Technical Report No. 2006-517
Scenario Integration via Higher-Order Graphs*

Zinovy Diskin, Juergen Dingel, and Hongzhi Liang

School of Computing, Queen's University,
Kingston, Ontario, Canada K7L 3N6
{zdiskin, dingel, liang}@cs.queensu.ca

Abstract. Requirements engineering (RE) is an inherently piecemeal and dynamic process. Arrangement, correlation and integration of models presenting different views of the system are an important component of the requirements engineer's work. If manually performed, these operations are error-prone and time consuming and, thus, an integrated computer-aided environment for them would be very useful. In the paper, we present an extendable framework that provides a formalization and a generic integration pattern for scenario management. The framework is based on mathematical category theory machinery of algebraic operations with higher-order graphs.
We demonstrate how the framework works by a (not entirely trivial) example of scenario integration. A distinctive feature of our integration pattern is an essential use of derived elements: for setting correspondences between views we augment view scenarios with derived executions. We show that this augmentation is essential for a proper integration: information explicitly specified in one view can be implicit in another view, where it is derived from the information considered basic in that view.

1 Introduction

Requirements engineering (RE) is inherently piecemeal: normally it results in a diverse set of models capturing different parts of the system, and different views of the same part provided by different groups of users. RE is also a dynamic process, where the requirement engineer often needs to rearrange the models, integrate them into bigger fragments and discuss them with the users, separate pieces of these bigger models that need reworking and thus create new models, then integrate them with the old ones and so on until a sufficiently coherent set of models is built.

Our long term project aims at developing an integrated environment where such model management (MMt) tasks could be performed in an intelligent way. Clearly, as for many other problems related to MMt, an immediate code-centric programming effort would be inefficient and error-prone [2]. We must begin with

* This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Communications and Information Technology Ontario (CITO), and IBM Software Lab in Ottawa, Canada

precise specifications of *what* are the operations to be performed with models, and then proceed to *how* they can be implemented in an efficient way. It would be very helpful to have a formal specification framework for RE-model manipulations, first of all, for model integration – the most important operation among them.

A general pattern is as follows. Let $\mathbf{M} = \{M_1 ... M_n\}$ be a set of RE-models (perhaps, of different types - UML sequence or communication diagrams [17], message sequence charts (MSCs) [13], use case maps [5]). To integrate them, we need to encode them in a suitable formalism $\mathcal{F}$ by some procedure f, and obtain a set of formal constructs $\mathbf{F} = \{F_1 ... F_n\}$, $F_i = f(M_i)$. The latter are amenable to formal manipulations, and we can perform with them various operations, particularly, integration: $F_\Sigma = F_1 \oplus ... \oplus F_n$. Finally, we come back to the RE-modeling language, $M_\Sigma = f^{-1}(F_\Sigma)$ and consider M_Σ to be the result of the integration (or other operation) over the models M_i.

A fundamental problem of model integration is that different models representing different views of the same universe can essentially overlap in different ways. A proper integration has to take this into account, otherwise the result will *implicitly* contain duplications and redundancies. The question is what should be specified in addition to the set of models $\mathbf{M}$ so that their integration would merge all the information contained in the views without loss and duplication. Moreover, we need to have a generic pattern for model overlap not dependant on peculiarities of a particular model language. Then we could build a generic pattern for the entire view/model integration operation.

The data/structure modeling sides of the view integration problems have been studied for many years in the field of semantic data modeling (in-between AI and database design) and have an extensive literature, see [3] for a recent attempt with many references to the literature; recently it gained a new stimulus in the context of ontology engineering due to Semantic Web flourishing. However, as a rule, these works heavily depend on the peculiarities of the particular data models they employ and hence could not be used for our goals. A rare exclusion is a recent paper [19], which considers a generic machinery very close to ours (more details are in sect. 6).

Behavior view integration problem in general, and scenario integration in particular, has gained much less attention in the literature. We can mention just few theoretical works devoted immediately to scenario integration, [14, 7], and a few indirectly related, [20, 21, 12, 22], which consider composition of state machines generated (synthesized) from scenarios (see more in sect.6). These works differ in their generality and scope but share basically the same approach to formalization: behaviors/scenarios are encoded in terms of relational structures and the ordinary logical syntax based on formulas (strings of symbols). Beyond the scope of toy examples, it results in a bulky formalization difficult to work with, particularly, the notion of model mapping crucial for model management [2, 10] becomes heavy to operate. Moreover, the transparency of diagrammatic scenario specifications is lost and the formal theory begins to live in its own isolated world.

We would like to stress that the choice of a suitable formal framework, $\mathcal{F}$ in our notation, is very important for the entire goal of setting MMt on a proper specificational foundation. Here is a list of the most essential requirements:

1. $\mathcal{F}$ should be universal enough so that a sufficiently rich set of RE-models could be encoded;
2. $\mathcal{F}$-constructs should be amenable for effective manipulations and easy to work with;
3. $\mathcal{F}$ should be *natural* for RE-models: well readable and close to the original RE-models, so that the "outline" of model M would be somehow seen in its formalization $f(M)$.

The last requirement is especially important for RE applications due to the interactive and dynamic nature of RE. We can hardly hope that view integration would consist of only two steps: firstly, all requirements, business rules and overlapping information is figured out and specified, then the merge procedure runs and returns the result of integration. In reality, model development and integration on one hand, and gathering requirements on the other, are tightly interweaved; even the very model merge procedure should be semi-automatic and intertwined with modeler's inputs rather than be entirely automatic.

Since an overwhelming majority of RE modeling languages is diagrammatic, the choice of formalization based on graph-based structures seems to be almost inevitable. Fortunately, the graph-based formalisms possess other properties highly desirable in our context: they are (i) amenable to effective algebraic manipulations, (ii) extremely expressive and provide a base for really generic specifications, (iii) have a solid theoretical support provided by mathematical *category theory* and a large body of work in graph rewriting, (iv) have a tool support. Unfortunately, despite this unique combination of properties making graph-based formal structures uniquely suited for RE, and MMt on a whole, the power and a proper mathematical treatment of such structures are still not familiar to the community.

An immediate goal of this paper is to demonstrate how naturally typed graphs and similar structures appear in an accurate formalization of scenario specifications; particularly, how close they are to UML2 sequence and communication diagrams. We also describe a generic procedure of typed graph merge (well known in category theory but not in the RE literature), and show how naturally the corresponding patterns appear in scenario integration. We present the framework in a easily extendable way, so that ordinary typed graphs, 2-typed graphs (i.e., graphs typed over typed graphs), 3-typed graphs (graphs typed over 2-typed graphs) and so on can be uniformly treated and merged. In addition, graphs (nodes and edges between nodes), 2-graphs (which in addition to edges between nodes have edges between edges, we will call them *2-edges*), 3-graphs (with edges between 2-edges) and so on are also captured "for free". We can well consider k-typed n-graphs (i.e., n-graphs typed over $(k-1)$-typed n-graphs) without, in fact, any essential complications of the framework and merge algo-

rithms.[1] We use the term *higher-order graphs* to refer to this sort of graph-based structures in a generic way.

Specifically, we present a formalization of some core of UML2 sequence diagrams based on 2-typed graphs (sect. 2, and argue that 2-typed 2-graphs are necessary for more adequate formalization, sect. 5). We do not claim that this formalization is adequate to the full power of sequence diagrams (though our core is much more expressive than, say, basic MSCs). Our goal is to demonstrate how naturally this core can be formalized with higher-order graphs, and how effectively this formalization works in scenario integration. To this end, we consider a not entirely trivial example of scenario integration (sect. 4.1,4.2), where we glue an integrated scenario from different pieces of the views (component scenarios). This topological interpretation explains the essence of the merge procedure, but the latter is actually based on a sequence of formal algebraic manipulations with elements from which the scenario graphs are built (sect. 3). Viewing algebraic manipulations in an abstract and well-modularized way is one of the main benefits that category theory could bring to the subject. We then abstract our example to a general pattern of view integration (sect. 4.3).

A distinctive feature of our integration pattern is an essential use of derived elements: for proper integration we augment view scenarios with derived executions. We show that this augmentation is in the very heart of the integration procedure: information explicitly specified in one view can be implicit in another view, where it is derived from the information considered basic in that view.

Sections 5 and 6 provide a general discussion of the approach and its relation to other works.

2 Sequence diagrams/scenarios via typed graphs

In general terms, a scenario is a record of possible message exchanges between communicating objects. Figure 1 presents an example: a Sale scenario specified by a UML2 sequence diagram [17]. The meaning of the diagram is hopefully evident from the message names. Our, and any other, scenario has a structural base: a set of the interacting objects and (implicitly) the types of messages they can exchange. The set of objects is explicitly specified in the boxes on the top of the lifelines. As for messages, in the syntax of sequence diagrams, the modeler just names the messages but, in fact, some typing is implicit there. For example, the messages *initialOffer*(x_0) and *counterOffer*(x_1) are reasonable to consider as two different *occurrences* of the same message type *offer*(x) between two instances of class *Agent*. Similarly, we can have multiple occurrences of messages *setPrice* and *getPrice* (but in the scenario depicted in Fig. 1 they appear only once).

[1] We are, thus, preparing to live in the world dominated by UML5 and XML3 taught in elementary schools :)

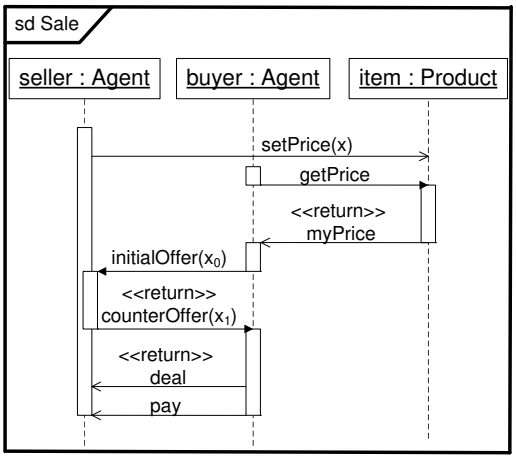

Fig. 1. A sample sequence diagram

The graph, whose nodes are objects participating in the interaction and edges are message types, is shown in the middle cell of Fig. 2.[2] Basically, this graph is what is called a collaboration diagram in UML2, and we will call it *collaboration graph*. Note that this graph is itself typed by class and channel names, which can also be organized into a graph shown in the bottom cell of Fig. 2. We will call the latter graph the *collaboration base*.

Thus, a sequence diagram can be presented as a chain of two typing mappings $G_2 \stackrel{\tau_1}{\to} G_1 \stackrel{\tau_0}{\to} G_0$ between three graphs as shown in Fig. 2. This compact presentation would be enough to capture basic message sequence charts (MSCs), where graph G_2 is just a partial order of event occurrences and graph G_0 consists of one node (objects are not distributed over classes) and one loop (universal channel for all message types).

However, UML sequence diagrams add to MSCs more than just distribution over classes. They allow us

(i) to have nested executions over the same lifeline and, hence, concurrency not only between objects but internally within the same object lifeline too (see the seller's lifeline in Fig. 1),

(ii) to distinguish between synchronous and asynchronous messages (denoted by, respectively triangle and angle arrow heads), and

(iii) to specify control structures immediately in sequence diagrams rather than in high-level diagrams like HMSCs. We do not consider the capability (iii) in the paper; as for (i) and (ii), they can be captured in the typed graph formalism in an intuitively appealing and transparent way as we are going to show.

[2] More accurately, *buyer* and *seller* are roles (formal parameters in the interaction) that real objects could play. To simplify wording, we will call them objects when it will not lead to confusion.

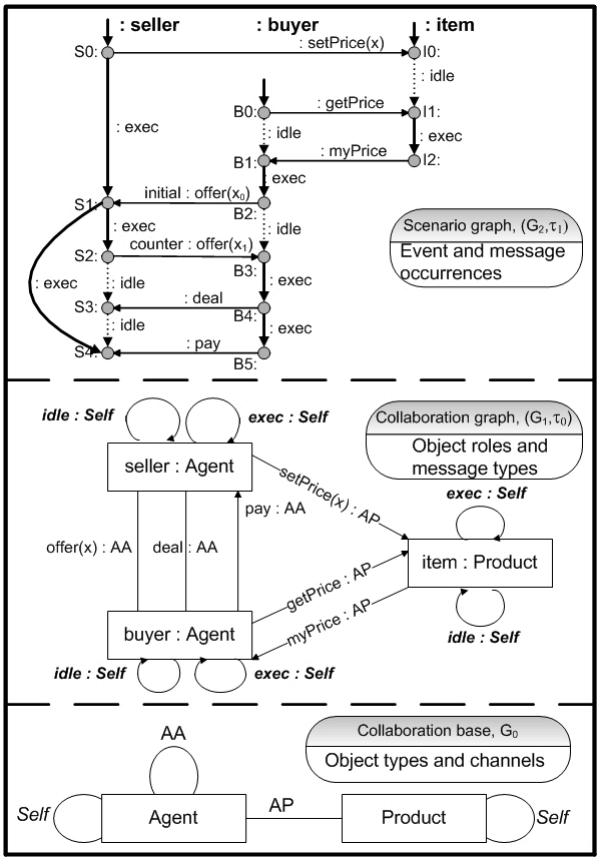

Fig. 2. Sequence diagrams via typed graphs

First, we introduce for each node in the collaboration base graph G_0 a loop arrow called ***Self***, where the bold font indicates that this is a constant rather than a user defined item in the graph. Then we introduce for each node in the collaboration graph G_1 two constant loop arrows ***idle*** and ***exec*** labeled by ***Self***. Note, if an arrow in the occurrence graph G_2 is labeled by ***idle*** or ***exec***, then because the latter are loops, such an arrow necessarily goes along the lifeline of some object (namely, that one to which the loop in G_1 is attached) rather than between the lifelines. Intuitively, labeling an arrow by ***idle*** or ***exec*** means, respectively, that in the time period between the source and target event occurrences, the object is idling or executing some procedure. In the latter case, this is the procedure triggered by the message coming to the source event. To ease distinguishing between ***idle*** and ***exec***-typed arrows, we use the following concrete syntax for them: the former are shown with dotted and the latter with bold lines.

The difference between synchronous and asynchronous messages is captured by whether the next piece of the sender's lifeline is an ***idle***- or ***exec***-arrow.

The following definitions presents our descriptions in a formal way.

Definition 1 (Graphs and their morphisms) *A* (directed multi)graph *is a quadruple* $G = (N, E, \mathbf{so}, \mathbf{ta})$ *with* N, E *sets of* nodes *and* edges *(or* arrows*) respectively, and* $\mathbf{so}$,$\mathbf{ta}$ *mappings from edges to nodes (giving, respectively, the* source *and the* target *of an edge). We also write* $e\colon x \to y$ *when* $e.\mathbf{so} = x$ *and* $e.\mathbf{ta} = y$*. An edge* e *is called a* loop *if* $e.\mathbf{so} = e.\mathbf{ta}$*. Saying "*$e$ *is an element of* G*" means* $e \in N \cup E$*.*

A graph morphism *or* graph mapping $h\colon G_1 \to G_2$ *is a pair of set mappings* $h_N\colon N_1 \to N_2$*,* $h_E\colon E_1 \to E_2$ *compatible with* $\mathbf{so}$ *and* $\mathbf{ta}$ *mappings:* $e.h_E.\mathbf{so}_2 = e.\mathbf{so}_1.h_N$ *and* $e.h_E\mathbf{ta}_2 = e.\mathbf{ta}_1.h_E$ *for any* G_1*-edge* e*.*

Definition 2 (Reflexive graphs) *A* double reflexive *graph is a graph with two additional mappings* **idle** *and* **exec** *from nodes to edges as shown in Fig. 3(a1) (in UML terms, the latter is the metamodel of reflexive graphs). In addition, for any node* $x \in N$*,* $x.\mathbf{idle}.\mathbf{so} = x = x.\mathbf{idle}.\mathbf{ta}$ *and similarly for* **exec***. In other words,* **idle** *and* **exec** *are specially designated loops assigned to every node.*

A reflexive *graph is a double reflexive graph for which* **idle** *and* **exec** *coincide. The only special loop in this case is often called* ***Identity*** *or* ***Self****. By some abuse of terminology, we will also use the term "reflexive" generically for both types of reflexive graphs.*

A reflexive graph morphism $h\colon G_1 \to G_2$ *is a graph morphism compatible with looping in the following way: for any node* x *in graph* G_1*,* $x.\mathbf{idle}_1.h_E = x.h_N.\mathbf{idle}_2$ *but*
$x.\mathbf{exec}_1.h_E \in \{x.h_N.\mathbf{idle}_2, x.h_N.\mathbf{exec}_2\}$ *(see Fig. 3(a2) for the metamodel).*

The latter condition is well-known for labeled transition systems morphisms ([23]). It means that an ***exec***-transition (procedure) in one system can be simulated by ***idle***-transition in another system.

Definition 3 (Typed graphs) *We say that a (reflexive) graph* G *is* typed *if there is a (reflexive) graph morphism* $\tau\colon G \to G_\tau$*. The target is called the* type *graph or* base *and the very mapping is called* typing or labeling*. We will also say that the graph* G *is* typed over G_τ*.*

Given two typed graphs over the same base, $\tau_i\colon G_i \to G_\tau$*,* $i = 1, 2$*, their* morphism *is a graph morphism* $h\colon G_1 \to G_2$ *commuting with typing:* $e.h.\tau_2 = e.\tau_1$ *for any element* e *in graph* G_1*.*

Definition 4 (Collaboration and Scenario graphs) *(i) A* collaboration graph *is a double reflexive typed graph,* $\tau_0\colon G_1 \to G_0$*, whose nodes are called* objects *or* instances*, edges are* message types *and the two special loops are called* **idle** *and* ***exec****. The type base is a reflexive graph called* collaboration base*, whose nodes are* classes *and edges are* (communication) channels*. The special loop is called* ***Self****.*

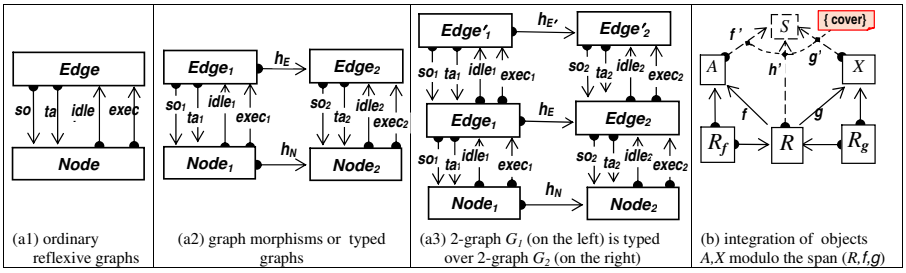

Fig. 3. Metamodels for (reflexive) graphs (a1), typed graphs or graph mappings (a2), typed 2-graphs (a3), model integration (b). In (a1..a3) rectangles denote sets and arrows are set mappings. In (b) rectangles are objects (sets, graphs, higher-order graphs) and arrows are their mappings; bold/ordinary arrow tails denote totally/partially defined mappings.

(ii) A scenario graph *is a graph typed over a collaboration graph,* $\tau_1 : G_2 \to G_1$ *with* $\tau_0 : G_1 \to G_0$ *as above. Nodes and arrows of* G_2 *are called, respectively, event and message* occurrences*. Thus, a scenario graph is a three-element chain of graph mappings* $G_2 \overset{\tau_1}{\to} G_1 \overset{\tau_0}{\to} G_0$ *similarly to what we have for sequence diagrams.*

(iii) The notions of morphisms between collaboration and scenario graphs are defined in an evident way.

Later we will also need the following notions.

Definition 5 (2-Graphs and their morphisms) *A* 2-graph *is a tuple* $G = (N, E, \mathbf{so}, \mathbf{ta}, E', \mathbf{so}', \mathbf{ta}')$ *with* N, E, E' *sets of* nodes, edges *and* prime- or two-edges (or arrows)*, and* $\mathbf{so}, \mathbf{ta}, \mathbf{so}', \mathbf{ta}'$ *are the* source *and* target *mappings, respectively, for edges and 2-edges. We write* $e : x \to y$ *when* $e.\mathbf{so}' = x \in Edge$ *and* $e.\mathbf{ta}' = y \in Edge$ *for a 2-edge* e*. Saying "*e *is an element of* G*" means* $e \in N \cup E \cup E'$*.*

A mapping of 2-graphs *is a triple of mappings* $h = (h_N, h_E, h_{E'})$ *between the corresponding sets, which are compatible (commuting) with the source and target mappings in the evident way (see metamodel (a3) in Fig. 3).*

Clearly, we can define the notions of *n-graph* and *n-graph mapping* in a similar way, thus coming to the notion of *typed n-graph*. In this way, by extending inductively the metamodel (a3) in Fig. 3 with sets $Edge''$, $Edge'''$ and so on upward, and with mappings $G_2 \overset{h_2}{\to} G_3 \overset{h_3}{\to} G_4 \to \ldots$ and so on to the right, we come to the notion of *k-typed n-graph*, or *higher-order graph*.

3 The machinery of merge

An explanation of the typed graph merge operation can be found in many sources, for example, [19] for a very transparent exposition. However, to make the

paper more self-contained, and to highlight some additional aspects we consider important in our context, we will briefly consider the essence of the machinery. Typed graph merge is based on graph merge, which in its turn is based on set merge. That is why we begin with careful consideration of what is set merge.

3.1 Setting correspondences between sets

Suppose we have two sets $A = \{a, b, c\}$ and $X = \{x, y, z\}$ considered as *schemas*, that is, sets whose elements are considered as names for some "real world" entities – denotations of the names (these entities could be objects, or events, or messages, or sets of these, whatever that could have a name). We will write $[\![n]\!]$ for the denotation of name n. We can always restrict the scope of these names to the containing schema (e.g., by qualifying them with the schema name, say, $b{::}A$, $z{::}X$ *etc*), and hence consider the schema-sets to be disjoint. If there is no information about inter-relations between denotations, then the integration (merge) of these sets is nothing but their disjoint union $I = A \biguplus X \stackrel{\text{def}}{=} \{a, b, c, x, y, z\}$.

Suppose we know that elements $a \in A$ and $x \in X$ actually refer to the same "real world" element, $[\![a]\!] = [\![x]\!]$, and similarly $[\![c]\!] = [\![z]\!]$. The question is how to specify these fact syntactically, without reference to semantics. Informally, we need to set equivalences between the names: $a \sim x$ and $c \sim z$. In fact, each such an equivalence is just a pair of names (n, m) taken from sets A and X respectively. A set of such equivalences is then a set R together with two *projection* mappings, $f\colon R \to A$ and $g\colon R \to X$, so that if for an element $r \in R$ we have $r.f = a$ and $r.g = x$, it can be seen as having an equivalence $a \sim x$. Thus, formally, interconnection between two schema-sets is specified by an *arrow span*: a set R with two projection mappings as above. In our case, $R = \{p, r\}$ and projections are given by equalities:

$$p.f = a,\ r.f = c,\ p.g = x,\ r.g = c.$$

We will also call projections the *arms* of the span, the sets A, X its *hands* and the set R is the *head*.

The process of discovering the same denotations of different names can be rather complicated, it needs a special investigation and may involve various heuristic techniques studied in the domain of *schema matching* (see [18] for a survey). In this paper, we consider schema matching as a black box procedure that returns a span connecting the schemas to be integrated. What is important for us is that the schema matching investigation often reveals new entities related to the subject matter yet not named in either of the component schemas. Suppose, for example, that in our domain specified by schemas A and X, we have discovered some new object Q named in neither A nor X. Then it is reasonable to enter a name for this object, say, q, into the head of the interconnection span and set $R = \{p, q, r\}$. We have the same span configuration as above but now the arms are partially defined mappings. Total mappings can be retained if we replace the arms by their graphs (extensions) R_f, R_g as shown in Fig. 3(b). The entire configuration is then a couple of spans (R_f, R_g) sharing the common hand R, and it lives in the universe of sets and total mappings.

3.2 Merging sets

Integration of sets A and X with their interconnecting span $\mathbf{R}$ taken into account or, as we will say, integration *modulo* the span $\mathbf{R}$, $S = A \bigoplus_{\mathbf{R}} X$, is performed as follows. We first take the disjoint union of all participating sets $S_0 = A \biguplus X \biguplus R$, and then identify those elements which are declared to be "the same" by mappings f, g. In more detail, we first compose a binary relation E over S_0, $E \stackrel{\text{def}}{=} \{(p, a), (r, c), (p, x), (r, z)\}$. Then we generate the least equivalence relation E^* containing E and, finally, take the factor set (the partition) S_0/E^* of S_0 by this equivalence. In our case, the partition is a five element set

$$P_{E^*} = \{\{a, p, x\}, b, \{c, r, z\}, y, q\}$$

(the reader can easily recognize our two $\sim$-equivalences here). It remains to agree how to name those denotations which are named differently in different component schemas. A reasonable agreement is to set the priority of the interconnecting schema and to name the multi-named objects by their names from R. Thus, the factor set is the five element set $S = S_0/E^* = \{p, b, r, y, q\}$. Together with evident mappings $f' \colon A \to S$, $g' \colon X \to S$ and $h' \colon R \to S$ it forms the result of integration.

We emphasize that the actual result of integration is an arrow configuration shown in Fig. 3(b) in dashed lines rather than just its head S. The mappings f', g', h' are important and allow us to trace how the component schemas are represented in the merge. Note that these mappings are totally defined (which is shown by their bold tails) and, hence, nothing is lost in the integration. Moreover, the triple of mappings (f', g', h') *jointly covers* their common target, and hence nothing extra is acquired. The set S is the "least upper bound" of sets A, X modulo span $\mathbf{R}$ in some lattice of set information contents.

Evidently, the property of "nothing lost and nothing extra" is in the very heart of the integration procedure, and it is necessary to have its precise formal explication. For sets, totality of primed mappings together with their jointly covering property do the job. Unfortunately, the notion of covering is hard to reformulate for structures other than sets in a a way suitable for our context ("covering = nothing extra"), and we need to look for another formalization. A fundamental approach to handle such issues was found in category theory under the name of *universal properties*.

Let us call a family of arrow with a common target a *cospan*. The merge procedure is then can be phrased as building a special cospan $\mathbf{S} = (S, f', g', h')$ over the input span $\mathbf{R}$ to form a commutative diamond (see Fig. 3(b)). This cospan possesses the following remarkable property: let $\mathbf{S}' = (S', f'', g'', h'')$ be any other cospan similar to $\mathbf{S}$, which also makes with $\mathbf{R}$ a commutative diamond. Then there is a unique mapping $! \colon S \to S'$ making all the triangle diagrams commutative. If we treat a mapping between sets as a sort of information embedding, then an arbitrary cospan (S', f'', g'', h'') over $\mathbf{R}$ can be seen as an information upper bound of $\mathbf{R}$ while the merge cospan is the *least* information upper bound. In categorical terms, the latter is called *colimit* or else *amalgamated sum*.

The procedure we have just considered can be immediately generalized for the case of multiple sets to be merged modulo multiple interconnections between them. Moreover, a general pattern for merge is not restricted by interconnections of the span shape. In fact, any collection of sets A_i, $i = 1..m$, together with any collection of mappings between them $f_j, j = 1..k$, can be merged (integrated) into the least upper bound set S with canonic embeddings of the component $\iota' : A_i \to S$. The algorithm is described in Fig. 4. We will sometimes call a general configuration $(\mathbf{A}, \mathbf{f})$ with $\mathbf{A} = (A_i, i = 1..m)$, $\mathbf{f} = (f_j, j = 1..k)$ a *(generalized) span* int he universe of sets and set mappings.

SET MERGE$(A_1, \dots, A_m, f_1, \dots, f_k)$
- Let $A = \biguplus_{i=1..m} A_i$ and $\iota_i : A_i \to A$ be canonic inclusions;
- Let $E = \emptyset \subset A \times A$;
- For every mapping f_j, $j = 1...k$, **do**
  - For every element a in the domain of f_j, **do**
    $E := E \cup \{(a, f_j(a))\}$ **od od**
- Set E^* = reflexive symmetric transitive closure of E;
- Set $\varepsilon : A \to A/E^*$ be the canonic surjection;
- Return $S = A/E^*$ and $\iota'_i = \iota_i \mathbin{⨾} \varepsilon : A_i \to S$, $i = 1...m$.

Fig. 4. General set merge procedure

3.3 Merging graphs, typed graphs and higher-order graphs

Suppose we have a generalized span $\mathbf{R} = (\mathbf{G}, \mathbf{f})$ in the universe of graphs and graph mappings, that is, a configuration of graphs $\mathbf{G} = (G_i, i = 1..m)$ and mappings between them $\mathbf{f} = (f_j, j = 1..k)$. In means, in fact, that we have two similar configurations of sets (two generalized spans) $\mathbf{R}_N = (G_{iN}, f_{jN})$, and $\mathbf{R}_E = (G_{iE}, f_{jE})$, $i = 1..m, j = 1..k$, connected, in addition by the family of "vertical" (w.r.t. Fig. 3(a1)) mappings $\boldsymbol{so}_i, \boldsymbol{ta}_i$, $i = 1..m$. It follows from general category theory results that any span of graphs has the least upper bound (in the sense of the universal property we considered), which can be built as follows.

We first merge the spans of nodes and edges separately, thus getting sets S_N and S_E together with the respective inclusion mappings making them cospans, $\iota'_{iN} : G_{iN} \to S_N$ and $\iota'_{iE} : G_{iE} \to S_E$, $i = 1..m$. Now we need to relate them between themselves. Take a "vertical" mapping in the graph metamodel Fig. 3(a), say, $\boldsymbol{so}$. For each of the input graphs G_i we have a set mapping $\boldsymbol{so}_i \mathbin{⨾} \iota'_{iN} : G_{iE} \to G_{iN} \to S_N$, which together make S_N a cospan over the configuration of edge sets G_{iE}. Because of the universal property of S_E, there is a unique mapping $!_{\boldsymbol{so}} : S_E \to S_N$. Similarly, we proceed with the target mappings $\boldsymbol{ta}_i$ and obtain a mapping $!_{\boldsymbol{ta}} : S_E \to S_N$. In this way we come to a graph $S = (S_N, S_E, !_{\boldsymbol{so}}, !_{\boldsymbol{ta}})$, and it is just a routine check to show that (i) set mapping pairs $\iota'_i = (\iota'_{iN}, \iota'_{iE})$, $i = 1..m$ form graph mappings and hence the tuple $(S, \iota'_i, i = 1..m)$ is a cospan

over G_i, and (ii) this cospan possesses the universal property wrt other graphs, that is, is the colimit (merge) of the initial span of graphs in the universe of graphs and graph mappings.

A simple example of graph merge is presented in Fig. 5. Mathematically, this is just the merge of the configuration (A, R, X, f_1, f_2) of graphs and their mappings. Substantially, we suppose that the initial task was to merge graphs A, X modulo some correspondence between them; the latter to be found in the process of "schema matching" [18]. We assume that during this investigation, apart of establishing interconnections between schemas recorded in the mapping tables f_1, f_2, existence of an arrow s (missed in both schemas) was discovered.

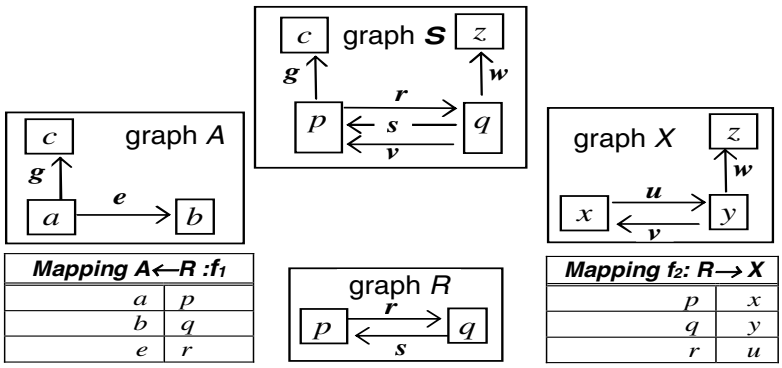

Fig. 5. Example of graph merge: $S = A \bigoplus_{R,f_1,f_2} X$

Clearly, if we deal with reflexive graphs and need to consider other "vertical" arrows in the metamodel, ***idle*** and ***exec***, we apply the same arguments but now to get mappings $!_{idle}, !_{exec} \colon S_N \to S_E$ we use the universal property of S_N rather than S_E. Moreover, it is not hard to see that the algorithm of graph merge presented above works in the general situations of higher-order graphs as well. Consider, for example, merge of n-graphs. Instead of two sets *Node* and *Edge*, we have a structure consisting of a family of sets $Edge_0$, $Edge_1 ... Edge_k$ interrelated by a number of "vertical" mappings $v_0 ... v_l$ (for example, for reflexive graphs we have $k = 1$ and $l = 3$). Given a generalized span of such higher-order graphs, we can merge them by, first, merging separately their component sets $Edge_0 ... Edge_k$ into cospans $S_0 ... S_k$ (like S_N and S_E for graphs). Then we use the universal property of the respective cospan for setting vertical mappings $!_0 ... !_l$ (like $!_{so} ... !_{exec}$) for reflexive graphs).

Remark: Graph + Constraints = Sketch. It was shown in category theory that the machinery described above works well even in more general situations but with an important restriction. To wit: the component sets and mappings are either unconstrained or, perhaps, subjected to a very special class of constraints expressible by equations (like commutativity conditions for graph

mappings) or implications built from equations [1]. As soon as we allow using more complex constraints, e.g., with existential quantifiers, we leave the world of higher-order graphs and come into the world of higher-order or generalized *sketches*. Roughly, the latter are graphs endowed with predicates defined on mappings targeted into these graphs. If a pair (G, π) with G a graph and π a predicate is a sketch, then only some of the graph mappings $h\colon G' \to G$ into G are valid, and then we write $h \models \pi$. The existence of constraints and, hence, non-valid graph mappings essentially complicates the merge procedure but does not disables it. Mathematical theory can be found in [15] and applications to OO visual modeling in [9], see also [6] for examples of sketch integration.

4 Scenario integration

In this section we consider an example of how the machinery developed above can work. The goal is to build a model (scenario) of BrokeredSale behavior from two copies of our model of Sale behavior. Suppose that after interviewing a few subject matter experts (SMEs), we know that a Brokered Sale (*bSale*) is a composition of two Sales, called the Wholesale (*wSale*) and the Retail Sale (*rSale*), such that the folowing conditions hold:

(o) Both sales deal with the same *item*.

(i) The *buyer* in *wSale* is the *seller* in *rSale*, and is called the *retailer* for the entire *bSale* behavior.

(ii) In general, *rSale* follows after *wSale* but

(iii) the *retailer* obeys the rule to pay to the whole-seller after she gets the payment from the retail *buyer*.

The question is whether it is possible to specify this information according to the patterns described in the previous section so that the procedures of graph merge would produce the intended result automatically. We will begin with the merge of collaboration bases and then will consider behavior.

4.1 Collaboration integration

Specifying interconnections for collaboration graphs in question is fairly simple. In the language of equations (sect. 3.2) we set *item*::*wSale* = *item*::*rSale* and *buyer*::*wSale* = *seller*::*rSale*. Suppose, in addition, that our interviews with SMEs revealed that

(iv) the retailer's role requires essential intellectual efforts and during *bSale* the retailer sometimes needs to perform a special procedure called *think*, and also needs to do complex *banking*.

In our language of collaboration graphs it means that there are two new types of messages, *think* and *banking*, from the node *retailer*:*Agent* to itself.

The information contained in the requirements (o,i,iv) is specified by the span shown in Fig. 6: the head is the Retailer collaboration graph and the arms are shown by dashed arrows.

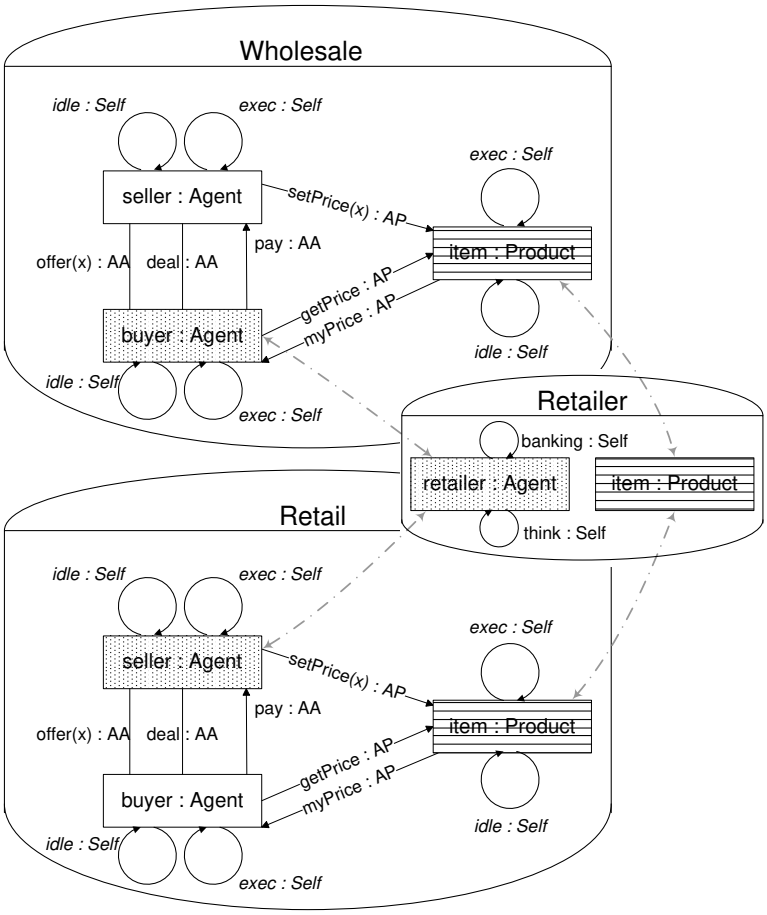

Fig. 6. Interconnection of the *wSale* and *rSale* collaborations.

The result of merge is shown in Fig. 7. Note that apart from gluing the two pairs of nodes, the Retailer diagram brought to the merge collaboration graph two new message types, which were not specified in the Sale collaboration.

4.2 Behavior integration

The two scenarios to be integrated, *wSale* and *rSale*, are shown in the left half of Fig. 8 (disregard bold dashed lines in them for a while). Positioning *rSale* under *wSale* does not have any formal semantic meaning, so far, they are entirely independent scenario graphs. The graph merge procedure discussed in sect. 3 suggests that to explicate the correspondence between the scenarios (i.e., to specify behavioral requirements (ii) and (iii)), we need to build a certain inter-connecting scenario and relate it to *wSale* and *rSale* graphs. Clearly, this correspondence graph should specify the retailer's lifeline in the entire brokered sale.

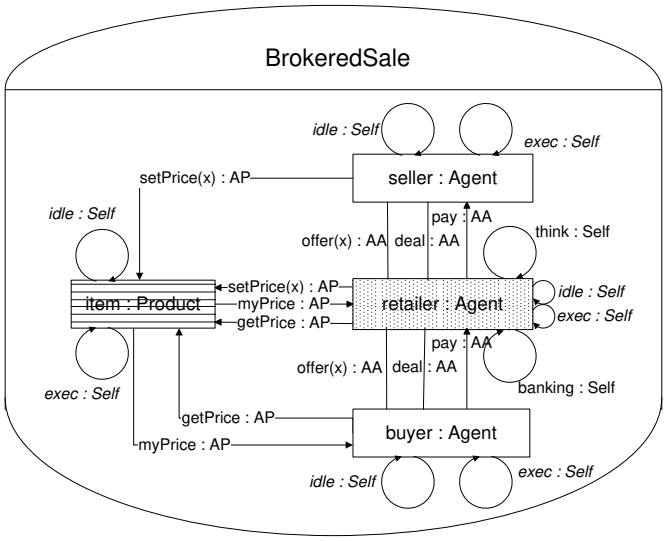

Fig. 7. Result of merging the collaboration diagrams in Fig. 6.

The retailer's lifeline is presented by the scenario graph *rLife* in the right half of Fig. 8, which is typed over the Connector collaboration graph. The first fragment R0-R1 encodes a sequence of ***idle*** and ***exec***-edges of retailer's life as a buyer in the *wSale* scenario from the very beginning at B0 till the moment B4 of sending the deal message. Informally, we can say that at the point R0 the control goes to *wSale* and at the point B4 leaves it for coming back to *rLife*, and the arrow R0-R1 is nothing but composition of arrows B0-B1...B3-B4 in the graph *wSale*. To capture this idea formally, we first augment the *wSale* graph with a new derived arrow B0-B4 (of type ***exec***) built by composition of the intermediate arrows: /B0-B4$\stackrel{\text{def}}{=}$ B0-B1▷...▷B3-B4, where ▷ denotes the operation of arrow composition; following the UML notational habits, we prefix the names of derived elements by slash. Then we can set an equivalence B0-B4::*wSale*$\sim$ R0-R1::*rLife*.[3]

The Retailer's lifeline from R1 to R3 is a new piece of information (not covered by either of sale scenarios) about the retailer's activity in between her roles as the *buyer*::*wSale* and the *seller*::*rSale*. After that, the control goes to the *rSale* scenario at point S0 and leaves it at point S4, coming back to *rLife* at R4. Here another new fragment (neither in *wSale* nor *rSale*) of the retailer's activity starts: processing the payment from the retail buyer and preparation to pay to the whole-seller. At point R6 the control returns to the *wSale* scenario.

[3] To be precise, we also define composition in the type graph and then factorize it by the *equivalence* ***exec***▷***idle*** = ***exec*** = ***idle***▷***exec***. Thus, any composition of ***idle*** and ***exec*** arrows results in ***exec***.

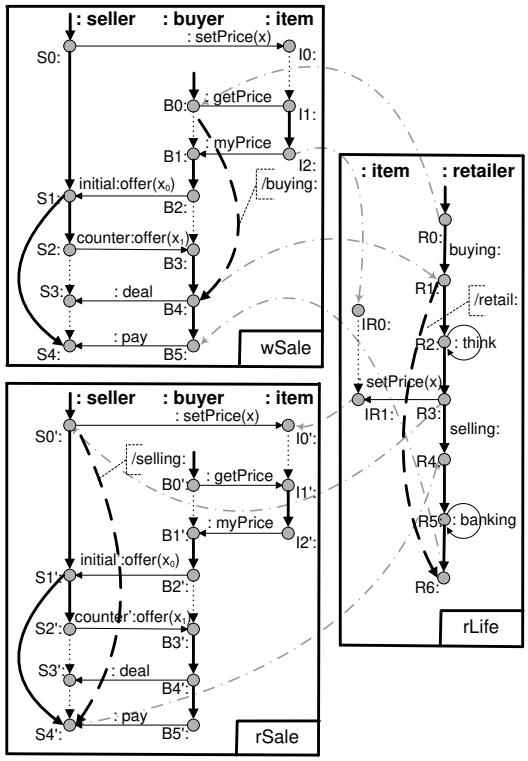

Fig. 8. Three scenarios to be merged

An important fact still not specified is that the B4-B5 arrow in *wSale* is, in fact, unfolded into a complex fragment R1-R6 of retailer's activity in *rLife*. The latter is a derived arrow /R1-R6 $\stackrel{\text{def}}{=}$ R1-R2⧐...⧐R5-R6, and we set an equivalence B4-B5 $\sim$ /R1-R6. Note that this equivalence is really important for getting the intended result of the formal merge procedure: without it, in the resulting graph we would have two paths from B4 to B5: one coming from graph *wSale* and the other from *rLife*. With this equivalence, we again have two paths but now the path (arrow) coming from *wSale* is stated to be derived (by composition of other arrows) and hence later can be removed from the merge graph (see below). Note that to declare the equivalence in question, we need to have the derived arrow /*retail* in the *rLife* scenario, which, in turn, requires the presence of the arrow R3-R4. The latter, in fact, presents the corresponding composed activity in *rSale*. Thus, we need to introduce a derived arrow /S0'-S4' $\stackrel{\text{def}}{=}$ S0'-S1'⧐S1'-S2'⧐S2'-S3'⧐S3'-S4' in graph *rSale*, and declare one more equivalence: R3-R4::*rLife*$\sim$ /S0'-S4'::*wSale*. Note that while arrow /B0-B4 was introduced to facilitate reading the *rLife* scenario graph, having derived arrows /R1-R6 and /S0-S4 and the respective equivalences is absolutely necessary for proper integration.

Note also that the dashed-dotted curly lines in Fig. 8, which show a flow of control between the view scenarios, formally set relations between *rLife* and *wSale*, and between *rLife* and *rSale* graphs. It precisely corresponds to the integration schema specified in Fig. 3(b) with $A = wSale$, $X = rSale$, $R = rLife$ and $R_f = \{$(R0-R1, B0-B4), (R1-R6, B4-B5)$\}$, $R_g = \{$(R3-R4, S0-S4)$\}$.

If we now run the merge procedure described in sect. 3 on this input configuration, it will return the graph shown in Fig. 9. Note three derived arrows among the elements of this graph. When two arrows, one is basic (in one graph) and the other is derived (in another graph) are glued together in the merge, the result is a derived arrow because it can be indeed derived exactly in the same way as it is derived in its component graph. For example, the arrow B4-B5 was basic in *wSale* scenario but becomes derived in the merge after gluing it with the derived arrow /R1-R6, because all the operands for its derivation are present in the merge graph. Keeping derived arrows in the merge graph may be useful for traceability, but apart from that they can be safely removed. We will call this last step of integration *normalization*. In our example normalization is fairly trivial but it can be more complicated, see [6, 8] for some details. A sequence diagram equivalent to the merge scenario graph without derived arrows is shown in Fig. 10.

The merge procedure in our example can be also seen in topological terms. It cuts-off the arrow B4-B5 from *wSale* and inserts into the "hole" the arrow R1-R6::*rLife* with its adjoint "environment", simultaneously gluing together the arrows R0-R1::*rLife* and B0-B4::*wSale* and the nodes IR0::*rLife* and I2::*wSale*. Then, it cuts-off the arrow R3-R4 in the graphs *rLife* (within the result of the previous operation) and inserts into the hole the arrow S0-S4::*rSale* simultaneously gluing together the arrows R3-IR1::*rLife* and S0-I0::*rSale*. Thus, we "break" the initial scenarios into pieces and then assemble from these pieces the integral behavior. This topological view of scenario merge operation can provide a useful guidance in complex situations.

4.3 General pattern

The example we have just considered suggest the following general format/pattern of scenario integration.

Formalization. We fix some universe $\mathcal{U}$ of higher-order graph-based structures like scenario graphs. We will call objects of this universe *$\mathcal{U}$-graphs* or just *graphs*. Scenarios to be integrated are presented as $\mathcal{U}$-graphs, $\mathbf{G} = \{G_1...G_m\}$, which we call *views*.

Specifying view correspondences. Correspondences between scenarios are specified by another family of graphs, $\mathbf{R} = \{R_1...R_n\}$. As we have seen, the latter may contain new information not captured by views. Mathematically, graphs R_j play the same role of input structures for the merge algorithm as view graphs G_i. Thus, we come to a family of graphs $\mathbf{H} = \{H_1...H_{m+n}\}$, $\mathbf{H} = \mathbf{G} \cup \mathbf{R}$, to be integrated modulo some correspondences (equivalences) between them.

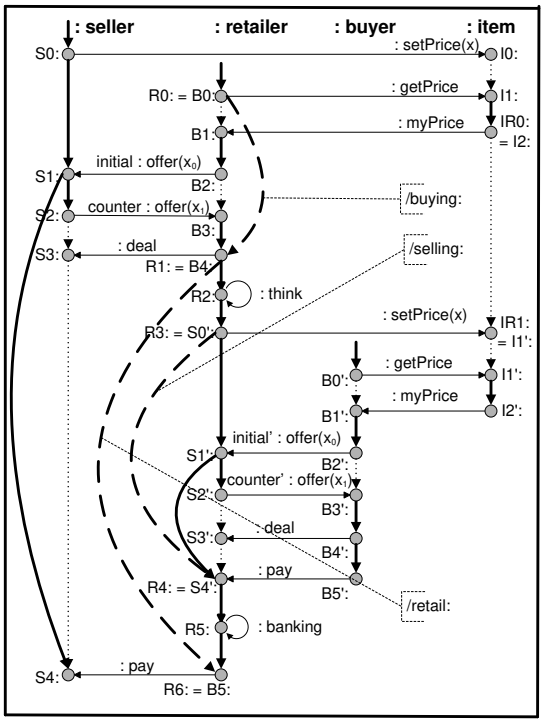

Fig. 9. Result of merging scenarios in Fig. 8. The event and message occurrences coming from *rSale* are "primed", and shown as "equal" to elements coming from *wSale* if they are glued together.

To set these correspondences, we may need to augment the view graphs with new elements derived by the operation of arrow composition. In this way we come to a family of augmented graphs $\overline{\mathbf{H}} = \{\overline{H}_1 ... \overline{H}_{m+n}\}$ together with a family of mappings (determined by correspondences) $\mathbf{h} = h_1 ... h_k$ between them.

Merge. The configuration (generalized span) $(\overline{\mathbf{H}}, \mathbf{h})$ is automatically merged according to the algorithm described in sect. 3. The procedure returns a cospan of graphs and mappings, $\mathbf{S} = (S, \overline{\iota'}_1 ... \overline{\iota'}_{m+n})$, $\overline{\iota'}_i : \overline{H}_i \to S$. Its head S may contain derived elements.

Normalization. In the merge graph S a subgraph S_0 should be chosen in such a way that any element in S can be derived from elements of S_0. Besides this technical requirement, the chosen subgraph should be compact and semantically meaningful, and should provide transparent meaning for derivations required to augment it up to S.

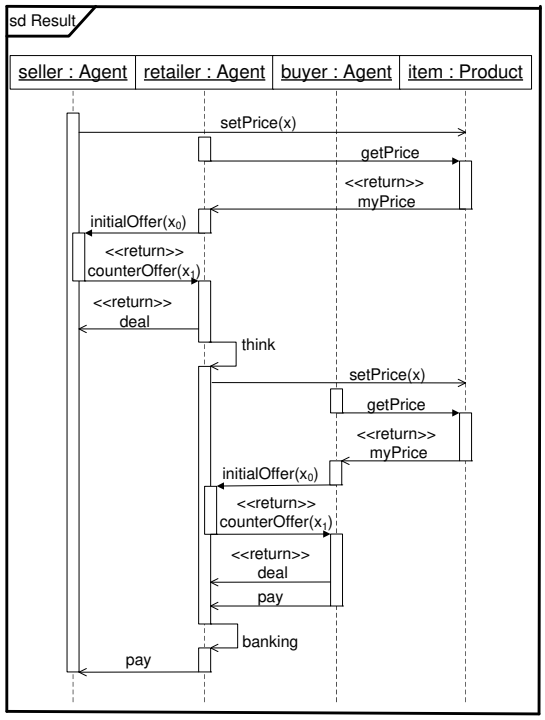

Fig. 10. Result of merging scenarios in Fig. 8 as a sequence diagram

5 General discussion and future work

In this section we briefly discuss a few issues skipped in the presentation above, particularly, possible extensions and limitations of the approach

5.1 Schema matching. An important point in making the pattern above really working is how to discover the correspondences between views. Actually it is a highly non-trivial and essentially heuristic issue known in the MMt literature under the name of *schema matching*, and it is entirely beyond the scope of our paper. We can only mention that data schema matching is an area of active research, where sophisticated algorithms based on special AI techniques (particularly machine learning) are studied and developed, see [18] for a survey and [11] for a discussion of promising novel ideas in the field. We believe that a similar activity is necessary for behavioral view matching as well

5.2 Beyond the merge operation. Other operations over higher-order graphs can be useful in scenario management. Consider, for example, extraction of the common part from two or several scenarios. Or, given a mapping between scenarios, we may need to find the image of this mapping in the target scenario, and then, perhaps, extract that part of the target scenario which is disjoint to this image (note that the notion of disjointness in graphs differs from that in sets).

It is shown in category theory that these and other counterparts of ordinary set-theoretical operations can be performed with higher-order graphs as well [4].

5.3 Adding more structure to behavior. An advantage of our framework of scenario graphs with their focus on typing is that it brings the benefits of strict type discipline to scenario-based behavior modeling. This type discipline can be elaborated even more. First of all, it is possible to introduce 2-edges into collaboration graphs in order to specify relations between message occurrences. For example, we could introduce (i) a 2-arrow *return* from the arrow *myPrice* to the arrow *getPrice* into the graph G_1 of message types (see Fig. 2) and, correspondingly, (ii) 2-arrows form message occurrences :*myPrice* to occurrences :*getPrice* in the occurrence graph G_2 to be labeled by the type *return*. Particularly, it would add more discipline to UML2 sequence diagrams by making the correspondences between *return* arrows and their "owning" procedures explicit. Another step towards bringing more structural discipline could be to consider subclassing in collaboration graphs underlying scenarios.

One more step is to add predicates to the collaboration graphs, which would constrain the possible scenario graphs over them. For example, so far we implicitly assumed that each execution in the scenario is triggered by exactly one trigger. To explicate this condition formally, we need to specify the corresponding predicate and attach it to each ***exec***-arrow in the collaboration graph G_1 (Fig. 2). We define that a graph mapping $\tau\colon G_2 \to G_1$ *satisfies* the predicate iff for any arrow e in G_2 such that $\tau(e) = \boldsymbol{exec}$, there exist a unique arrow m in G_2, called the *trigger* of e, such that $m.\boldsymbol{ta} = e.\boldsymbol{so}$.

In general, there may be multiple predicates/constraints embodied into the collaboration graph. A scenario graph $\tau_1\colon G_2 \to G_1$ is called *well-formed* if (G_2, τ_1) satisfies all constraints in G_1.

The main difficulty with introducing constraints into the formalism is that they essentially affect its algebraic properties. Graphs with constraints are *sketches* (see Remark 3.3 in sect.3.3 above), which are much more expressive but much harder to work with. We plan to explore the possibilities and problems of using generalized sketches in scenario based modeling in the nearest future.

5.4 Heterogeneous view integration. Scenarios specified in different languages can be mapped to higher-order graphs and then integrated. The language of graphs is sufficiently expressive to make this idea practically interesting. Moreover, if we allow using constraints in addition to graphs as explained above, then graphs becomes sketches, whose language is universally expressible. It was proven in category theory that any formal construction can be specified in the sketch language (see [9] for a discussion).

5.5 Merging incomplete and inconsistent scenarios. The technique proposed in [19] to manage incomplete and inconsistent views can be immediately applied in our graph-based framework as well. We see here a promising area of future research.

6 Relation to other work

Sequential, alternative, parallel and iterative compositions of behavior models are well known and well explored in different contexts. Their versions for scenarios are specified in the standards addressing scenario-based modeling: interaction overview diagrams (IOD) in UML2 [17] and high-level MSCs in ITU [13]. These diagrams are essentially graphs whose nodes represent scenarios and edges show the control flow between them. In this schema, behaviors specified by nodes are considered non-overlapping, and the system behavior is composed from component nodes as holistic units.

In the paper we address an essentially different issue of how to specify overlapping between scenarios, and then integrate (merge) them without duplication. Following the terminological tradition of semantic data modeling mentioned in Introduction, we call this problem view integration. In contrast to an extensive literature on the subject in the field of data modeling, only few papers considered view integration (more or less directly) in behavior modeling [14, 20, 21, 12, 22, 7] We can also mention the Use Case Maps – a graphical high-level scenario modeling technique [5], which partially addresses inter-scenario overlapping but does not provide an integration algorithm.

A common feature of a majority of works in view integration (in both data and behavior modeling) is non-genericness of the definitions and algorithms: they essentially depend on the particular modeling language they employ and a generic format for specifying view overlapping is not offered. Indeed, specifying operations with models and their relationships in a generic way is a non-trivial issue, which hardly can be designed from scratch. Fortunately enough, a machinery for building generic specifications was developed in category theory and is waiting for its applications in many areas of model management (see [10, 8] for a brief presentation of the framework and its applications).

Still there are a few papers employing the categorical framework for view integration in different contexts and for different notions of view: for data modeling in database design in [6] and for schema merge in [8], for early requirement engineering in [19], for MSCs as scenario models in [14] and for software merge in [16]. The most important distinction of our integration pattern is that we work with views augmented with derived elements because information considered basic in one view can be derived in another. This phenomenon is fundamental for the entire integration problem yet seems not recognized by the community. Another distinction of our integration pattern is that we consider the possibility of discovering a new information not captured by views during the investigation of their overlapping.

The presentation of the machinery in our paper follows closely to that in [19] but with some important differentiations. Particularly, we emphasize the value of the universal property of the merge (as the *least upper bound* of merged models in some precisely defined sense described in sect. 3), so that nothing is lost and nothing extra is acquired. We use the universal property to show that basically the same merge machinery works well for higher-order graph-based structures while [19] works with structures "not higher" than typed graphs.

In [14], the general categorical machinery is employed for merging particular formal models: scenarios specified by MSCs are encoded as partially-ordered multisets – a well-known and deserved language for specifying scenarios. However, this formalism is far less expressive than UML2 sequence diagrams, which in part motivated our search for another formal framework and led us to higher-order graphs. Also, as it was already mentioned in Introduction, string-based (rather than graph-based) formalization used in [14] results in a bulky definition of morphism and makes the entire integration procedure less transparent and less scalable beyond small examples. In addition, only injective morphisms are considered in [14], which might be a serious yet not relevant restriction. On the other hand, [14] considers also control structures in scenario modeling (high-level MSCs), which we did not touch in the paper leaving it for future work.

References

1. M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall International Series in Computer Science, 1995.
2. P. Bernstein, A. Halevy, and R. Pottinger. A vision for management of complex models. *SIGMOD Record*, 29(4):55–63, 2000.
3. P. Bernstein and R.Pottinger. Merging models based on given correspondences. In *Proc. Very large databases, VLDB'2003*, 2003.
4. R. Brown, I. Morris, J. Shrimpton, and C. Wensley. Graphs of morphisms of graphs. Uiversity of Wales, Bangor, Preprint 06.04, 18pp, 2006.
5. R.J.A. Buhr and R.S. Casselman. *Use Case Maps for Object-Oriented Systems*. Prentice Hall, 1996.
6. B. Cadish and Z. Diskin. Heterogenious view integration via sketches and equations. In *Foundations of Intelligent Systems, 9th Int.Symposium*, Springer LNAI #1079, pages 603–612, 1996.
7. J. Desharnais, M. Frappier, R. Khédri, and A. Mili. Integration of sequential scenarios. *IEEE Trans. Softw. Eng.*, 24(9):695–708, 1998.
8. Z. Diskin. Mathematics of generic specifications for model management. In Rivero, Doorn, and Ferraggine, editors, *Encyclopedia of Database Technologies and Applications*, pages 351–366. Idea Group, 2005.
9. Z. Diskin and B. Kadish. Variable set semantics for keyed generalized sketches: Formal semantics for object identity and abstract syntax for conceptual modeling. *Data & Knowledge Engineering*, 47:1–59, 2003.
10. Z. Diskin and B. Kadish. Generic model management. In Rivero, Doorn, and Ferraggine, editors, *Encyclopedia of Database Technologies and Applications*, pages 258–265. Idea Group, 2005.
11. A. Halevy and J. Madhavan. Corpus-based knowledge representation. In *Proc. Int. Joint Conf. on Artificial Intelligence, IJCAI'03*, 2003.
12. D. Harel and H. Kugler. Synthesizing State-Based Object Systems from LSC Specifications. In *5th International Conference on Implementation and Application of Automata(CIAA '00)*, pages 1–33, London, UK, 2001. Springer-Verlag.
13. ITU-TS. Recommendation Z.120: Message Sequence Chart (MSC), 2000.
14. J. Klein, B.Caillaud, and L. Hlout. Merging scenarios. In *9th Int.Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, ENTCS, pages 209–226, 2004.

15. M. Makkai. Generalized sketches as a framework for completeness theorems. *Journal of Pure and Applied Algebra*, 115:49–79, 179–212, 214–274, 1997.
16. N. Niu, S. M. Easterbrook, and M. Sabetzadeh. A category-theoretic approach to syntactic software merging. In *Int.Conference on Software Maintainance*, pages 197–206, 2005.
17. Object Management Group, `http://www.uml.org`. *Unified Modeling Language: Superstructure. version 2.0. Formal/05-07-04*, 2005.
18. E. Rahm and P. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.
19. M. Sabetzadeh and S. Easterbrook. An algebraic framework for merging incomplete and inconsistent views. In *13th Int.Conference on Requirement Engineering*, 2005.
20. S. Uchitel and M. Chechik. Merging partial behavioural models. In *12th ACM SIGSOFT Int.Symposium on FSE*, pages 43–52. ACM Press, 2004.
21. S. Uchitel and J. Kramer. A workbench for synthesising behaviour models from scenarios. In *23rd IEEE Int.Conference on Software Engineering*, May 2001.
22. J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *22nd International Conference on Software Engineering*, pages 314–323, New York, NY, USA, 2000. ACM Press.
23. G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, 1995.