World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:9, No:9, 2015

Software Evolution Based Sequence Diagrams Merging

Zine-Eddine Bouras, Abdelouaheb Talai

***Abstract*—**The need to merge software artifacts seems inherent to modern software development. Distribution of development over several teams and breaking tasks into smaller, more manageable pieces are an effective means to deal with the kind of complexity. 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. In addition, earlier changes are introduced into the life cycle and easier is their management by designers. Interaction-based specifications such as UML sequence diagrams have been found effective in this regard. As a result, sequence diagrams can be used not only for capturing system behaviors but also for merging changes in order to create a new version. The objective of this paper is to suggest a new approach to deal with the problem of software merging at the level of sequence diagrams by using the concept of dependence analysis that captures, formally, all mapping, and differences between elements of sequence diagrams and serves as a key concept to create a new version of sequence diagram.

***Keywords*—**System behaviors, sequence diagram merging, dependence analysis, sequence diagram slicing.

I. Introduction

PRACTICAL software systems are constantly changing in response to changes in user needs and the operating environment. This arises when new requirements are introduced into an existing system, specified requirements are not correctly implemented, or the system is to be moved into a new operating environment. One way to cope with this problem is to manage individually each change in a separate and independent way leading to a new version by merging those changes.

The need to merge 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 [1], [2].

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 merging is required for a large variety of artifacts, it appears particularly necessary for requirements. This is because requirements are especially prone to change and evolution [3]. Due to the increasing size and complexity of software applications, the design, and specification have become an important activity in the software life cycle [4].

Interaction-based specifications such as UML sequence diagrams have been found effective in this regard, as they describe system requirements in the most intuitive way. A sequence diagram captures dynamic aspects of a system by means of messages and corresponding responses of collaborating objects. In other words, method calls, parameters, return values, and the collaborating objects can be explicitly modeled in a sequence diagram. As a result, sequence diagrams can be used not only for capturing system behaviors but also for merging changes in order to create a new version. In addition, earlier changes are introduced into the life cycle and easier is their understanding by designers. In this way, two versions of a concurrently evolved sequence diagram have to be combined into one consolidated, correct sequence diagram using information from the original sequence diagram and the associated variants. This includes addressing problems like (1) understanding what a sequence diagram does and how it works, (2) capturing the differences between several sequence diagrams, and (3) creating new sequence diagram by combining pieces of old sequence diagrams.

The objective of this paper is to suggest an approach to overcome these problems by using dependency analysis with the concept of slicing. Dependency analysis is a technique that facilitates the understanding while slicing captures, formally, all mapping and differences between elements of sequence diagrams and serves as a key concept to create a new version of sequence diagram. This paper will show the applicability of this algorithm through an appropriate example.

The remainder of this paper is structured around the following sections. The related works are described in Section II. Section III is dedicated to the concepts needed in this work and the running example to be used throughout this paper. Section IV details our approach by presenting the general algorithm, its formalization, and its applicability via the running example.

II. Related Works

Software engineering research deals extensively with model merging. We review some recent approaches here. There is a large volume of work on merging software code; we do not discuss this work. A complete state of the art is in [1]. Here we discuss about merging diagrams.

A primitive way to merge diagrams was to translate them into plain text (e.g. XML). Viewing diagrams as plain text is not very helpful for differencing and merging [5]. Text-based

Z.E.Bouras and A. Talai are with the Department of Mathematics and Computer Sciences, EPST Annaba Algeria (Phone: 00213560369147, 00213542370893 e-mail: z. bouras@epst-annaba.dz, a.talai@epst-annaba.dz).

tools for differencing and merging are sensitive to changes of the order in which lines appear in a text file. Therefore, structure-based algorithms and tools are required for differencing and merging of software diagrams.

Segurain [6] provides a framework for merging graph transformation systems. Each transformation system is given by a type graph and a set of rewrite rules. A mapping between a pair of graph transformation systems is made up of a mapping between their type graphs and a set of mappings between their rules. Rule mappings are required to satisfy certain properties to avoid undesirable interactions between different rules. The merge operation is characterized using pushouts.

Sabetzadeh [7] describes an approach for merging state machines. A state machine is represented as a set of states, a set of transitions between states, and a set of variables whose values vary from state to state. A mapping between a pair of state machines consists of two parts, a signature map and a truth map. The signature map describes the correspondences between the state machines and further establishes a common vocabulary for the merge. The notion of mapping results in a straightforward binary merge algorithm for state machines.

Letkeman [8] provides a generic approach for merging diagrams in the UML notation. Given a pair of diagrams, the approach first finds the differences between the diagrams and their common ancestor. The differences are described as a sequence of elementary transformations for creating, deleting, and modifying diagram elements. To construct a merge, the differences are applied to the common ancestor. The work provides a practical tool for merge and offers interesting insights about the challenges presented by model merging in a production environment. Mehra [9] also independently, proposes a tool-supported approach for merging graphical diagrams based on computing differences and incorporating them into a common ancestor. But, in contrast to [8], the approach conceives of conflict resolution during merge as an entirely manual process.

Despite their versatility, all related works omit the consequences of dependence analysis between elements in a given UML diagram. Dependence analysis involves the identification of interdependent elements of a system. It is referred to as a "reduction" technique, since the interdependent elements induced by a given inter-element relationship forms a subset of the system [10]. Associated with a lattice (slicing) dependence analysis provides a mathematical characterization of the merge operation.

Our approach of merging is based on this association and applied to merge a specific UML diagram that is sequence diagram.

III.Basic Concepts

A. A Motivating Example

The following example about an ATM sequence diagram is to motivate the approach developed in this paper.

1. Base Sequence Diagram

In the initial ATM system (Base) a session is started when a customer inserts an ATM card into the card reader slot of the machine. The ATM pulls the card into the machine and reads it. The customer is asked to enter his/her PIN, and is then allowed to perform one or more transactions, choosing from a menu of possible types of transaction in each case. The customer chooses a withdrawal transaction. An appropriate menu is then displayed to choose an amount from a menu of possible amounts. Customer enters amount. The system verifies that its balance is sufficient by sending the transaction to the bank. If the transaction is approved by the bank, the appropriate amount of cash is dispensed by the machine and debited immediately from the account before it issues a receipt and, finally eject the card. Concerned sequence diagram is depicted by Fig. 1.

Starting from an initial Base Sequence Diagram of ATM System "Withdraw" Scenario of the suggested example, we introduce two independent requirement changes that are expected to be compatible (non-interfering). For this purpose, two independent copies of Base are first created and then modified (Variant A and Variant B).

2. Variant A Sequence Diagram

In Variant A, the Read Card action is improved by the fact that if the reader cannot read the card due to improper insertion or a damaged stripe, the card is ejected, an error screen is displayed, and the session is aborted. In addition, a customer can have several accounts. After entering the adequate PIN, an appropriate menu is then displayed to choose an account from a menu of possible accounts. Customer chooses the concerned account. The objective is to integrate these new requirements in order to create a new version of Base.

In Variant A, software designer A inserts new messages to take account the concerned own changes. This leads to the following changes: (1) add a message "cannot read card" between objects Card Reader and ATM Screen just after "Read Card" method, and (2) add a message "Select Account" between Customer actor and ATM Screen object after entering PIN code. Fig. 2 shows the sequence diagram of designer A. Changes according to Base are depicted in red arrows and characters.

3.Variant B Sequence Diagram

In Variant B and in cases where balance or cash reserve are insufficient error screens are displayed and customer is asked to enter a new amount. Software designer B proceeds to the following changes: (1) inserts a method "Verify cash reserve" between objects Bank and Cash Dispenser just after "Withdraw Fund" method, (2) inserts a message "cash reserve insufficient" between objects Bank and ATM Screen after verifying cash reserve, and (3) inserts a message "Balance insufficient" between objects Bank and ATM Screen after verifying balance. Fig. 3 represents the sequence diagram of designer B. Changes according to Base are depicted in blue arrows and characters.

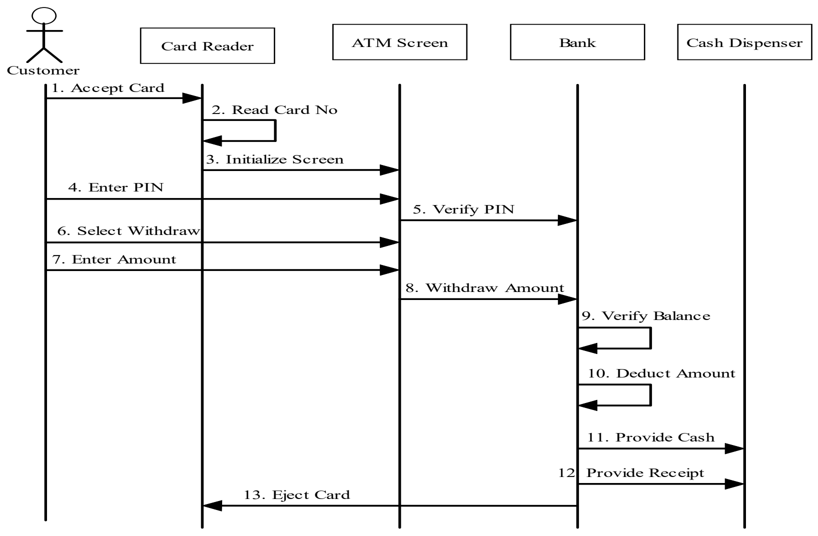

Fig. 1 Base Sequence Diagram of ATM "Withdraw"

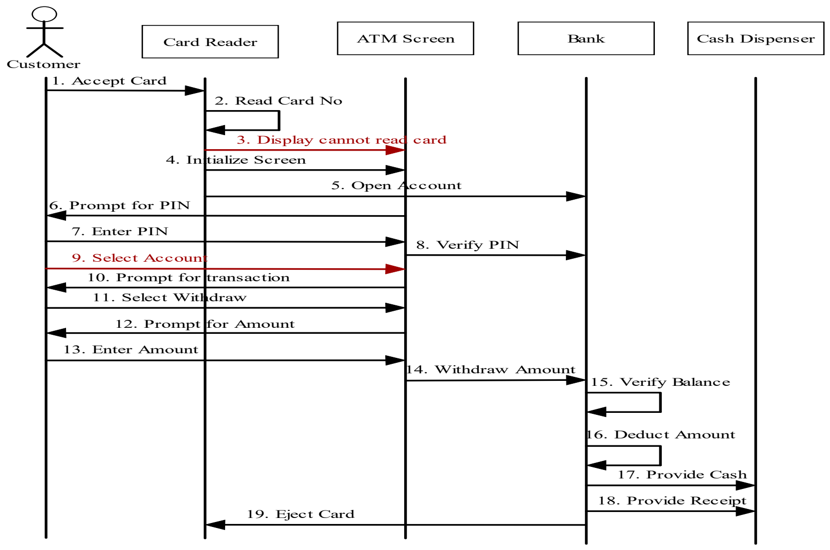

Fig. 2 Variant A Sequence Diagram of ATM "Withdraw"

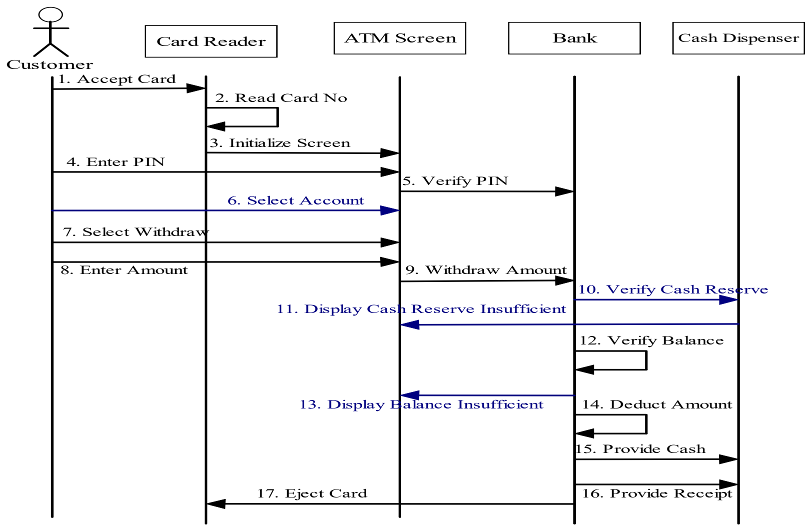

Fig. 3 Variant B Sequence Diagram of ATM "Withdraw"

B. Concepts

1. Dependency Analysis

Dependency analysis is a useful technique that has many applications in software engineering activities including software understanding, testing, debugging, maintenance, and evolution [11]-[13]. Dependence analysis involves the identification of interdependent elements of a system. It is referred to as a "reduction" technique, since the interdependent elements induced by a given inter-element relationship forms a subset of the system [10]. So it is very important to understand element's context and its running environment in order to efficiently manage all kinds of dependencies. In general, as soon as a new element is installed/removed/updated in given software, it has an impact on a part of the system. The new element may refer to certain elements, and be used by other elements [14], [15].

2. Sequence Diagram Graphs

UML consists of nine kinds of different diagrams that can be combined together to provide a complete picture of a system. The diagrams include use case, class, object, sequence, collaboration, state, activity, component, and deployment diagrams. Among them sequence diagram refers to time dependent sequences of interactions between objects. They show the sequence of the messages.

Sequence Diagrams have many advantages, but they are not directly amenable to formal manipulations. Since Sequence Diagrams are diagrammatic, a formalization based on graph-based structures seems to be advantageous. Indeed, graph-based formalisms possess the following desirable properties: they (i) provide a foundation for a large class of software specifications, e.g., Sequence Diagrams, (ii) have solid theoretical foundations, and (iii) have tool support. Thus, graph-based structures are amenable to effective manipulations. Moreover, a Sequence Diagram has an (implicit) structural base that is a set of interacting objects and the types of messages can exchange. A transformation as a graph makes both the behavior and the structural base explicit.

Thus, we use the transformation of sequence diagram as a graph representation proposed by [16]. In such approach, the Model Flow Graph (MFG) represents the possible message/method sequences in an interaction. A MFG can be viewed as a graph G= (V, E), where V is a set of nodes of G, and E is a set of edges. The nodes of G represent messages and edges represent transition between two nodes exists, if the corresponding messages in the sequence diagram occur one after the other. The message that initiates the interaction is made the root of the graph.The MFG for *sequence diagram* is created by (1) Associating methods in the sequence diagram with their originating objects (by using object Method Association Table), and (2) Traversing the sequence diagram from beginning to end, showing choices and condition for method execution. For example, Table I is the Object Method Association Table of Base Sequence Diagram (Fig. 1) and the resulting Model flow Graph is presented in Fig. 4.

TABLE I
OBJECT METHOD ASSOCIATION TABLE OF FIG. 1

Symbol	Object Method Association
A	Card Reader: Accept Card
B	Card Reader: Read Card No
C	ATM Screen: Initialize Screen
D	ATM Screen: Prompt for Pin
E	Bank: Verify PIN
F	ATM: Ready for Withdraw
G	ATM: Ready for Amount
H	Bank: Withdraw Amount
I	Bank: Verify Balance
J	Bank: Deduct Amount
K	Cash Dispenser: Provide cash
L	Cash Dispenser: Provide Receipt
M	Card Reader: Eject Card

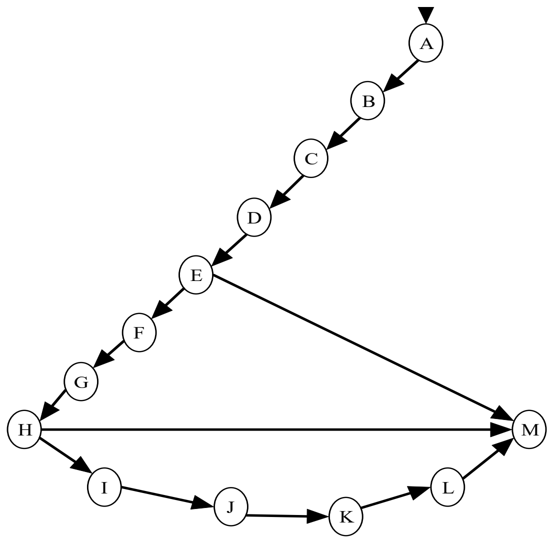

Fig. 4 MFG of Base Sequence Diagram of ATM "Withdraw" Scenario

3. Sequence Diagram Slicing

When a maintenance programmer wants to modify a component in order to satisfy new requirements, the programmer must first investigate which components will affect the modified component and which components will be affected by the modified component. By using a slicing method, the programmer can extract the parts containing those components that might affect, or be affected by, the modified component. This can assist the programmer greatly by providing such change impact information.

Using sequence diagram slicing to support change impact analysis promises benefits for sequence diagram evolution. Slicing is a particular application of dependence graphs. Together they have come to be widely recognized as a centrally important technology in software engineering. Because they operate on the deep structure in programs rather than surface structures, they enable much more sophisticated and useful analysis capabilities than conventional tools [7].

Traditional slicing techniques cannot be directly used to slice sequence diagram. Therefore, to perform slicing at the sequence diagram level, appropriate slicing notions must be defined with new types of dependence relationships.

To calculate a slice it is first required to transform a sequence diagram into a suitable intermediate representation that is Model Flow Graph (MFG).A slice for a specific scenario can be computed by identifying the different elements and the dependencies among them from MFG. These elements are identified based on a certain condition termed as a slicing criterion. In our case we use slicing technique to find out slices at each message point in a sequence diagram. A slice contains only those parts of a sequence diagram that actually affect a value at each message point. In our case the slicing criterion (m, V) specifies a message location m in the sequence diagram and V is the set of all locations that are used in the location m. For example, if we want to find the set of all locations (H) that affect the "Withdraw Amount" from Bank object (m) by applying slicing concept we proceed as the following simple algorithm through the MFG: "initialize an empty set of message locations m. Start by adding all nodes preceding *m* that have a direct link with m. For any node, add all message locations that precede *m* and that affect a required property of this message location. Repeat this procedure until no message locations are found". From MFG of Fig. 4, we obtain the following sub graph (Fig. 5) that reflects all actions that affect "Withdraw Amount" action. "Withdraw Amount" (H)action is affected by: Accept Card (A), Initialize Screen (B), Prompt for Pin (C), Verify PIN (D), Ready for Withdraw (H), and Ready for Amount (H).

4. Graph Similarities

Comparing two graphs needs at first to find, for a given node (or edge) in a graph, its corresponding node (or edge) in the other; this can be done by signature and structural matching [17].

A pair of corresponding elements needs to share a set of properties, which can be a subset of their syntactical information. Such properties may include type information, which can be used to select the elements of the same type from the candidates to be matched because only elements with the same type need to be compared. Therefore, a combination of syntactical properties for a node or an edge can be used to identify different elements. Such properties are called the *signature*, and are used as the first criterion to match elements as proposed by [17].

The algorithm first needs to find all the candidate nodes in MFG2 that have the same signature as node v1 in MFG1. If there is only one candidate found in MFG2, the identified candidate is considered as a unique mapping for v1 and they are considered as syntactically equivalent. If there is more than one candidate that has been found, the signature cannot identify a node uniquely. Therefore, v1 and its candidates in MFG2 will be sent for further analysis where structural matching is performed. Structural matching is based on calculation of Graph Similarity using Maximum Common Edge subgraphs [18]. The first algorithm to find the candidate node with maximal edge similarity for a given host node from a set of candidate nodes takes the host node and a set of candidate nodes of N2 as input, computes the edge similarity of every candidate node and returns a candidate with maximal edge similarity. The second algorithm for computing edge similarity between a candidate node and a host node takes two maps as, input, stores all the incoming, in, and outgoing edges of the host and candidate nodes indexed by their edge signature. By examining the mapped edge pairs between these two maps, the algorithm computes the edge similarity as output.

All nodes in N1 have been examined by signature and structural matching; all possible node mappings between N1 and N2 are found.

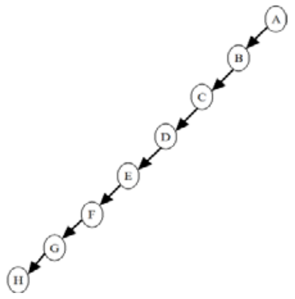

Fig. 5 Slicing MFG with slice criterion ("Withdraw Amount", MFG)

IV. Approach of Merging

Our merging process consists of the following steps: (1) start from a Base Sequence Diagram, (2) build a set of variants (resulting from Base changes), (3) compare each variant with the Base, (4) determine the sets of changed and preserved slices, and (5) combine these sets to form a single integrated new version (if changes don't interfere). Steps (1) and (2) are done concurrently by software architects, details of steps (3) and (4) are described below.

Step3. Compare each variant with the base:

a. Build the MFGs of the Sequence Diagrams Base and variants.
b. Extract, from each MFG, its associated slices.
c. From each Variant MFG, determine peer nodes according to Base MFG by using graph similarities
d. For each Variant MFG
d.1. Map each slice of the Base MFG with its peer in variant.
d.2. Determine and collect changed and preserved slices.

Step4. Combine changed and preserved slices to form a new MFG.

a. Merge preserved of base and changed slices of variants.
b. Check that variants do not interfere
c. Derive the resulting MFG.
d. Generate the Sequence Diagram of the new version from the resulting MFG.

Steps 3.a and 3.b have already been solved, in [19]. Our contribution in this paper is to develop the sub-steps from 3.c until the end of the process in order to merge sequence diagrams. In the following, we formalize these sub-steps. We formalize and illustrate each step through our motivating example of Section III.

A. Build MFGs of Sequence Diagrams

MFG of Base (MFG_{Base}) was constructed previously in Section III (Fig. 4) and with the same manner we construct the Model Flow Graph MFG_A and MFG_B of Variant A and B respectively. Fig. 6 illustrates the MFG_B of Variant B, sequence diagram done by Software designer B.

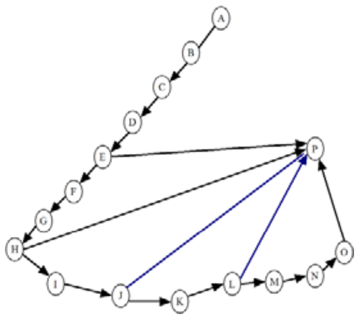

Fig. 6 MFG of Variant B (MFG_B)

B. Extract Associated Slices

Any slice can be computed by identifying the different elements and the dependencies among them from MFG. These elements are identified based on a certain condition termed as a slicing criterion. In our case we use slicing technique to find out slices at each message point in a sequence diagram, that is at each node in MFG. A slice contains only those parts of a sequence diagram that actually affect a value at each message point. In our case the slicing criterion (m, V) specifies a message location m in the sequence diagram and V is the set of all locations that are used in the location m. We proceed as the following simple algorithm through the MFG: "initialize an empty set of message locations m. Start by adding all nodes preceding *m* that have a direct link with m. For any node, add all message locations that precede *m* and that affect a required property of this message location. Repeat this procedure until no message locations are found". Fig. 5 of Section III reflects an example of slice extraction.

C. Determine Peer Nodes Using Graph Similarities

Comparing two graphs needs at first to find, for a given node (or edge) in a graph, its corresponding node (or edge) in the other; this can be done by signature and structural matching [17]; this was detailed in Section III.

Let (node1, node2), where node1 is a node in Base and node2 a node in a variant. (node1, node2) denotes that it exists a node similarity between node1 and node2.

Node similarities between Base and Variant A are the following sets:

SimBase_VarA= {(A,A), (B,B), (C,D), (D,E), (E,F), (F,H), (G,I), (H,J), (I,K), (J,L), (K,M), (L,N), (M,O)}.

C and G are new nodes in Variant A.

SimBase_VarB= {(A,A), (B,B), (C,C), (D,D), (E,E), (F,F), (G,G), (H,H), (I,K), (J,M), (K,N), (L,O), (M,P)}.

I, J, and L are new nodes in Variant B.

D. Determining and Collecting Changed and Preserved Slices

Given MFGs MFG_{Base}, MFG_A, and MFG_B, the algorithm performs three steps. The first step identifies three subgraphs that represent the changed behavior of *A* with respect to *Base*($\Delta_{A,\ Base}$), the changed behavior of *B* with respect to *Base* ($\Delta_{X,\ Base}$) and the preserved behavior that is the same in all MFGs ($Pre_{A,B,Base}$) by using the set of vertices whose slices in MFG_{Base}, MFG_A, and MFG_B are identical (i.e. $PP_{A,B,Base}$). The second step unifies these subgraphs to form a merged model dependence graph MFG_M. In the third step, a merged sequence diagram G_M is produced from graph MFG_M.

1.Changed Slices

Let $\Delta_{X,\ Base}$the set of all changed slices between the variant X and Base. Changed slices are computed as the following:

$$AP_{A,\ Base}=\{v \in V(MFG_A)\ |(MFG_{Base}/v) \neq (MFG_A/v)\}$$
$$AP_{B,\ Base}=\{v \in V\ (MFG_B)\ |(MFG_{Base}/v) \neq (MFG_B/v)\}$$
$$\Delta_{A,\ Base}= b(MFG_A,\ AP_{A,\ Base})$$
$$\Delta_{B,\ Base}=b(MFG_B,\ AP_{B,\ Base}).$$

where, $V(MFG_x)$ denotes the set of vertices in MFG of variant X.MFG_X/v is a vertex in the MFG of X from where we want to inspect its impact in the overall MFG of X. $b(MFG_X, AP_{X,\ Base})$ is the set of peer changed slices when comparing MFG_{Base} and MFG_X. For example, slicing MFG from Base with slice criterion ("Withdraw Amount", MFG_{Base}), in Fig. 6, differs from Slicing MFG from Variant A with slice criterion ("Withdraw Amount", MFG_A) in Fig. 7. Thus, the last one slice belongs to the set of changed slices.

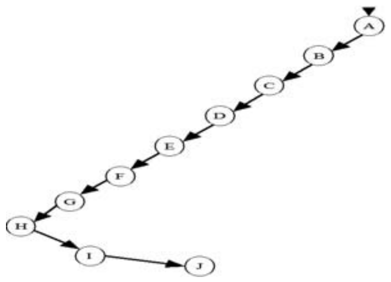

Fig. 7 Slice from Variant A with slice criterion ("Withdraw Amount", MFG_A)

2. Preserved Slices

Preserved MFGs slices ($Pre_{A,\ Base,\ B}$) are computed as:

$$PP_{A,\ Base,\ B} = \{v \in V(MFG_{Base})\ |(MFG_A/v) =(MFG_{Base}/v) =(MFG_B/v)\}.$$
$$Pre_{A,\ Base,\ B}=(G_{Base},\ PP_{A,\ Base,\ B})$$

In our example there is only one slice preserved in all MFGs, that is the sub-graph from node A to node B. This belongs to set of preserved slices.

E. Forming Merged MFG

The merged model flow graph MFG_M characterizes the MFG of the new version of the sequence diagram. G_M is computed as:

$$GM = \Delta_{A,\,Base} \cup \Delta_{B,\,Base} \cup Pre_{A,\,Base,\,B}.$$

Informally, slices that are changed in variants A and B with respect to Base and those that are unchanged in all sequence diagrams form the merged graph GM.

The new version of sequence diagram (Fig. 9) is obtained, finally, from the merged MFG of Fig. 8.

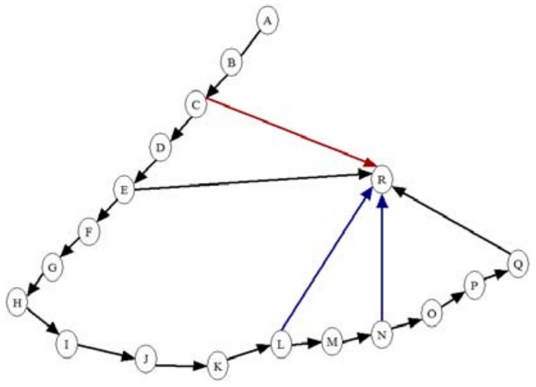

Fig. 8 MFG of the new version

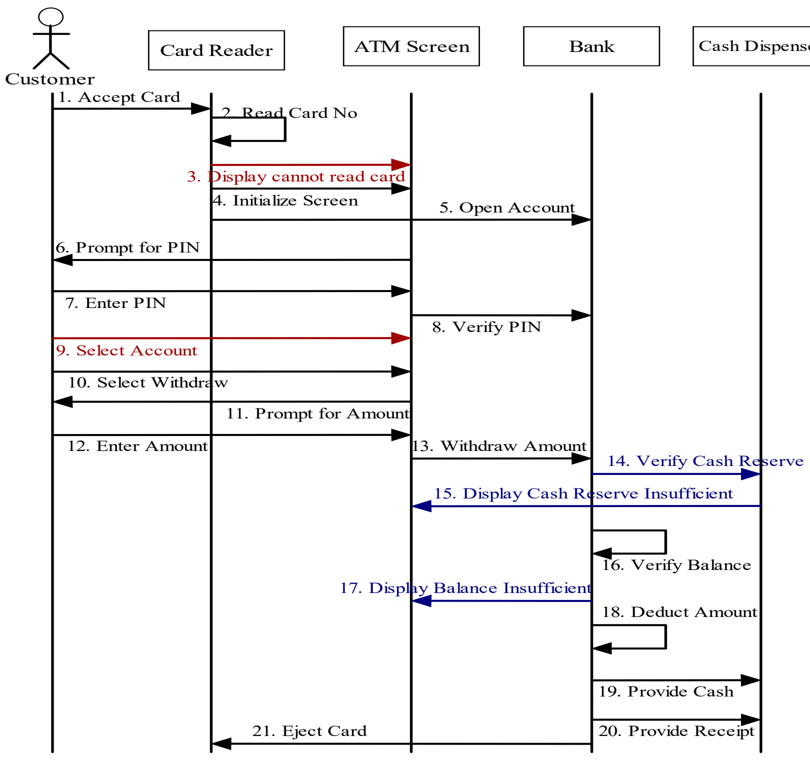

Fig. 9 New version of Sequence Diagram

V. Conclusion

Merging sequence diagram changes approaches allows for concurrent modifications, of the same sequence diagram by several developers, and merging them to obtain ultimately one consolidated version of sequence diagram again. In this paper we have shown that techniques of Software merging, initially defined to cope with program merging and extended to model merging, may be also used to bring the sequence diagram merging issue. The main benefit is that “earlier changes are introduced into the lifecycle and easier is their understanding by designers”.

Beginning with the idea that sequence diagrams are graphs with particular nodes and edges, we capture all mapping and difference between concepts and merge them into a new consolidate version by using a power issue in software engineering that is the dependence analysis. A concrete example illustrated the suggested approach.

References

[1] T. Mens, “A State-of-the-Art Survey on Software Merging”, *IEEE Trans. on Software Engineering*, Vol 28 No 5, pp. 449–462.

[2] P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer, “An Introduction to Model Versioning”, *Proc. Of the 12th international conference on Formal Methods for the Design of Computer, Communication, and Software Systems*, SFM 2012, Springer-Verlag Berlin Heidelberg, pp. 336–398.

[3] H. Liang, Z. Diskin, J. Dingel, and E. Posse, “A General Approach for Scenario Integration”,*Proc. ofthe11th International Conference on Model Driven Engineering Languages and Systems, MoDELS’08*, 2008.

[4] E. Ogheneovo, “On the Relationship between Software Complexity and Maintenance Costs”. *Journal of Computer and Communications*, vol2, pp. 1-16, 2014.

[5] D. Ohst, M. Welle, and U. Kelter, “Differences between versions of UML diagrams”, *Proc. of ESEC/FSE-11*, pages 227–236, New York, NY, USA.ACM Press.

[6] S. Segura, R. M. Hierons, D. Benavides, and A. Ruiz-Cortes, “Automated metamorphic testing on the analyses of feature models”. *Information and Software Technology*, vol53, No 3, pp.245–258, 2011.

[7] M. Sabetzadeh, S. Nejati, S. Liaskos, S. M. Easterbrook, and M. Chechik, “Consistency checking of conceptual models via model merging”. *Proc. of Requirement Engineering, RE 2007.IEEE,*pp. 221-230, 2007.

[8] K. Letkeman, “Comparing and Merging UML Models in IBM Rational Software Architect”, IBM, Aug. 2005.

[9] A. Mehra, J. Grundy, and J. Hosking, “A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design”, *Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE'05*, pages 204–213, 2005.

[10] J.A. Stafford, A. L. Wolf, and M. Caporuscio “The Application of Dependence Analysis to Software Architecture Descriptions”, 3rd*International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures, SFM 2003*, Bertinoro, Italy, September 22-27, pp. 52-62, 2003.

[11] T. Kim, Y. Song, and L. Chung, “Software architecture analysis: a dynamic slicing approach”, *International Journal of Computer & Information Science*, vol 1, no 2, pp. 91-103, 2000.

[12] J. Zhao, “Using dependence analysis to support software architecture understanding”, *CoRR*, vol. cs.SE/0105009, 2001.

[13] B. Li, “Managing Dependencies in Component-Based Systems Based on Matrix Model” Proc. of Net. Object. Days, pp. 22-25, 2003.

[14] B. Li, Y. Zhou, Y. Wang, and J. Mo, “Matrix-based component dependence representation and its applications in software quality assurance” *SIGPLAN Notices*, vol 40, no 11, pp. 29–36, 2005.

[15] J. Lalchandani, “Static Slicing of UML Architectural Models”, *Journal of Object Technology*, vol 8, no 1, pp. 159-188, 2009.

[16] S. Kumar, D. P. Mohapatra, “Test Case Generation from Behavioral UML Models”, *International Journal of Computer Applications,* vol 6, no8, September 2010.

[17] Y. Wang, J. DeWitt, and J. Cai, “X -Diff: An Effective Change Detection Algorithm for XML Documents”, Proc. of19th Intern. Conference on Data Engineering, India, pp. 519-530.

[18] J. Raymond, E. Gardiner, and P. Willett, “RASCAL: Calculation of Graph Similarity using Maximum Common Edge Subgraphs”, *The Computer Journal,* vol45, no 6, pp. 631-644, 2002.

[19] P. Samuel, R. Mall, “Slicing-Based Test Case Generation from UML Activity Diagrams”, *ACM SIGSOFT Software Engineering Notes*, Vol 34 No 6, November 2009.