implies that model interaction should be coherent with metamodel interaction, and hence "the metamodel" of a heterogeneous multimodel is a system of metamodels *together* with their relationships rather than a discrete set of isolated metamodels. To address this new dimension of multimodeling, we need a language for specifying systems of interacting metamodels.

4. HETEROGENEOUS OVERLAP AND CONSISTENCY BY EXAMPLES

In this section we incrementally introduce our approach. We will consecutively consider very simple examples addressing the principle points: (i) building overlap metamodels to ensure type-safe matching, (ii) the necessity of derived elements, (iii) inter-model constraints, and (iv) N-ary multimodeling with a non-trivial correspondence diagram.

4.1 From heterogeneous to homogeneous overlaps and type-safety

Consider the overlap between class diagram cd and sequence diagram sd in Fig. 4. Suppose we know that class `Order` together with methods `addItem`, `setSettled` in cd refer to the same elements in the system as class `Order` together with message types `addItem, settled` in sd. However, if we take the type discipline strictly, direct linking of these elements is prohibited because their types reside in different metamodels. Hence, before matching models we need to match their metamodels, mmCD and mmSD, as shown in Fig. 5. Namely, we state that metaclasses `Class@mmCD` and `Class@mmSD` refer to the same concept, and metaclasses `Operation@mmCD` and `MsgType@mmSD` are also synonyms. These declarations can be presented by a span in the middle of Fig. 5. The head of this span is a new *overlap* metamodel mmCA, and two legs $\mathsf{m}_{1,2}$ map it to the two metamodels we are matching.

Note that the overlap metamodel can be considered as a common view between mmCD and mmSD, and mappings m_1,m_2 as the corresponding view definitions. The view definition $\mathsf{m}_1\colon \mathsf{mmCA} \to \mathsf{mmCD}$ can be executed for any instance of mmCD (i.e., for any class diagram) by extracting its mmCA-portion and respectively changing its type mapping. For example, class diagram cd shown in left upper corner of Fig. 6 (we have slightly simplified the class diagram from Fig. 4 to save space) will be translated into diagram $\mathsf{cd2CA}$ typed over metamodel mmCA. We write $\mathsf{cd2CA} = \mathsf{get}^{\mathsf{m}_1}(\mathsf{cd})$ 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 $\mathsf{cd2CA}$ *the* mmCA*-projection* of cd. Since the ownership between classes and actions is not specified in the overlap, the $\mathsf{cd2CA}$-view of cd will be just a discrete set of named elements. Note also that the view is computed along with traceability mappings $\overline{\mathsf{m}_1}\colon \mathsf{cd2CA} \to \mathsf{cd}$

Similarly, sequence diagram sd in the top right corner of Fig. 6 is translated into a discrete set $\mathsf{sd2CA} = \mathsf{get}^{\mathsf{m}_2}(\mathsf{sd})$ of named elements also typed over mmCA, along with its traceability mapping $\overline{\mathsf{m}_2}$. Since both views are instances of the same metamodel, we can type-safely match them and build a span $(\mathsf{ca}_1, f_1, f_2)$. This span and the corresponding merge (colimit) are shown in the middle part of Fig. 6. They reveal a conflict between the models: actions `setPaid@cd2CA` and `paid@sd2CA` are linked but their names are different (in the merge model cd+sd, the action with two names is shown by ?).

4.2 Indirect overlap

A closer inspection of the original models cd and sd shows that the conflict above is mistaken because message 'paid' is actually an operation of class OrderManager rather than Order. The error occurred because our overlap model does not capture the relationship between classes and actions (operations). To build a better overlap, we need to match the ownership edge `Class-Operation@mmCD` and similar edge `Class-MsgType@mmSD`. However, the latter is not directly included into the metamodel mmSD. Nevertheless, the concepts of `MsgType` and `Class` are related indirectly via a sequence of intermediate edges: a message ends at the lifeline, which belongs to an object, which belongs to a class. We can compose these three edges into a new — *derived* — edge `Class-MsgType` shown in the metamodel mmSD^+ (Fig. 7) with a dashed line. In addition, we use UML stereotypes and prefix the names of derived elements by a slash.

In more detail, we augment metamodel mmSD with a new element `mtp` (read "`messageType`") coupled with its definition, i.e., specification of some operation computing the instances of the derived element. In our case, the operation is sequential composition of four association links leading, consecutively, from instances of Class to instances of MsgType. It can be written in OCL as follows:

```
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. 7, and defining $m_1(\texttt{act}) = \texttt{oper}$, $m_2(\texttt{act}) = \texttt{/mtp}$. Since mappings m_1, m_2 in Fig. 7 define richer views than earlier defined mappings m_1,m_2 in Fig. 5, projections $\mathsf{cd2CA}$ and $\mathsf{sd2CA}$ in Fig. 7 are also richer than in Fig. 5 and include links between classes and operations. We at once see that matching `setPaid@cd2CA` and `paid@sd2CA` is illegal, and the corresponding "equation" must be removed from the span. The result of merging models $\mathsf{cd2CA}$ and $\mathsf{sd2CA}$ modulo the new span ca_1 is shown in the middle bottom of Fig. 7. It is a correct mmCA model satisfying the constraints of mmCA: an element may have only one name, and different actions owned by a class are named differently.

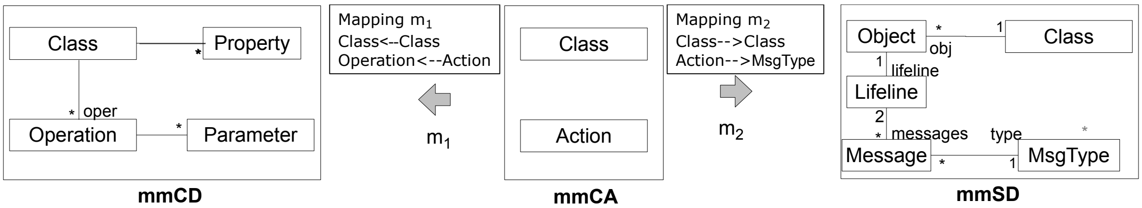

Figure 5: Example of metamodel overlap

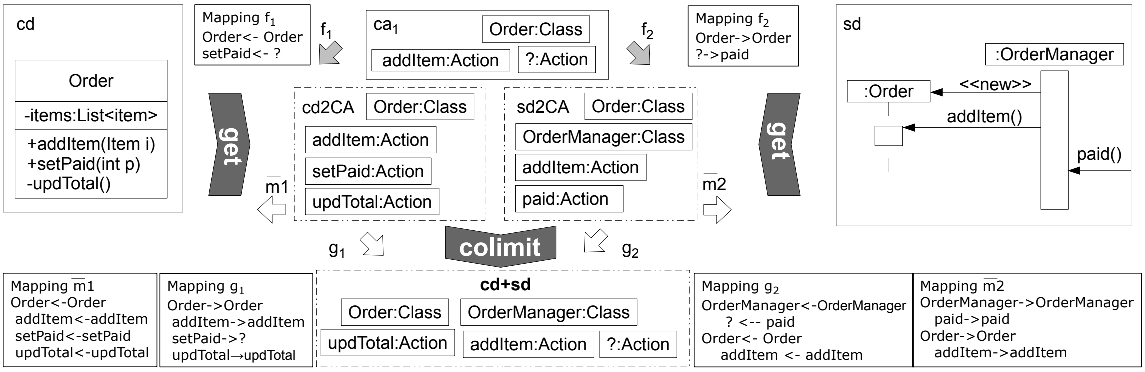

Figure 6: Example of model overlap over the respective metamodel overlap (see Fig. 5 for view definitions)

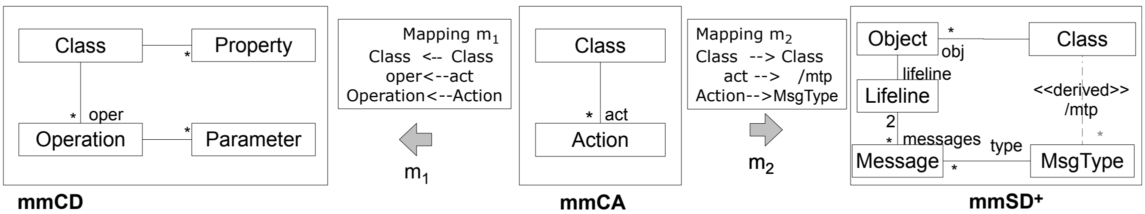

Figure 7: Matching basic and derived meta-elements

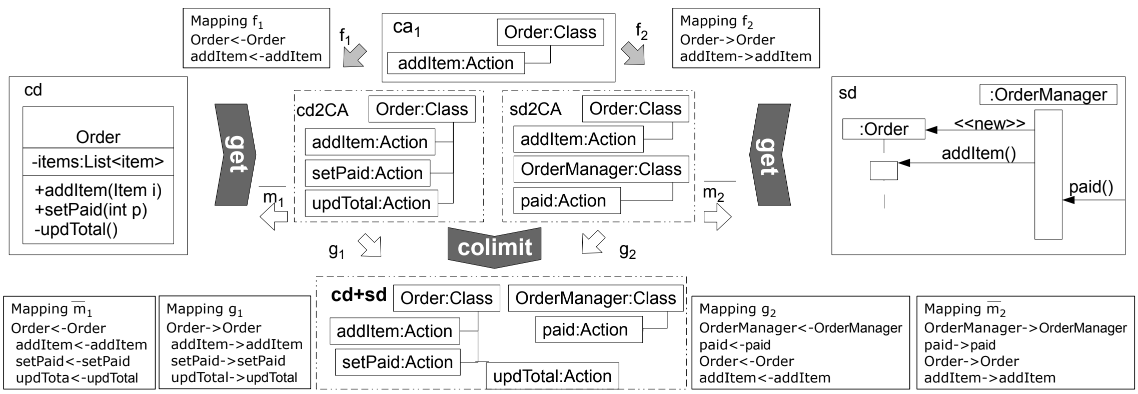

Figure 8: Matching basic and derived elements (see Fig. 7 for view definitions)

The next section will show more interesting cases of using derived elements in overlap specification.

4.3 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, as discussed in Section 3, there may be important constraints which reside 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 $\boldsymbol{t\sharp sm}$ meaning "Traces are to conform to the StateMachine". Declaration of the constraint $\boldsymbol{t\sharp sm}$ requires elements from both metamodels, mmSD and mmSC, and cannot be done in either of them. Hence, a new metamodel in which $\boldsymbol{t\sharp sm}$ could be specified has to be built. In this section we first show how to build such a metamodel, and then 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 $\boldsymbol{t\sharp sm}$.

To declare $\boldsymbol{t\sharp sm}$, 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. 9. The upper half of this metamodel is "taken" from the sequence diagram metamodel mmSD as specified by mapping m_1 in Fig. 9. 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. 9 (and we again use derived elements). Having built metamodel mmCTrSM, we declare in it the constraint $\boldsymbol{t\sharp sm}$ with its intended semantics. We call the configuration (m_1, 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. 9 and other figures below, a semi-arrow head indicates partiality of the mapping.

The next step is to project models sd and sc to the metamodel mmCTrSM. We cannot directly execute view definitions m_j ($j = 1, 2$) because they are partial, but we can execute them in three steps.
Step 0. We explicitly specify the domains mmCTr and mmSM of mappings m_j ($j = 1, 2$; see Fig. 10) on which they become totally defined mappings $m!_j$; inclusion mappings i_j embed the domains into the head of the span.
Step 1. Total view definitions $m!_j (j = 1, 2)$ are executed for models sd and sc and produce views sd2CTr and sc2CSM over metamodels mmCTr and mmCSM resp.
Step 2. Because 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 sd2CTr, sc2CSM with inclusion mappings i_j, $j = 1, 2$ and get typing mappings into mmCTrSM. In Fig. 10, these new typing mappings are marked by $*$.

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

Now we have two models sd2CTr and sc2CSM over the same metamodel mmCTrSM. To finish consistency checking, the user must match the models and build a correspondence span, say, (f_1, ca2, f_2). The head of the span is denoted by ca2 because it is, in fact, an instance of metamodel mmCA built in Section 4.2 (it can be formally proved). After that, the system merges models modulo the span and checks the result against the constraints in mmCTrSM, including the inter-metamodel constraint $\boldsymbol{t\sharp sm}$. The entire procedure is well seen in the right half of Fig. 10: 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).

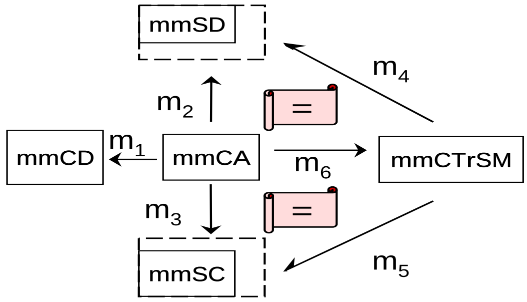

Figure 11: Metamodel schema of the example in Fig. 4

4.4 N-ary multimodeling and metamodel schemas

In this subsection we consider our full example involving all three models, cd, sd and sc.

First we build a ternary span (mmCA, m_1, m_2, m_3) specifying correspondences between operations, messages and transitions in cd, sd, sc resp. as shown in Fig. 11; a dashed frame indicates that the metamodel is augmented with derived elements defined by queries. Ternary span mmCA is a straightforward extension of binary span mmCA built in Section 4.2 with a new leg towards sc. Projecting the three models to the head, matching them with a ternary correspondence span, say, ca3 (see Fig. 12), merging projections modulo ca3, and finally checking the constraints against the merge can be done in exactly the same way as in Section 4.2. A minor distinction is that the leg ca3$\rightarrow$get^{m2}(sd) is partial because there are binary (rather than ternary) correspondences like (`setPaid`@cd, `paid`@sc) that do not involve sd's elements; colimit operation consumes such correspondences as well.

The second point of consistency checking is at the span (mmCTrSM, m_4, m_5) where constraint $\boldsymbol{t\sharp sm}$ is to be checked as explained in Section 4.3. However, when we consider all three models, the correspondence span ca2 between projections get^{m4}(sd) and get^{m5}(sc) can be derived from the span ca3 rather than specified independently. Indeed, we have mapping m_6 that sends nodes `Class` and `Action` and edge `act` between them to the corresponding elements in mmCTrSM. By applying the retyping procedure explained in Section 4.3, we project the span ca3 into mmCTrSM and get a span ca2 as shown in Fig. 12 (where the block arrow rtp^{m6} denotes the retyping operation). After the span ca2 is computed, we proceed exactly as described in Section 4.3 and check the constraint $\boldsymbol{t\sharp sm}$.

An important property of the metamodel schema in Fig. 11 is commutativity of the two triangle diagrams (note two =-

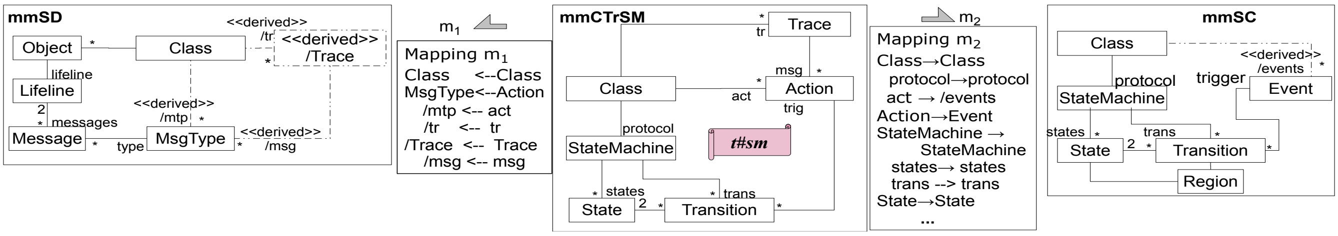

Figure 9: Specifying inter-metamodel constraints

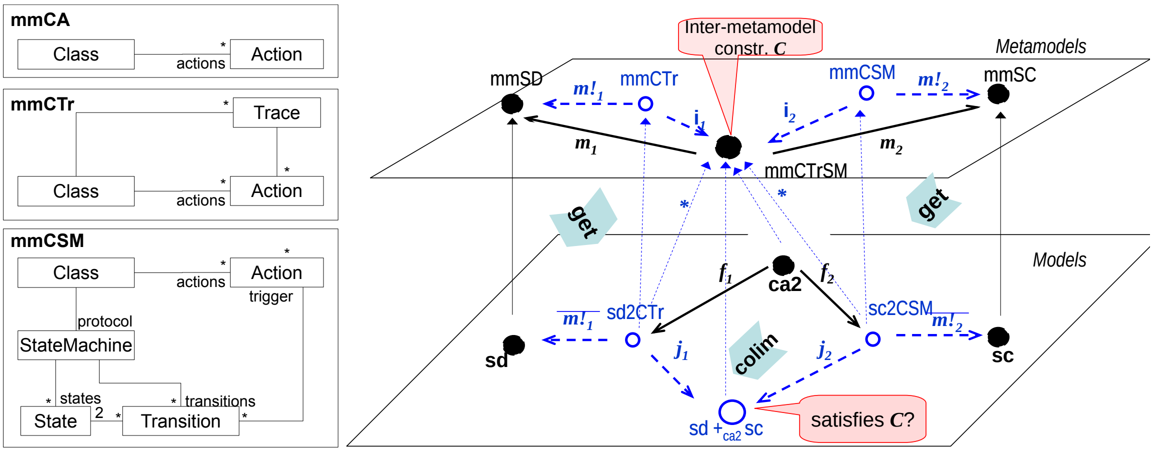

Figure 10: Verifying inter-metamodel constraints

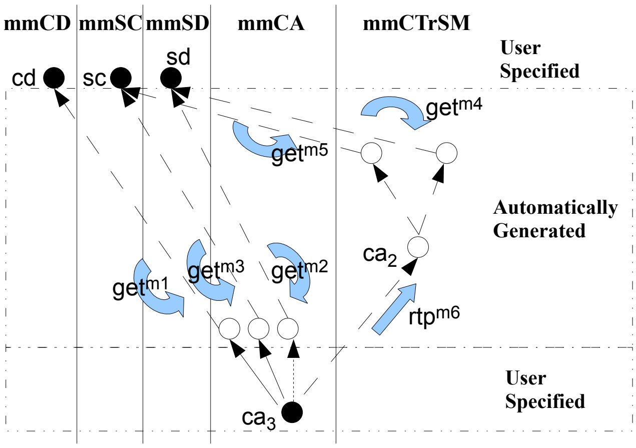

Figure 12: Global consistency checking of the example in Fig. 4

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 preserve metamodel mapping composition (we will formalize these properties in Section 5), we have commutativity for view execution mappings as well:

$$(=)_{\mathsf{get}} \quad \mathsf{get}^{\mathsf{m}4}; \mathsf{get}^{\mathsf{m}6} = \mathsf{get}^{\mathsf{m}2} \text{ and } \mathsf{get}^{\mathsf{m}5}; \mathsf{get}^{\mathsf{m}6} = \mathsf{get}^{\mathsf{m}3}.$$

Hence, we have only one projection of sequence diagram sd to the instance space of mmCA, and only one projection of sc to the same space.

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.

5. MAKING MULTIMODELING PRECISE: A GENERAL FRAMEWORK

The three basic ingredients of our approach are (i) metamodels and their mappings, (ii) models and their mappings, and (iii) a mechanism of model translation from one metamodel to another. We build a (minimal in a sense) mathematical framework allowing to define these concepts and their inter-relations in Section 5.1. In Section 5.2 we show that global consistency checking can be indeed realized in this framework. In Section 5.3 we show how the abstract framework of Section 5.1 can be implemented with constructs close to modeling practice: typed structures, query and constraint languages.

Due to space limitations, the presentation is very brief and semi-formal: we show how the concepts could be formally defined rather than present real formal definitions. We use simple category theory concepts without explanation, and refer to basic concepts of the *institution theory* [14] — an abstract framework for logic and model theory.

5.1 Abstract multimodeling framework

An *abstract multimodeling framework* $\mathcal{F}_{\mathrm{abstr}}$ is a tuple of constructs defined below.

1) A category ***MMod*** whose objects are called *metamodels*

and arrows are *metamodel mappings*.

2) Each metamodel M is assigned with two categories, one being a subcategory of the other, $[\![M]\!] \subset [\![M]\!]^?$. Intuitively, objects of $[\![M]\!]^?$ are structures properly typed over M but perhaps violating M's constraints (hence the question mark); we will call them *structural instances*. Objects of $[\![M]\!]$ are *(legal) models*: structural instances of M satisfying, in addition, all constraints in M.

We require all categories $[\![M]\!]^?$ to be closed under colimits (merging). This is the case for many classes of structures carrying metamodels and models like graphs or attributed graphs. But we do not require this property on $[\![M]\!]$. Our examples above show that in practically interesting situations $[\![M]\!]$ is *not* closed under colimits.

3) Any metamodel mapping $m\colon M \to N$::***MMod*** is assigned with a *getView* functor $\mathsf{get}^m : [\![N]\!] \to [\![M]\!]$ that maps in the opposite direction (think of m as a view definition and get^m as a view execution).

Moreover, if $m = \mathbf{1}_M$ is the identity mapping of metamodel M, then get^m is the identity functor on $[\![M]\!]$, and for two consecutive mappings $M \xrightarrow{m_1} N \xrightarrow{m_2} O$,

$$\mathsf{get}^{m_1;m_2} = \mathsf{get}^{m_2}; \mathsf{get}^{m_1} : [\![O]\!] \to [\![M]\!]$$

(a sequentially composed view definition is executed consecutively).

4) A subcategory ***MMod***$_{\mathsf{inc}} \subset$ ***MMod*** of *inclusion* mappings is fixed: it has the same objects but fewer mappings than ***MMod***. A formal inclusion mapping $\mathsf{i}\colon M \to N$::***MMod***$_{\mathsf{inc}}$ is to be thought of as inclusion of metamodel M into a bigger metamodel N.

Any inclusion $\mathsf{i}\colon M \to N$ is assigned with a *retyping* functor $\mathsf{rtp}^{\mathsf{i}} \colon [\![M]\!]^? \to [\![N]\!]^?$ (think of retyping described in Sections 4.3-4).

Note that in contrast to operation get, rtp maps structural instances (particularly, models) to structural instances (not necessarily models): if even an instance A is an M-model, we cannot guarantee that $\mathsf{rtp}^{\mathsf{i}}(A)$ would satisfy all constraints in N.

Similarly to get, we require $\mathsf{rtp}^{\mathbf{1}_M}$ to be the identity functor on $[\![M]\!]^?$, and for two consecutive mappings m_1, m_2 as above, $\mathsf{rtp}^{m_1;m_2} = \mathsf{rtp}^{m_1}; \mathsf{rtp}^{m_2} : [\![M]\!]^? \to [\![O]\!]^?$.

We will write an abstract multimodeling framework in a short form as a triple $\mathcal{F}_{\rm abstr} = (\boldsymbol{MMod}, \mathsf{get}, \mathsf{rtp})$ assuming that the $[\![\_]\!]$-part of the construction is "included" into get, and the $[\![\_]\!]^?$ and ***MMod***$_{\mathsf{inc}}$parts are "included" into rtp.

Operations get and rtp together provide model translation over partial mappings. A *partial mapping* $m\colon M \rightharpoonup N$ between metamodels (note the semi-arrow head) is, formally, a diagram $M \xleftarrow{\mathsf{i}_m} D_m \xrightarrow{f_m} N$ with $D_m \subset M$ a metamodel called the *domain* of m (while M is the *source* of m), i_m is the corresponding inclusion, and f_m is an ordinary (total) metamodel mapping (the *function* of m). Evidently, sequential composition $\mathsf{get}^{f_m};\mathsf{rtp}^{\mathsf{i}_m}$ provides a functor $[\![M]\!]^? \leftarrow [\![N]\!]^?$ translating N's structural instances and their mappings into M's ones. We will denote this composition by get^m (so that the actual meaning of get^m depends on whether m is a total or a partial mapping).

5.2 Multimodels and their consistency

Let $\mathcal{F}_{\rm abstr} = (\boldsymbol{MMod}, \mathsf{get}, \mathsf{rtp})$ be an abstract multimodeling framework.

A *homogeneous multimodel* over $\mathcal{F}_{\rm abstr}$ is a pair $(M, \mathcal{A})$ with $M \in$ ***MMod*** a metamodel and $\mathcal{A}$ a diagram in $[\![M]\!]$; the latter can be thought of as a family of models together with a system of correspondence spans. A multimodel is *consistent* if colimit $A_\Sigma \stackrel{\rm def}{=} \Sigma\mathcal{A}$ (which always exists in $[\![M]\!]^?$) satisfies M's constraints, i.e., $A_\Sigma \in [\![M]\!]$.

A *heterogeneous multimodel* is a tuple

$$\mathcal{A\!A} = (\mathcal{A}_1{:}M_1 \ldots \mathcal{A}_n{:}M_n)$$

with $M_i \in$ ***MMod*** and $\mathcal{A}_i$ a homogeneous multimodel over M_i, $i = 1..n$. Consistency of a heterogeneous multimodel is much more involved than in the homogeneous situation, and we will begin with a simpler case of *discrete* multimodels, for which each diagram $\mathcal{A}_i$ is actually a set of models without mappings between them.

The algorithm for checking global consistency of a discrete heterogeneous multimodel $\mathcal{A\!A}$ is as follows. We begin with specifying a system of common views (overlaps) between metamodels M_i. For simplicity, we assume that such a system amounts to a set $\mathcal{M}$ of total and partial spans like that one shown in Fig. 11 if we remove mapping m_6 between spans themselves. Global consistency of $\mathcal{A\!A}$ is checked at the heads of these spans. That is, for each span S in $\mathcal{M}$ we perform the following procedure.

Let H be S's head. First, we project to the space $[\![H]\!]^?$ of structural H-instances all models $\mathcal{A}_i$, whose metamodels M_i are reachable from H by the legs of the span. If the span is total, projecting is provided by the view mechanism. If the span is partial, projecting needs both view execution and model retyping as explained above. In this way we obtain a set of instances $\mathcal{A}_H \subset [\![H]\!]^?$.

Second, instances in $\mathcal{A}_H$ are matched by a correspondence diagram $\mathcal{E}_H$ (for example, think of spans $\mathsf{ca2}$ or $\mathsf{ca3}$ in our examples). Note that $\mathcal{E}_H$-data are provided by the user and are, in fact, part of the multimodel's state.

Third, all instances in $\mathcal{A}_H$ are merged modulo the correspondence diagram $\mathcal{E}_H$ into a structural instance

$$(A_\Sigma)_H \stackrel{\rm def}{=} (\Sigma\mathcal{A}_H/\mathcal{E}_H) \in [\![H]\!]^?.$$

Finally, we check whether $(A_\Sigma)_H \in [\![H]\!]$, i.e., whether it satisfies all constraints declared in H.

A general multimodeling case with $\mathcal{A}_i$ being diagrams rather than sets can be treated similarly. The key is that translation operations get and rtp are functors, that is, they translate not only instances but also instance mappings, and hence correspondence diagrams as well. Then the projection $\mathcal{A}_H \subset [\![H]\!]^?$ will be a diagram rather than a set of instances, and diagram $\mathcal{E}_H$ will provide a second level correspondence structure. As colimit operation consumes any sort of input diagrams, the algorithm works well for the general case too.

Another generalization of the algorithm, for which the metamodel schema is more complicated than a set of spans, is harder and is a work in progress.

5.3 Concrete multimodeling framework

In a nutshell, a *concrete multimodeling framework* consists of three components: (i) a *base category* $\mathbb{G}$ of graph-like structures to be thought of as the carriers of metamodels and models, (ii) a *constraint language* $\mathbb{C}$ together with binary relations $\models$ of satisfying a constraint by a model, and (iii) a *query language* $\mathbb{Q}$ together with operations of query execution over a model. In more detail (but still very briefly with many important conditions skipped), a concrete framework is given by the following constructs

1) $\mathbb{G}$-objects are to be thought of as graphs, or many-sorted (colored) graphs, or attributed graphs [11]. The key point is that they are definable by a metametamodel itself being a graph with, perhaps, a set of *equational* constraints. In precise categorical terms, we require $\mathbb{G}$ to be a presheaf topos [3], and hence possessing limits, colimits, and other important properties. We will call $\mathbb{G}$-objects *"graphs"*.

For a "graph" G thought of as a metamodel, an *instance* of G is a pair $A = (D_A, t_A)$ with D_A another "graph" and $t_A\colon D_A \to G$ a mapping (arrow in $\mathbb{G}$) to be thought of as *typing*. An *instance mapping* $f\colon A \to B$ is a "graph" mapping $f\colon D_A \to D_B$ commuting with typing: $f; t_B = t_A$. This defines a category $\llbracket G \rrbracket$ of G-instances.

Any mapping $m\colon G' \to G$ determines a functor $\mathsf{pb}^m\colon \llbracket G \rrbracket \to \llbracket G' \rrbracket$ built with pullback operation in the standard way (see e.g.[15, p.48]).

2) Constraints are defined exactly like in the institution theory. We postulate a functor $\mathbb{C}\colon \mathbb{G} \to \boldsymbol{Sets}$ and a binary relation $\models_G \subset \llbracket G \rrbracket \times \mathbb{C}(G)$ for every "graph" G. For an instance $A \in \llbracket G \rrbracket$ and a constraint $c \in \mathbb{C}(G)$, we write $A \models_G c$ for $(A, c) \in \models_G$.

3) Queries are an original part of the definition. We begin with a functor $\mathbb{Q}\colon \mathbb{G} \to \mathbb{G}$ of *query specifications*. For a "graph" $G \in \mathbb{G}$, the "graph" $\mathbb{Q}(G) \supset G$ is to be thought of as "graph" G augmented with definitions of derived elements. (Actually we require $\mathbb{Q}$ to be a monad [3]). Functor $\mathbb{Q}$ also acts on constraints: for a "graph" G and a set of constraints $C \subset \mathbb{C}(G)$ over G, there is a set $\mathbb{Q}(C) \subset \mathbb{C}(\mathbb{Q}(G))$ of constraints derived from C.

Semantics of query specifications is given by an operation $\llbracket \mathbb{Q} \rrbracket$ that maps G-instances to $\mathbb{Q}(G)$-instances as specified by the inset diagram on the right (two derived arrows are dashed and the derived node is underlined). We require this diagram to be a pullback square, which means that "graph" D_A is the inverse image of "graph" $D_{\llbracket \mathbb{Q} \rrbracket(A)}$, that is, the original data are not changed by the query execution.

$$\begin{tikzcd} D_A \arrow[r, hook, dashed] \arrow[d, "t_A"'] & \underline{D_{\llbracket \mathbb{Q} \rrbracket(A)}} \arrow[d, dashed, "t_{\llbracket \mathbb{Q} \rrbracket(A)}"'] \\ G \arrow[r, hook] & \mathbb{Q}(G) \end{tikzcd}$$

To ensure that derived instances satisfy derived constraints, we require the following to hold for any instance A:

$$\text{(QC)} \qquad A \models_G C \text{ implies } \llbracket \mathbb{Q} \rrbracket(A) \models_{\mathbb{Q}(G)} \mathbb{Q}(C).$$

Finally, we requite operation $\llbracket \mathbb{Q} \rrbracket$ to act also on instance mappings: for any injective arrow $f\colon A \to B$ in $\llbracket G \rrbracket$, there is defined an injective arrow $\llbracket \mathbb{Q} \rrbracket f\colon \llbracket \mathbb{Q} \rrbracket(A) \to \llbracket \mathbb{Q} \rrbracket(B)$ in $\llbracket \mathbb{Q}(G) \rrbracket$. In the database literature, this property of a query language is called *monotonicity*, and it is known that queries without negation are monotonic [18].

From these data we can derive an abstract framework $\mathcal{F}_{\text{abstr}}$ along the following lines. We first fix a subcategory $\mathbb{G}^\circ \subset \mathbb{G}$ of finite "graphs" to be the carriers of metamodels. A *metamodel* is a pair $M = (G_M, C_M)$ with $G_M \in \mathbb{G}^\circ$ a carrier graph and $C_M \subset \mathbb{C}(G_M)$ a set of constraints. Structural instances of M are instances of G_M, i.e., $\llbracket M \rrbracket^? \stackrel{\text{def}}{=} \llbracket G_M \rrbracket$, and models of M are G_M's instances satisfying C_M.

Metamodel *mappings* are $\mathbb{G}$-arrows of the form $m\colon G_M \to \mathbb{Q}(G_N)$ (Kleisli arrows of monad $\mathbb{Q}$), which are compatible with constraints: $\mathbb{C}(m)(C_M) \subset \mathbb{Q}(C_N)$. Any such mapping determines a functor $\mathsf{get}^m \stackrel{\text{def}}{=} \llbracket \mathbb{Q} \rrbracket; \mathsf{pb}^m\colon \llbracket N \rrbracket \to \llbracket M \rrbracket$, which satisfies conditions postulated in the definition of the abstract framework. The retyping functors $\mathtt{rtp}$ are defined by composition (like in the example of Section 4.3) and also satisfy necessary conditions. With accurate formal definitions, it can be proved that every concrete multimodeling framework gives rise to an abstract multimodeling framework. Hence, the algorithm of global consistency checking can be used with a concrete framework as well.

6. RELATED WORK

Specifying overlaps of homogeneous models by correspondence spans is known for a long time [13, 5, 4, 17]. Close relations between consistency checking and model merging were noticed in [7] for behavioral, and in [22] for structural models. A large body of work in this direction was done in databases in the context of view integration, where they worked mainly with ER-diagrams [23]; a generalization for a much more expressive graph-based language was developed in [5]. A serious limitation of these approaches was that they work for the homogeneous case only because it was unclear how to merge heterogeneous models.

Consistency of *heterogeneous* models is a central issue of the *living with inconsistency* frameworks [20, 24, 19, 10]. Their basic idea is to translate all models and constraints into a common logical formalism, and check if there are conflicts between logical formulas. Although these approaches handle many cases in heterogeneous multimodeling, the configuration of model overlap (which may be very intricate as our examples show) is flattened and hidden in arrays of formulas. As a result, none of the approaches fully covers heterogeneous multimodeling: they mainly handle well-defined cases where elements are matched by names, or only pairwise cases. In contrast, the structure of inter-model relationships is made visible and essentially used in our approach.

Several approaches also transform models to aid model merging and consistency management. Egyed [8] proposes a flexible framework based on model transformation and mapping; however, it is the user's responsibility to use them correctly. Ehrig et al. [12] use graph transformation to derive views from a reference model, and integrate modified views using colimit. Compared to our work, their work requires users to define the transformation manually. Jurack and Taentzer [16] consider multimodeling (they say *composite models*) in a distributed environment. Their setting is mainly operational and is based on graph transformations. None of the approaches handles inconsistent views.

Many researchers focus on discovering traceability links between heterogeneous models [2] and discovering differences between homogeneous models [1]. Their results can be integrated into our approach as a means for an automated construction of correspondence spans.

7. CONCLUSION

The paper describes a general approach to global consistency checking of heterogeneous multimodels. The approach 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. We have shown that type-safe matching, indirect model overlap, and inter-metamodel constraints can be uniformly managed along the lines described. 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 formulated in Section 5. Third is the adaptability of the framework to the *living with inconsistencies* paradigm: 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 (and hard to prove) 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 *direct* global consistency checking. By the latter we mean merging all metamodels into one global metamodel MM, then all partial models becomes partial instances of MM, whose joint consistency can be checked by a homogeneous CCM-algorithm. There are strong formal arguments justifying this assumption but an accurate formal proof is still to be complete.

An 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 along the lines described in the paper would allow useful interaction of the two fields.

8. REFERENCES

[1] M. Alanen and I. Porres. Difference and union of models. In *UML*, pages 2–17, 2003.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[3] M. Barr and C. Wells. *Category theory for computing science*. Prentice Hall, 1995.

[4] P. Bernstein and R.Pottinger. Merging models based on given correspondences. In *VLDB*, 2003.

[5] B. Cadish and Z. Diskin. Heterogenious view integration via sketches and equations. In *ISMIS*, pages 603–612, 1996.

[6] Z. Diskin. Model synchronization, mappings, tile algebra, and categories. In *GTTSE'09*. Springer. To appear.

[7] S. M. Easterbrook and M. Chechik. A framework for multi-valued reasoning over inconsistent viewpoints. In *ICSE*, pages 411–420, 2001.

[8] A. Egyed. *Heterogeneous view integration and its automation*. PhD thesis, University of Southern California, 2000.

[9] A. Egyed. Instant consistency checking for the UML. In *ICSE*, pages 381–390, 2006.

[10] A. Egyed. Fixing inconsistencies in UML design models. In *ICSE*, pages 292–301, 2007.

[11] H. Ehrig, K. Ehrig, U. Prange, and G. Taenzer. *Fundamentals of Algebraic Graph Transformation*. 2006.

[12] H. Ehrig, R. Heckel, G. Taentzer, and G. Engels. A combined reference model- and view-based approach to system specification. *Int. Journal of Software and Knowledge Engeneering*, 7:457–477, 1997.

[13] J. L. Fiadeiro and T. S. E. Maibaum. Interconnecting formalisms: Supporting modularity, reuse and incrementality. In *SIGSOFT FSE*, pages 72–80, 1995.

[14] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of ACM*, 39(1):95–146, 1992.

[15] B. Jacobs. *Categorical logic and type theory*. Elsevier Science Publishers, 1999.

[16] S. Jurack and G. Taentzer. Towards composite model transformations using distributed graph transformation concepts. In *MoDELS*, pages 226–240, 2009.

[17] H. Liang, Z. Diskin, J. Dingel, and E. Posse. A general approach for scenario integration. In *MoDELS*, pages 204–218, 2008.

[18] H. Liefke and S. Davidson. View maintenance for hierarchical semistructured data. In *DaWaK*, pages 114–125, 2000.

[19] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *ICSE*, pages 455–464, 2003.

[20] B. Nuseibeh, J. Kramer, and A. Finkelstein. Viewpoints: meaningful relationships are difficult! In *ICSE*, pages 676–683, 2003.

[21] M. Sabetzadeh and S. M. Easterbrook. View merging in the presence of incompleteness and inconsistency. *Requir. Eng.*, 11(3):174–193, 2006.

[22] M. Sabetzadeh, S. Nejati, S. Liaskos, S. M. Easterbrook, and M. Chechik. Consistency checking of conceptual models via model merging. In *RE*, pages 221–230. IEEE, 2007.

[23] S. Spaccapietra and C. Parent. View integration: A step forward in solving structural conflicts. *IEEE Trans. Knowl. Data Eng.*, 6(2):258–274, 1994.

[24] R. V. D. Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logic to maintain consistency between UML Models. In *UML*, pages 326–340, 2003.

[25] J. Warmer and A. Kleppe. *The Object Constraint Language. Precise modeling with UML*. Addison-Wesley, 2000.