A General Approach for Scenario Integration*

Hongzhi Liang[1], Zinovy Diskin[2,3], Juergen Dingel[1], and Ernesto Posse[1]

[1] School of Computing, Queen's University, Canada
{liang,dingel,eposse}@cs.queensu.ca
[2] Department of Computer Science, University of Toronto
[3] Department of Electrical & Computer Engineering, University of Waterloo
zdiskin@swen.uwaterloo.ca

Abstract. An approach to integrating UML Sequence Diagrams is presented. It rests on a well-established theory, is generalizable to a large class of requirements engineering models, and supports many different kinds of scenario integration operations. An implementation of the approach as an Eclipse extension is described. Lessons learned from the implementation and during first, preliminary experiments to study the practical aspects of the approach, are discussed.

1 Introduction

The need to integrate software artifacts seems inherent to modern software development. On the one hand, the development may be distributed over several teams to leverage different expertise, experience or capabilities. On the other hand, breaking a task into smaller, more manageable pieces often is an effective means to deal with the kind of complexity that comes from, e.g., large numbers of stakeholders, views, features, or platforms. In each case, the separately developed artifacts need to be assembled as efficiently as possible into a consistent whole in which the parts still function as described.

While support for integration is required for a large variety of artifacts to, e.g., support separation of development or concerns, it appears particularly necessary for models of requirements. This is because requirements models are especially prone to change and evolution. This phenomenon is widely accepted within the software engineering community and much work has been done to address it.

Many approaches aimed at mitigating the effect of changing requirements appear to rely on some kind of integration. For instance, in [11] use case slices need to be composed, gradually refined and kept synchronized. Consequently, support for the integration of a different, possibly more detailed and separately developed, model into other models is required. In some situations this integration may be adequately realized through a simple kind of replacement operation; however, to achieve the traceability needed for large-scale, distributed development, a less destructive form of integration may be necessary. Moreover, the

* This work was supported by NSERC, the Ontario Centres of Excellence, IBM CAS Ottawa, Bell Canada through the Bell University Labs and partially by the Ontario Research Fund.

K. Czarnecki et al. (Eds.): MoDELS 2008, LNCS 5301, pp. 204–218, 2008.

refined part of a model may have complex relationships with its context that need to be preserved by the refinement.

Another class of approaches is based on role modeling, that is, the identification of the parts of an object that address a particular concern such as performing a task or maintaining an invariant. Changing requirements can then often be dealt with either by adding a role to an object or by modifying a single role without affecting the others. Role-based software development methodologies include Reenskaug's Object Oriented Role Analysis Method (OOram) [21] and VanHilst's Role-Oriented Programming (ROP) [27]. Both methodologies feature a "synthesis" step in which the implementation of an object is obtained from its roles. Again, possibly separately developed models of requirements describing the roles need to be integrated to produce a description of the overall behavior of the object. Typically, roles are not disjoint, that is, they can "overlap" in complex ways. For instance, two different collaborations may require the same interaction with an object. Indeed, a look at [21,27] shows it is exactly these kinds of relationship between roles that complicate the synthesis step. The integration must properly deal with these relationships and, for instance, avoid the creation of duplicated parts in the integrated model in case of overlap among the integration participants.

Despite the apparent need for the integration of models of requirements, relatively little concrete support for this activity seems to exist. A lot of the existing work on the topic of model integration either assumes a large degree of disjointness between models (e.g., the work on composition operations for UML Sequence Diagrams and Interaction Overview Diagrams), or targets very specific notations with no clear potential for a more general application. For instance, all three methodologies mentioned above remain relatively silent on how exactly the integration of the requirements models is to be achieved. In [11], the composition of use case slices is performed on the code level using, e.g., AspectJ. UML's package merge is mentioned as a mechanism to compose slices on the model level although package merge is currently not defined on interactions [5]. Moreover, no indication is given of how synchronization between separately evolving models can be accomplished. OOram cautions the developer to take care that the result of the synthesis model is "consistent with the meanings of all its base models" [21, page 124]. However, no exact definition of the synthesis operation is provided. In [27], the transparent composition of roles on the code level is discussed in detail, while the model level is discussed much more informally. For another example, consider the "combine" operation offered in IBM Rational Software Architect (RSA) V7.0 which is intended to support the integration of different models, but appears limited to class and object diagrams [15].

This paper describes our work towards filling this gap and pursues the following questions: How would an integration operation look like that properly deals with overlap between models and is generalizable to different kinds of requirements models? How could this operation be applied to a particular kind of model such as UML Sequence Diagrams? How could it be supported through tools? Which additional work is necessary to realize the vision of a general model

integration operation? In particular, we present an approach to integrating a core subset of UML Sequence Diagrams which rests on a well-established theory, is generalizable to a large class of models, and supports many different kinds of scenario integration operations. Our approach uses category theory as a mathematical framework in which UML Sequence Diagrams are represented as *typed graphs* and integration is achieved through the explicit representation of the relationships between models via structure-preserving maps and a colimit construction. This construction results in an intuitive and versatile operation which not only provides traceability "for free" (the original models can easily be identified in the integration result), but also serves as an effective mechanism to structure the implementation and implement consistency checking. Moreover, the approach is generalizable to other diagrams used to represent requirements (e.g., Message Sequence Charts, Communication Diagrams). An Eclipse-based implementation of the approach is described. Lessons learned from the implementation and during first, preliminary experiments, e.g., to study different scenario integration patterns and the overall use of the approach, are discussed.

The remainder of this paper is structured as follows: The representation of UML Sequence Diagrams as typed graphs and the integration operation are described in Sections 2 and 3. The implementation is sketched in Sections 4. Related work is discussed in Section 5, while conclusions are given in Section 6.

2 Sequence Diagrams as Typed Graphs

In general, a scenario is a record of possible message exchanges between communicating objects. Figure 1(a) presents an example: a *Sale* scenario specified by a UML Sequence Diagram [18]. The *Sale*, and any other scenario has a structural base: a set of interacting objects and (implicitly) the types of messages they can exchange. Objects are explicitly specified in the boxes on the top of the lifelines. Messages are arrows from senders to receivers labeled with message names. In fact, some typing is implicit there. For example, it is reasonable to consider the two messages *initialOffer* and *counterOffer* as two different *occurrences* of the same message type *offer* between a *Seller* and a *Buyer*.

Sequence Diagrams have many advantages, but they are not directly amenable to formal manipulations. Since Sequence Diagrams (and an overwhelming majority of other modeling languages) are diagrammatic, a formalization based on graph-based structures seems to be advantageous. Fortunately, the graph-based formalisms possess other desirable properties: they (i) are amenable to effective algebraic manipulations, (ii) are expressive and provide a base for generic specifications, (iii) have solid theoretical support provided by *category theory* and by a large body of work in graph rewriting, (iv) have tool support. In this paper, we present an informal description of a formalization for the core subset of Sequence Diagrams. A more formal description can be found in [7].

In our formalization, a Sequence Diagram, as in Fig. 1(a), is represented as a typed graph, that is a chain of two type mappings between three graphs

$$G_0 \overset{\tau_1}{\leftarrow} G_1 \overset{\tau_2}{\leftarrow} G_2$$

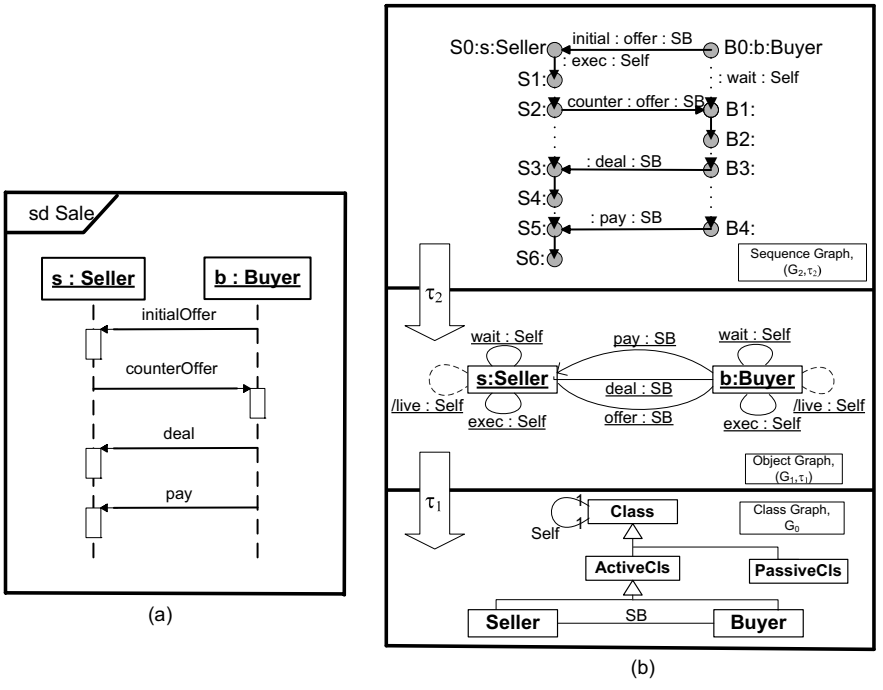

Fig. 1. A sample sequence diagram (a), and its typed graph (b)

as shown in Fig. 1(b). Graph G_0, or *class graph*, is shown in the bottom cell of Fig. 1(b) and is very similar to a UML Class Diagram, except that the edges represent *dynamic* rather than *static* associations [24]. We therefore interpret them as *message channels*, e.g., the *SB* message channel between *Seller* and *Buyer*. A root node called *Class* is contained in every class graph. A special message channel *Self* is represented as a self-association on *Class*. For concurrent systems, an object of a class may or may not own a thread of control. In other words, an object can be specified as either an *active object* having its own thread of control or a *passive object* without the control [22]. To distinguish these two different types of object, we introduce two sub-classes of the root *Class*, *ActiveCls* and *PassiveCls*. Inheritance is also used in our formalization to support sub-classing and association inheritance. For example, by introducing *ActiveCls* as a sub-class of *Class*, the message channel *Self* is inherited by *ActiveCls*.

Graph G_1, or *object graph*, whose nodes are objects participating in interactions and whose edges are *dynamic* links (instances of dynamic associations) or *message types* between objects, is shown in the middle cell of Fig. 1(b)[1]. Note that nodes and edges of this graph are typed (or labeled) by classes and

[1] 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.

message channels of Graph G_0 by mapping τ_1. We assume that an object of *ActiveCls* can be in one of two states: "executing" or "blocked and waiting for a response". To capture this, we introduce two special message types ***exec*** and ***wait*** as instances of self-association *Self*. Similarly, ***exec*** and ***rest*** (not used in Fig. 1) are introduced to capture the states of "executing" and "ready and really doing nothing" of an object of *PassiveCls*. A *derived* message type, */live*, is also shown in Fig. 1(b). A derived element is one that can be computed from other elements as a result of some algebraic operation. To be precise, we define $/live = \{\textbf{\textit{exec}}|\textbf{\textit{wait}}\}^*$, i.e., any sequential composition of ***exec*** and ***wait*** arrows results in */live*. In our formalization, although derived elements contain no new semantic information, they either facilitate the understanding of scenarios, or are even required for proper scenario integration. We borrow from UML and prefix the names of derived elements by "/".

Graph G_2, called *sequence graph*, is a partial order of *events* and *messages* typed over Graph G_1 by mapping τ_2, where nodes are event occurrences labeled by objects (to which these events happened) and arrows are message occurrences labeled by message types. The top cell of Fig. 1(b) shows the corresponding sequence graph of scenario in Fig. 1(a). Labels of all event occurrences besides S0 and B0 are omitted. If an arrow in G_2 is labeled by ***exec***, ***wait*** or ***rest***, then it will be attached to the lifeline of a single object (the one to which the self-association in G_0 is attached). For instance, all vertical arrows in our example sequence graph are labeled with either ***exec*** or ***wait***. Intuitively, labeling an arrow by ***exec*** or ***wait*** (***rest***) means, respectively, that in the time period between the source and target event occurrences the object is executing some procedure or doing nothing (is either blocked or ready). In the case of ***exec***, this is the procedure triggered by the message coming into the source event. To help us distinguish between arrows typed with ***wait*** (***rest***) or ***exec***, we use the following concrete syntax for them: the former are shown with dotted (e.g., between B0 and B1) and the latter with bold (e.g., between S0 and S1) arrows.

To summarize, we formalize Sequence Diagrams as three layers of directed, labeled graphs containing dynamic and static information, where lower layers serve as types for the higher layers. Such layered, typed structures are useful to support, e.g., consistency checking. Moreover, unlike formalizations based on, e.g., partially ordered multisets, our formalization retains the graphical nature of Sequence Diagrams. More precisely, there is an obvious similarity between the formalization and what is being formalized, which increases learnability.

3 Scenario Integration

We introduce scenario integration via a running example reminiscent of the role composition process required by OOram or ROP. Suppose we want to build a model (scenario) which integrates two copies of the Sale scenario into a brokered sale model: a BrokeredSale (*BS*) is a composition of two Sales, called the WholeSale (*WS*) and the RetailSale (*RS*). We want this integrated scenario to satisfy the following requirements:

(i) The *Retailer* is the *Buyer* in WholeSale, and it is the *Seller* in RetailSale.
(ii) The *Retailer*'s role requires two activities in addition to those of the sale transactions: a *Retailer* must do some *thinking* and some *banking*.

In order to integrate the two given scenarios we need to merge the corresponding class, object and sequence graphs. These three operations are in fact instances of the same generic operation of merging graphs, which we now sketch[2].

Given two typed graphs to be merged, we need to specify the overlapping elements (i.e., nodes and edges). That is, we need to establish a correspondence between these graphs of those elements which are supposed to be identified. To do this, a first approach, found in tools such as RSA [15], is to use heuristics like identifying elements by name. Such approach however, is not general enough. In particular it does not deal with requirements such as (i). A more general approach (arguably the most general approach) is to create a third graph, which we call the *head*, representing those common elements, and possibly containing new elements. We then specify how this third graph establishes the correspondence by defining a pair of maps which map elements of the head to elements of each of the original graphs. We call such maps *arms*. The head graph and the associated arms are called the *span* of the two original graphs. From the span, the merged graph will be generated.

Going back to the problem, we first look at how object graphs are merged, and then we consider sequence graphs. The merge of class graphs is obtained in an analogous manner.

Integration of object graphs. Fig. 2 shows the object graphs to be merged (WS_{OG} for the WholeSale and RS_{OG} for the RetailSale) together with the span (the head being the Retailer graph R_{OG} at the bottom) and the merged graph on top, satisfying the requirements (i) and (ii). The dashed arrows represent the correspondence mappings.

The head graph contains a new object *r:Retailer* with two self loops *thinking* and *banking*. There is a map from *r:Retailer* in R_{OG} to *b:Buyer* in WS_{OG}, representing the requirement that the retailer plays the role of buyer in WholeSale, and a map from *r:Retailer* in R_{OG} to *s:Seller* in RS_{OG}, representing the requirement that the retailer plays the role of seller in RetailSale. Hence, requirements (i) and (ii) are captured.

The merged object graph is BS_{OG}. Fig. 2 also shows how the elements of WS_{OG} and RS_{OG} are mapped to the elements of the resulting graph BS_{OG}. We now sketch how this graph is obtained. For full details, please see [7].

First, we form the disjoint union of the two original graphs and the head graph (WS_{OG}, RS_{OG} and R_{OG}.) Let us call this A. The mappings from the head graph (the arms) determine an equivalence relation on A where overlapping elements are identified. Then we partition the set of nodes of A according to this equivalence. That is, we group all equivalent nodes into sets. The collection of all these sets is the partition. In our example, we obtain three sets: $A_1 =$

[2] A detailed technical description of this operation can be found in many sources, e.g., [23] and our own technical report [7].

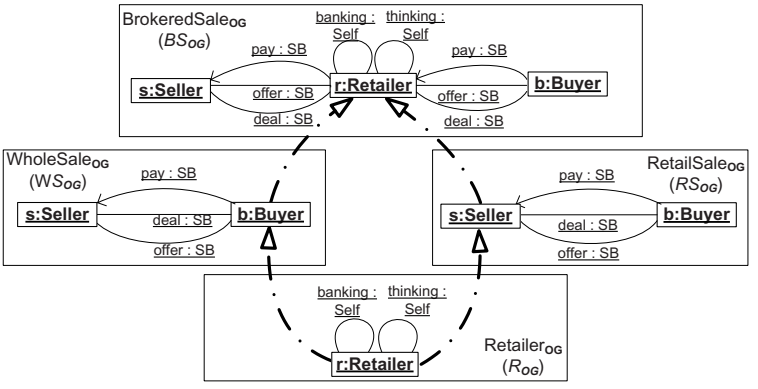

Fig. 2. The span of the WS_{OG} and RS_{OG} object graphs and the merged graph on top

$\{r{:}Retailer{::}R_{OG},\ b{:}Buyer{::}WS_{OG},\ s{:}Seller{::}RS_{OG}\}$, $A_2 = \{s{:}Seller{::}WS_{OG}\}$ and $A_3 = \{b{:}Buyer{::}RS_{OG}\}$, where we write $x : Class{::}Graph$ for the object x of type $Class$ in graph $Graph$.

The merged graph consists of a node for each of these sets. Hence we have a node for the *Retailer*, one for the *Seller* from WS_{OG} and one for the final *Buyer* from RS_{OG}. Edges of the merged graph are obtained in a similar fashion. The merged graph will also contain all edges from the WS_{OG} and RS_{OG} graphs, as well as new edges from the head graph, such as *thinking* and *banking*, which are not present in the two original graphs. These edges are present in the corresponding node of the merged graph. With the merged graph, we also obtain mappings from WS_{OG} and RS_{OG} into BS_{OG}, describing how the former are embedded in the latter and providing useful traceability information.[3]

Integration of sequence graphs. Now suppose that we have some additional requirements for our BrokeredSale scenario:

(iii) The RetailSale follows the WholeSale.
(iv) After buying from the whole-seller, the *Retailer* does some thinking. After this, he/she begins the process of retail selling, which is followed by some banking.
(v) The *Retailer* pays the whole-seller after he/she receives payment from the retail *Buyer*.

These are behavioral requirements and so they are to be captured by the result of merging the sequence graphs for the WholeSale and RetailSale.

As with object graphs, to merge sequence graphs we need to specify the points of overlap, and we do this by providing a span, i.e. a head graph and the associated arms defining the correspondence.

[3] In the terminology of Category Theory, the merged graph and the associated mappings form a *pushout* of the span.

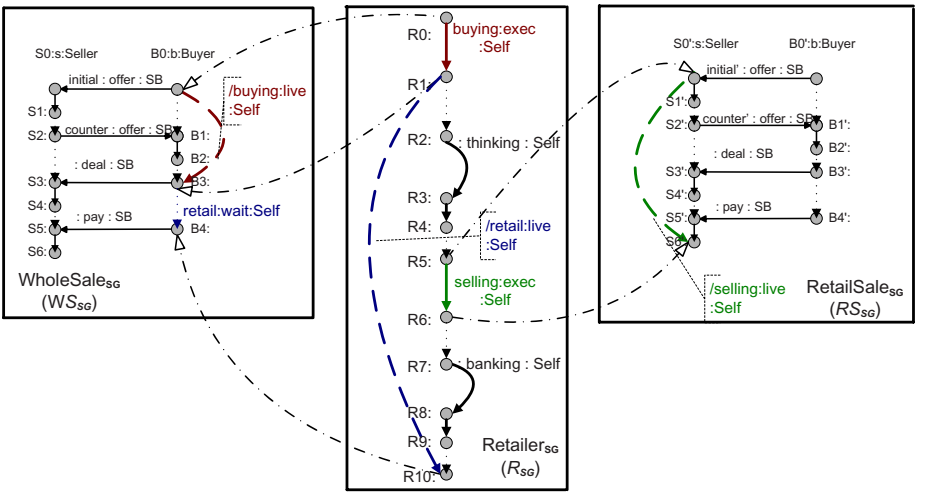

Fig. 3. The span of the sequence graphs WS_{SG} and RS_{SG} to be merged

Fig. 3 shows the sequence graphs to be merged with their span. The graph WS_{SG} is the sequence graph for the WholeSale, RS_{SG} is the sequence graph for the RetailSale, and R_{SG} is the sequence graph showing the lifeline of the *Retailer*, and thus it is the head of the span. Each graph is typed over its corresponding object graph.

By explicitly defining the correspondence between the two sequence graphs as a span in terms of the *Retailer*'s lifeline, we capture requirements (iii) - (v). In this graph, we are able to define new elements which where not present in either of the sale scenarios, such as the *thinking* and *banking* arrows. We also introduce arrows *buying* and *selling* which, while not present in the sale scenarios as individual arrows, correspond to a *composition* of arrows in their respective graphs. For example, the arrow *buying* in R_{SG} from R0 to R1 is mapped to the new derived arrow $/buying$ in WS_{SG}, which is the composition of the arrows from B0 to B3. Hence, the arrow *buying* can be seen, from the point of view of the retailer, as an abstraction of a sequence of actions that occur in the whole sale. Furthermore, we map a composition of arrows ($/retail$) in R_{SG} to a single arrow (*retail*) in WS_{SG}. Note that this composite arrow $/retail$ includes the *selling* arrow (which itself is associated to the composite $/selling$ in RS_{SG}.) This allows us to ensure that the RetailSale occurs within the retail process and before the *Retailer* pays the whole-seller, thus satisfying requirements (iii) - (v).

As with the object graph, we build the merged sequence graph by computing the equivalence relation induced by the span. This yields the sequence graph shown in Fig. 4(a). Note the derived (dashed-bold) arrows among the elements of this graph. When two arrows, where 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 derived exactly in the same way as it is derived in its component graph. For example, the arrow *retail* was basic in

WS_{SG} but becomes derived in the merge after gluing it with the derived arrow */retail*, because all the operands for its derivation are present in the merged graph. Derived arrows in the merged graph are useful for traceability, but apart from that they can be safely removed. We call this last step of the integration *normalization*. In our example normalization is fairly trivial but it can be more complicated without posing any fundamental problems[4]. A sequence diagram equivalent to the merged graph without derived arrows is shown in Fig. 4(b).

General process. The example we have just considered suggests the following general process for scenario integration:

1. *Formalization*: We define a universe of typed graphs and specify the scenarios to be integrated (the *views*) as typed graphs in this universe.
2. *Specification of view correspondences*: We define the correspondence between views by providing *spans*, which consist of 1) *head graphs*, i.e., typed graphs which contain elements (nodes and edges) that represent the overlap of the views, and 2) *arms*, i.e., mappings specifying the roles that elements of a head graph play in each view. The head graphs can contain new information which was not present in the original views.
3. *Merge*: For each graph of the typed graphs (class, object, sequence) the merged graph is obtained by computing the equivalence induced by the span, and the corresponding partition of the (disjoint) union of the views.
4. *Normalization*: In each merged graph, we eliminate redundant arrows, i.e. arrows which can be derived from basic (non-composite) edges.

A fundamental property of the merge operation is that the resulting graph contains exactly all of the information from the original graphs and the head graph: nothing is lost because views are mapped into the merge, and nothing extra is acquired owing to the universal property of pushout. It is also worth noting that while the example described the integration of just two scenarios, this pattern is applicable to any number of scenarios to be merged[5].

Integration and refinement. Model refinement can be understood in terms of integration: given a model A, a refinement A' of A can be seen as the result of merging A with a model B specifying details of A. For example, the BrokeredSale scenario can be seen as a refinement of the WholeSale scenario, where the retail activity has been detailed by merging the RetailSale scenario via R_{SG}.

With such a view of refinement as integration, the relationships between the refined part of a model and its context are preserved by the operation of refinement. Nevertheless, this notion of refinement contrasts with the notion of refinement as proposed in, e.g., the STAIRS framework [9]. In STAIRS, the meaning of a sequence diagram is defined by the set of traces that any implementation must satisfy, and refinement is defined in terms of containment of sets of traces. We present an orthogonal view of refinement, where an action

[4] See [3,6] for more details.
[5] This is achieved by computing *general colimits*, of which pushouts are a particular example.

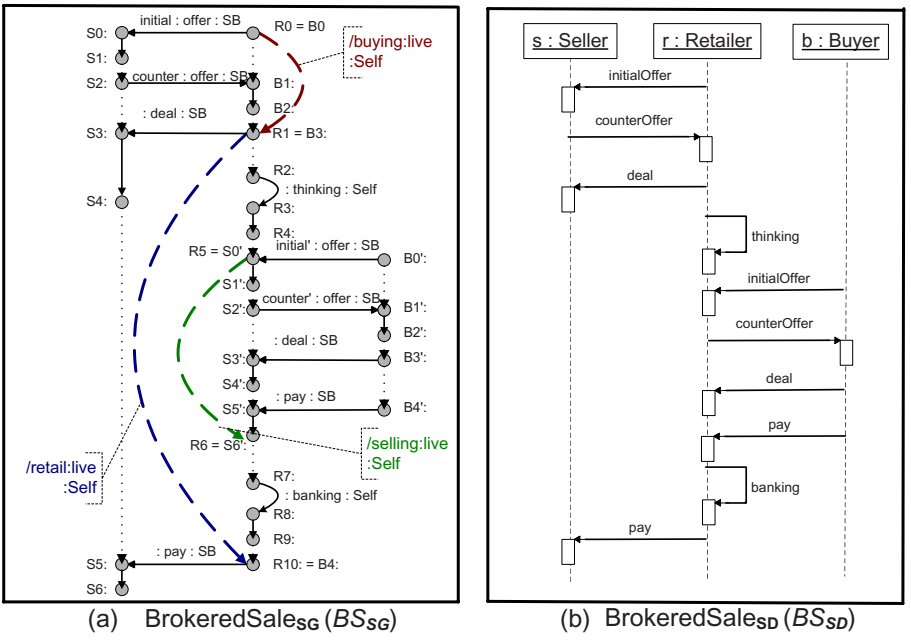

Fig. 4. Result of merge as a typed graph (a) and as Sequence Diagram (b)

or sequence of actions (and therefore the corresponding trace) is refined by a sequence of actions (resp. a trace) by expansion, i.e., an action (an arrow in the sequence graph) can by refined by a composition of actions (arrows.)

Our preliminary evaluations indicate that this view of refinement can be useful for model management in general and for the formalization and support for development methodologies that require the manipulation of requirements models (e.g., the methodology in [11] and OOram) in particular.

4 Implementation

A prototype implementation has been built as an Eclipse extension. This allows us to take full advantage of other existing Eclipse-based modeling tools (e.g., IBM RSA) which implement editing, visualization, import and export of Sequence Diagrams. We also rely on the EMF, UML2 and GMF frameworks.

The architecture of the prototype follows our general process described in Section 3. The main components of the prototype and the data flow among them are depicted in Fig. 5. Our prototype assumes that the Sequence Diagrams and the head of the merge are created by the user with the help of existing modeling tools, such as RSA. After importing the Sequence Diagrams, the *SD2TG Transformer* will automatically convert them into typed graphs. A variety of different existing transformation tools could have been used to develop this component. We have chosen TXL [25]. An Ecore-to-TXL grammar generator has been implemented to provide general support for writing TXL-based model-to-model

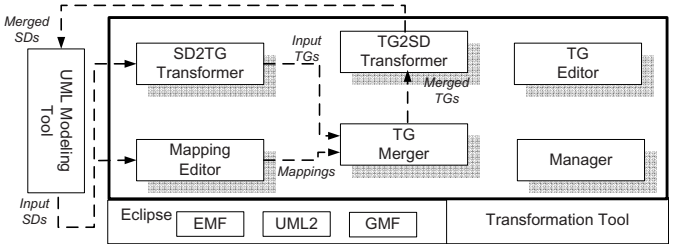

Fig. 5. The architecture of the prototype SD Integration Tool

transformations. The users interactively create mappings or correspondences between Sequence Diagrams with the help of the *Mapping Editor*. The correspondences between Sequence Diagrams are then translated to correspondences between typed graphs. Strictly speaking, a mapping editor operating on typed graphs directly rather than Sequence Diagrams would be more efficient, thus obviating the need for transforming mappings. Instead, the current implementation hides the typed graph representation from the users. As a consequence, it is likely more user-friendly as the users can work with familiar concepts, i.e., Sequence Diagrams. We are also working on an expert mode which allows users to work on typed graphs directly. In this expert mode, the similarity between a Sequence Diagram and its typed graph will be very helpful. Next, the *TG Merger* takes typed graphs and mappings between them, and produces a merged typed graph. Finally, before the merged result can be visualized by a UML modeling tool, the *TG2SD Transformer* transforms merged typed graphs back to Sequence Diagrams. Again, the transformation has been implemented in TXL.

An interactive graphical editor, *TG Editor*, based on the Eclipse Graphical Modeling Framework (GMF), is also provided. It allows for the creation, modification, and display of typed graphs. Moreover, it allows for the merge results to be validated through the examination of the typed graphs. To facilitate the overall Sequence Diagram integration process and to manage the data flows between components, the *Manager* has been implemented as an Eclipse wizard.

4.1 Evaluation and Observations

To evaluate the correctness of the implementation, we have run the prototype on a set of tests. The tests include, for instance, variations of the brokered sale. The merged results produced by the prototype were then visualized in RSA. Finally, the visualized results were manually inspected and compared with the expected results. These preliminary evaluation results show that the prototype was correctly implemented according to our general merge process. A more comprehensive evaluation involving different case studies and applications to some of the methodologies mentioned in the introduction are currently in progress.

We have made the following observations during the implementation and the evaluation of our prototype:

• An important step during integration is the discovery and specification of the correspondences between views. We agree with [2] that an explicit representation of these correspondences is unavoidable for a general and manageable notion of model integration. More work is necessary to alleviate the burden of discovering and specifying these relationships. A promising idea may be the use of high-level merge patterns. The identification of these patterns is subject of future work, but the brokered sale example, for instance, suggests that combinations and variations of, e.g., lifeline composition and refinement will be relevant here. By turning the identified patterns into user fillable templates, correspondences between views could be described in more high-level terms while the details are automatically generated. A complementary way to help identifying correspondences could be an activity similar to data schema matching [20].

• The UML Sequence Diagram syntax is slightly insufficient for the specification of the view correspondences. More precisely, it is impossible to select a portion of a lifeline between two occurrences that is not covered by an execution specification (in our formalization, the portion of the lifeline is a ***wait*** or ***rest*** message). A UML profile for typed graphs has been implemented as part of the prototype to remedy this situation. It can be optionally applied to Sequence Diagrams and allows a portion of a lifeline to be stereotyped as either a ***wait***, ***rest*** or derived message.

• The mathematical theory underlying our approach helped us structure our implementation, i.e., the prototype is clearly divided into several cohesive components and Java packages, each implementing a separate portion of the theory. Moreover, the prototype inherits some powerful properties from the theory. For instance, traceability between input and merged elements is easily achieved by our tool by following the span and the computed pushout. Finally, the typing information provided by typed graphs greatly facilitates consistency checking.

• Our approach to integration is quite general. Although we showed a two-way merge example, our merge process and prototype tool actually support multi-way merge with any number of scenarios and heads. Moreover, the merge process is not limited to Sequence Diagrams (or MSCs), but can be easily extended to other kinds of models, e.g., Communication Diagrams or Class Diagrams. In principle, every type of diagram representable as a directed, typed graph is amenable to our integration approach.

5 Related Work

The composition of behavior models has already been explored in different contexts. For instance, UML2 offers Interaction Overview Diagrams (IOD) [18], while MSCs have been generalized to high-level MSCs [10]. These diagrams are essentially graphs whose nodes represent scenarios and edges show the control flow between them. Thus, behaviors specified by nodes are considered non-overlapping. In contrast, we address the issue of how to specify overlap between

scenarios, and then integrate them without duplication. A similar problem has been studied in semantic data modeling for a long time, where it is called *schema* or *view integration* (see the recent survey [1] and a widely cited paper [19]). Most work in view integration is non-generic in the sense that definitions and algorithms depend on the use of a particular modeling language. However, a general and data-model independent approach to the problem can be designed with category theory means [6]. The categorical framework has been used for schema merging in the contexts of database design [3] and metadata management [6], for early requirement engineering [23], and for software merge [17].

While view integration is well studied in the context of data modeling, only a few papers attempt to tackle the issue in behavior modeling [26,4], including a categorical approach to merging MSCs in [12]. The most important distinction of our approach is that we work with views augmented with derived elements because information considered basic in one view may be derived in another. This phenomenon is crucial to the integration problem and gives rise to an effective notion of refinement. Another distinction is that we allow for the specification of new information not captured by views during the identification of their overlap. In [12], categorical machinery is employed for merging partially-ordered multisets representing MSCs. Thus, the formalization in [12] is "string-based", rather than graph-based like ours. We argue that this change in representation makes the integration procedure less transparent and more difficult to learn. Finally, only injective morphisms are considered in [12], which might be a serious yet not relevant restriction. On the other hand, [12] considers also control structures in scenario modeling (high-level MSCs), which we leave for future work.

Recently, graph grammars have been proposed for the integration of aspects into different kinds of models (e.g., class diagrams, state machines, and sequence diagrams) [28,29]. A pattern language is used for the specification of the overlap between models and aspects. Compared to our work, the use of a pattern language to specify the relationships between models makes the approach more user-friendly. However, it also makes it less general and versatile because possibly quite different pattern languages need to be defined for different diagram types; since the pattern language needs to strike a delicate balance between expressiveness and usability, this step may not be straight-forward. Finally, our approach does not exclude the use of patterns and, moreover, may benefit from them. Yet we prefer to start with a general expression of model relationships and leave the identification of suitable patterns for a later stage of development.

Some work on the synthesis of state-based behavior models from scenarios involve the integration of scenarios as a by-product (see [16] for a survey). These approaches allow for complex forms of integration, and the labeling mechanism in [14] is clearly akin to our idea of explicit mappings ("arms"). However, our approach also allows us to include new information in the correspondence "head", which is not contained in any of the integrated models. As our brokered sale example illustrates, this new information can provide a new useful context into which the merged scenarios are placed. In addition, our approach inherits some useful properties from the underlying colimit operation (traceability and

universality), which the majority of scenarios-from-state-machines procedures do not seem to offer.

Tools supporting tasks related to model management, particularly, model merging already exist, e.g., AMW [8] and Epsilon [13]. Conceivably, they could have been used to aid in the construction of our prototype. However, we chose to implement it without the use of existing merge tools because we wanted to study our approach in its purest form, unencumbered by the issues that can arise when encoding one theory in another. Moreover, we wanted to see to what extent the theory and its properties would be leveraged on the implementation level. We acknowledge, however, that the manual specification of mappings supported by AMW is similar to ours and could have been used for our mapping editor. In future work, it would be interesting to investigate the combination of AMW's automatic mapping generator or Epsilon's rule-based matching with our categorical approach.

6 Conclusions

We have presented an approach to integrating scenario-based models, particularly UML Sequence Diagrams, based on the colimit construction known from category theory. Our formalization of Sequence Diagrams as typed graphs retains the graphical nature of Sequence Diagrams, yet is amenable to algebraic manipulations and consistency checking. The approach provides traceability and guarantees that nothing is lost and nothing extra is acquired in the merge. Moreover, the algorithm is generalizable for other kinds of models and any number of models to be merged.

A prototype tool following the approach has been implemented. Initial evaluation has shown that the ideas presented in this paper represent promising steps towards more rigorous management of requirement models. However, the support for discovery and specification of model relationships needs to be improved.

References

1. Aleksandraviciene, A., Butleris, R.: A comparative review of approaches for database schema integration. Advances in Information Systems Development (2007)
2. Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., Sabetzadeh, M.: A manifesto for model merging. In: 1st International Workshop on Global Integrated Model Management (GaMMa 2006), Shanghai, China (May 2006)
3. Cadish, B., Diskin, Z.: Heterogenious view integration via sketches and equations. In: Michalewicz, M., Raś, Z.W. (eds.) ISMIS 1996. LNCS, vol. 1079, Springer, Heidelberg (1996)
4. Desharnais, J., Frappier, M., Khédri, R., Mili, A.: Integration of sequential scenarios. IEEE Trans. Softw. Eng. 24(9), 695–708 (1998)
5. Dingel, J., Diskin, Z., Zito, A.: Understanding and improving UML package merge. Software and Systems Modeling (2008), doi:10.1007/s10270-007-0073-9
6. Diskin, Z.: Mathematics of generic specifications for model management. In: Encyclopedia of Database Technologies and Applications, Idea Group (2005)

7. Diskin, Z., Dingel, J., Liang, H.: Scenario integration via higher-order graphs. Technical Report 2006-517, Queen's University (2006), http://www.cs.queensu.ca/TechReports/Reports/2006-517.pdf
8. Del Fabro, M.D., Valduriez, P.: Semi-automatic model integration using matching transformations and weaving models. In: Sympos. on Applied Computing (2007)
9. Haugen, Ø., Husa, K., Runde, R., Stølen, K.: STAIRS: towards formal design with sequence diagrams. Software & Systems Modeling 4(4), 355–367 (2005)
10. ITU-TS. Recommendation Z.120: Message Sequence Chart (MSC) (2000)
11. Jacobson, I., Ng, P.: Aspect-Oriented Software Development with Use Cases. Addison-Wesley Professional, Reading (2004)
12. Klein, J., Caillaud, B., Hélouët, L.: Merging scenarios. In: 9th Int.Workshop on Formal Methods for Industrial Critical Systems. ENTCS, pp. 209–226 (2004)
13. Kolovos, D., Paige, R., Polack, F.: Merging Models with the Epsilon Merging Language (EML). In: Int. Conf. on Model Driven Engineering, Languages and Systems (MoDELS 2006) (2006)
14. Krüger, I., Grosu, R., Scholz, P., Broy, M.: From MSCs to statecharts. In: Int. Workshop on Distributed and parallel embedded systems, Norwell, MA, USA (1999)
15. Letkeman, K.: Ad-hoc modeling - Fusing two models with diagrams, http://www.ibm.com/developerworks/rational/library/07/0410_letkeman
16. Liang, H., Dingel, J., Diskin, Z.: A comparative survey of scenario-based to state-based model synthesis approaches. In: 5th International Workshop on Scenarios and State Machines, SCESM 2006 (2006)
17. Niu, N., Easterbrook, S.M., Sabetzadeh, M.: A category-theoretic approach to syntactic software merging. In: Int. Conf. on Software Maintainance (2005)
18. Object Management Group. Unified Modeling Language: Superstructure. version 2.1.2 Formal/2007-11-04 (2007)
19. Pottinger, R., Bernstein, P.: Merging models based on given correspondences. In: Proc. Very large databases, VLDB 2003 (2003)
20. Rahm, E., Bernstein, P.: A survey of approaches to automatic schema matching. VLDB Journal 10(4), 334–350 (2001)
21. Reenskaug, T.: Working With Objects: The OOram Software Engineering Method. Manning (1995)
22. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley, Reading (2004)
23. Sabetzadeh, M., Easterbrook, S.: An algebraic framework for merging incomplete and inconsistent views. In: 13th Int.Conference on Requirement Engineering (2005)
24. Stevens, P.: On the Interpretation of Binary Associations in the Unified Modeling Language. Software and Systems Modeling 1(1) (2002)
25. TXL. About TXL (2007), http://www.txl.ca/nabouttxl.html
26. Uchitel, S., Chechik, M.: Merging partial behavioural models. In: 12th ACM SIGSOFT Int.Symposium on FSE, pp. 43–52. ACM Press, New York (2004)
27. VanHilst, M.: Role-Oriented Programming for Software Evolution. Ph.D. dissertation, Univ. of Washington, Dept. of Computer Science and Engineering (1997)
28. Whittle, J., Jayaramana, P.: MATA: A Tool for Aspect-Oriented Modeling based on Graph Transformation. In: Aspect-Oriented Modeling Workshop (2007)
29. Whittle, J., Moreira, A., Araújo, J., Rabbi, R., Jayaraman, P., Elkhodary, A.: An Expressive Aspect Composition Language for UML State Diagrams. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735. Springer, Heidelberg (2007)