Extracting Sequence Diagrams from Execution Traces using Interactive Visualization

Hassen Grati, Houari Sahraoui, Pierre Poulin
DIRO, Université de Montréal
Montréal, Canada
{*gratihas, sahraouh, poulin*}*@iro.umontreal.ca*

***Abstract*—We present a semi-automated approach for the reverse engineering of UML sequence diagrams. Our approach starts with a set of execution traces that are automatically aligned in order to determine the common behavior. Sequence diagrams are then extracted with an interactive visualization, which allows navigating into execution traces and performing extraction operations. We provide a concrete illustration of our approach with a case study, and show in particular that the resulting diagrams are more meaningful and more compact than those extracted by automated approaches.**

***Keywords*-reverse engeneering; sequence diagrams; visualization;**

I. Introduction

The reverse engineering of analysis and design models helps improving many activities of software development and maintenance. Examples of these activities include comprehension, migration, maintenance at the model level, etc. In this context, many contributions have been proposed to reverse engineer the static structure of object-oriented (OO) systems [2]. With the exception of a few issues, such as relationship recovery, these contributions are mature enough to be integrated in commercial tools such as *Rational*[1], *Together*[2], and *NetBeans*[3].

This reality contrasts with the difficulty of extracting the behavior models of OO systems, such as sequence diagrams. Indeed, modern programs extensively use dynamic language features (polymorphism, dynamic class loading, dynamic class generation, and reflection) that make it difficult, and often impossible, to capture the behavior by static analysis [14]. To circumvent this limitation, many research teams have adopted an alternative approach based on dynamic analysis [3], [4], [5], [6]. In these approaches, execution traces are used to find the elements of message sequence diagrams.

Although these approaches have improved significantly the quality of the extracted behavior models, two problems remain. First, the extracted models are defined in terms of implementation and do not provide an abstract view. In other words, the extraction process cannot distinguish between abstract information such as communication messages between business objects and implementation details. The second problem of existing approaches is that in many cases, the extracted diagrams correspond to single executions, which limits their generalizability.

In this paper, we present an approach that circumvents the limitations inherent to automation. Instead of fully automating the reverse-engineering process of sequence diagrams, we propose a semi-automated process with the participation of an analyst through an interactive visualization environment. Our environment performs automated actions for which the state-of-the-art knowledge is mature enough. It also recommends actions that the analyst could accept or reject where approximations are used. Finally, the analyst may perform manual actions at any moment of the process using the contextual knowledge.

To evaluate our approach, we performed a study on the case described by Briand et al. [2]. In particular, we compared the sequence diagrams obtained using our approach with those derived automatically. Our results show that with limited time, an analyst extracts sequence diagrams that contain the abstract information without the unnecessary implementation details.

The remainder of this paper is structured as follows. The overview of our approach is described in Section II. Its two main steps are then detailed in Sections III (extraction and alignment of execution traces) and IV (extraction of sequence diagrams by interactive visualization). The results of a case study are reported and discussed in Section V. Related work is outlined in Section VI. Conclusions and future research directions are given in Section VII.

II. Approach Overview

To better describe our approach, we start this section by giving an example that we will use throughout the paper.

A. Running Example

Our running example is a scenario of using a drawing tool[4]. This Java application contains 16 classes and allows

[1]http://www-01.ibm.com/software/rational/
[2]http://www.borland.com/us/products/together/
[3]http://netbeans.org/
[4]http://www.javafr.com/codes/JavaDessin_36623.aspx

the creation of drawings and paintings using a graphical interface. The selected scenario allows drawing geometric shapes. Three executions of this scenario are performed to draw respectively a rectangle, a circle, and a segment. In each case the user provides information such as fill color, border color, size, etc.

B. Overview

Reverse engineering of behavioral models could be done for several reasons such as code analysis, comprehension, and documentation. Our work is motivated by the re- documentation of an existing application. It extracts sequence diagrams for execution scenarios. Concretely, our approach, based on dynamic analysis, follows three steps as shown in Figure 1. First, it generates a set of execution traces corresponding to a specific execution scenario. The traces are obtained by instrumenting and executing the code. We use several execution traces to better capture the variations inside the same scenario.

Afterwards, we align these traces by identifying common execution fragments and situations that are specific to each execution. The alignment algorithm allows also identifying behavior that could be abstracted from similar but not identical execution fragments. Our algorithm is based on the sequence-alignment method of Smith and Waterman [9].

The third step of our approach consists of extracting the sequence diagram using an interactive visualization environment. This environment allows navigating into the combined trace and performing extracting operations.

III. Generating and Aligning Execution Traces

A. Generating Traces from a Scenario

Because our goal is to document the existing use-case scenarios, we make the assumption that the extraction of sequence diagrams is done accordingly to already-existing scenarios. For a particular scenario, we determine the set of executions that help capturing the variations within the scenario. For our running example, the scenario corresponds to drawing a geometric shape using a drawing software. We select three executions corresponding to three different geometric shapes: rectangle, circle, and segment. These shapes were chosen because they have different properties.

Before starting the execution, the code is instrumented to capture information that is difficult to extract otherwise. To this end, we inject notification statements for the start and the end of each method, each loop block, and each conditional block. These notification statements allow us to determine where each method call is performed and how the calls are organized. Consequently, the obtained trace is a sequence of method calls, where each method call contains the following information:

- Type and memory address of the message-sender object.

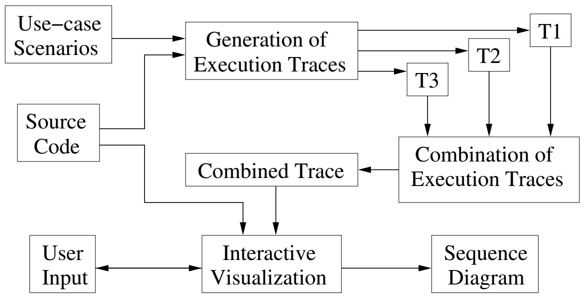

Use–case Scenarios
Generation of Execution Traces
T1
T2
T3
Source Code
Combined Trace
Combination of Execution Traces
User Input
Interactive Visualization
Sequence Diagram

Figure 1. Overview of our approach.

- Type and memory address of the message-receiver object.
- Method name and a generated execution identifier of the calling method. This identifier allows us to differentiate between two executions of the same method.
- Method name and execution identifier of the called method.
- Stack of conditional and loop blocks where the method call is performed.

Figure 2 shows an example of execution traces corresponding to the drawing of a circle. The first event corresponds to the execution of the first method *StartDraw* by the object *PanelDraw* having memory address 21668571. For this call, there is no caller object or calling method. This is indicated by the character "_". The method *StartDraw* calls three methods: the constructors *Figure* (event 2) and *Circle* (event 9), and the method *insert* (event 11). To differentiate between two executions of the same method, the execution of *StartDraw* is assigned the identifier [T1M1] that is used in the three calls (events 2, 9, and 11). Finally, methods *Circle* and *draw* in events 9 and 10 are called inside the execution of the conditional block (*State.getFiguretype()==MODE_CERCLE*).

B. Combining Traces

The next step of our approach combines execution traces. Use-case scenarios generally define abstract execution possibilities of a particular system. The same scenario could have different variations that produce different yet similar execution traces. In our example, it is not realistic to have a scenario for every type of geometric shape. To document the program, one scenario of drawing a geometric shape is enough. For this reason, it is interesting to execute the scenario with different variations. Figure 3 shows as an example the trace corresponding to the drawing of a rectangle. In addition to the fact that a rectangle is instantiated instead of a circle, the method *getThickness* is not called because the thickness value was not present in the dialog box.

To help identify the common behavior and differences between different executions of the same scenario, we align the corresponding traces. When more than two traces are

1	_, PanelDraw [21668571], _, StartDraw [T1M1], _;
2	PanelDraw [21668571], Figure [3916193], StartDraw [T1M1], Figure [T1M2], _
3	Figure [3916193], State [23930626], Figure [T1M2], getTransparency [T1M3], _
4	Figure [3916193], State [23930626], Figure [T1M2], getMode [T1M4], _
5	Figure [3916193], State [23930626], Figure [T1M2], getLineColor [T1M5], _
6	Figure [3916193], State [23930626], Figure [T1M2], getThickness [T1M6], _
7	Figure [3916193], State [23930626], Figure [T1M2], getFillingColor [T1M7], _
8	Figure [3916193], State [23930626], Figure [T1M2], getFigureType [T1M8], _
9	PanelDraw [21668571], Circle [17282414], StartDraw [T1M1], Circle [T1M9], <%(State.getFiguretype()==MODE_CERCLE)%>
10	Circle [17282414], Figure [3916193], Circle [T1M9], draw [T1M10], <%(State.getFiguretype()==MODE_CERCLE)%>
11	PanelDraw [21668571], Figure [3916193], StartDraw [T1M1], insert [T1M11], _

Figure 2. Fragment of execution trace of drawing a circle. The event number is provided to the left.

1	_, PanelDraw [64836544], _, StartDraw [T2M1], _
2	PanelDraw [64836544], Figure [9826352], StartDraw [T2M1], Figure [T2M2], _
3	Figure [9826352], State [19286486], Figure [T2M2], getTransparency [T2M3], _
4	Figure [9826352], State [19286486], Figure [T2M2], getMode [T2M4], _
5	Figure [9826352], State [19286486], Figure [T2M2], getLineColor [T2M5], _
6	Figure [9826352], State [19286486], Figure [T2M2], getFillingColor [T2M6], _
7	Figure [9826352], State [19286486], Figure [T2M2], getFigureType [T2M7], _
8	PanelDraw [64836544], Rectangle [1271231], StartDraw [T2M1], Rectangle [T2M8], <%(State.getFiguretype()==MODE_RECTANGLE)%>
9	Rectangle [1271231], Figure [9826352], Rectangle [T2M9], draw [T2M9], <%(State.getFiguretype()==MODE_RECTANGLE)%>
10	PanelDraw [64836544], Figure [9826352], StartDraw [T2M1], insert [T2M10], _

Figure 3. Fragment of execution trace of drawing a rectangle.

combined, the combination is done incrementally. In our example, we align first the rectangle and circle traces, and then the resulting trace with the segment trace.

As the execution traces represent trees where each node is a method call, the alignment algorithm implements a recursive process where the trees are compared level by level. Two nodes are aligned if they have the same types of sender and receiver objects, the same names for respectively calling and called methods, and the same stack of conditionals and loops. Moreover, two nodes can be compared if their respective parents are already aligned.

The algorithm starts by comparing the root nodes of the two traces. If the two nodes are aligned, then their child nodes are compared. Afterwards, the algorithm is recursively applied to each pair of aligned nodes.

In each trace, the child nodes represent the method calls performed within the method of a parent node. As the calls are ordered, they define a sequence. To compare the sequences of child nodes in the two traces, we use the Smith- Waterman sequence-alignment algorithm [9]. This algorithm, based on dynamic programming, explores all possible alignments between two strings, using substitution matrices to generate an optimal alignment between two sequences of characters. We adapted this algorithm to align call sequences.

The Smith-Waterman algorithm is performed recursively to store the score of the already-matched sub-sequences. When aligning two sequences $(a_1, ..., a_n)$ and $(b_1, ..., b_m)$, we define a substitution matrix M having $n+1$ lines and $m+1$ columns. Each position $M_{i,j}$ corresponds to the best score of alignment considering the previously aligned elements of the sequences. The algorithm can introduce gaps (represented in the following example by "_") to improve the matching of sequences. Formally,

$$M_{i,0} = M_{0,j} = 0$$

$$M_{i,j} = \begin{cases} M_{i-1,j-1} + match(a_i, b_j) \\ M_{i-1,j} + gap(a_i, \_) \\ M_{i,j-1} + gap(\_, b_j) \end{cases}$$

$match(a_i, b_j)$ defines the score of matching two characters a_i and b_j. When a gap is inserted before a_i (resp. b_j), it incurs a penalty of $gap(a_i, \_)$ (resp. $gap(\_, b_j)$).

Our alignment algorithm adaptation is straightforward. We simply set the gap penalty to -1 and use our node (method-call) alignment to measure the matching score. Thus, for two nodes a_i and b_j, $match(a_i, b_j)$ is set to 1 if a_i and b_j could be aligned, and to 0 otherwise.

The alignment of traces T1 and T2 in Figure 2 and Figure 3 respectively is presented in Figure 4. For the sake of conciseness, we identify nodes by: "a" <trace number> "-" <event number>. In the combined trace, nodes are labeled by the numbers of the traces where they appear. For example, node a1-3 [1,2] in T1+T2 indicates that the corresponding method call occurs in T1 (a1-3) and T2 (a2-3). Because memory addresses differ from one trace to another, we keep by convention those of the first trace, which explains why we reuse the node names of T1 in T1+T2.

To illustrate child-node sequence alignment, let us consider methods a1-3 to a1-8 called by a1-2 in T1 and methods a2-3 to a2-7 called by a2-2 in T2. These two sequences can be compared because a1-2 and a2-2 are already aligned. Figure 5 shows the best alignment between the two sequences. A unique gap is introduced in T2 to compensate for the absence of the call of *getThickness*.

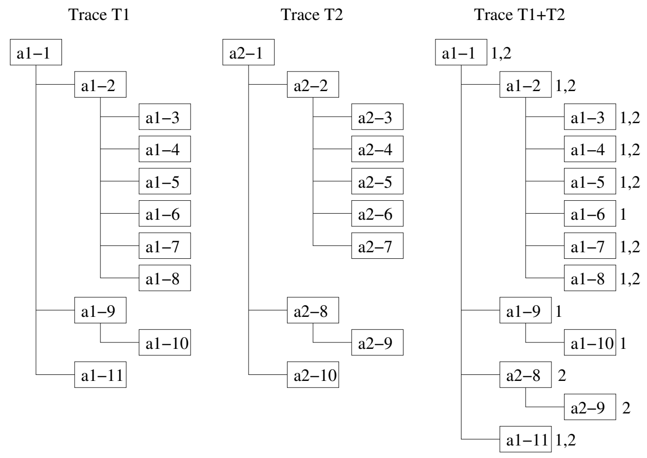
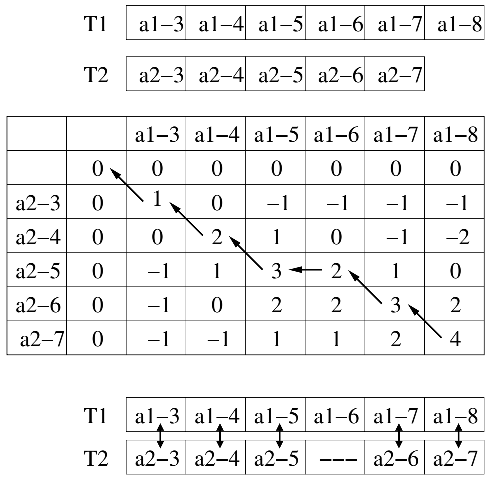


Figure 4. Example of combined traces.



T1	a1−3	a1−4	a1−5	a1−6	a1−7	a1−8
T2	a2−3	a2−4	a2−5	a2−6	a2−7	

		a1−3	a1−4	a1−5	a1−6	a1−7	a1−8
	0	0	0	0	0	0	0
a2−3	0	1	0	−1	−1	−1	−1
a2−4	0	0	2	1	0	−1	−2
a2−5	0	−1	1	3	2	1	0
a2−6	0	−1	0	2	2	3	2
a2−7	0	−1	−1	1	1	2	4

T1	a1−3	a1−4	a1−5	a1−6	a1−7	a1−8
T2	a2−3	a2−4	a2−5	−−−	a2−6	a2−7

Figure 5. Child-node sequence alignment.

IV. Sequence Diagram Extraction using Interactive Visualization

In this section, we detail the process of sequence diagram (SD) extraction. This process is a set of successive cycles that contains automatic transformations and user interactions to modify and complete these transformations. Each cycle represents a step in the exploration of the combined trace. Before explaining the visualization environment, the navigation actions, and the possible interactions, we start by giving the basic transformations from the combined trace to the sequence diagram.

A. *Trace-to-SD Basic Transformation*

When exploring the combined trace, each method call is transformed by default as follows:

- A method call is mapped to a message in the SD.
- The source of a message is the lifeline of the participant corresponding to the object executing the method when the call occurs.
- If the called object exists already as a participant in the diagram, the message is related to its lifeline. Otherwise, a new participant is created with the memory address of the called object as ID and the message connected to its lifeline.
- opt/alt/loop boxes are drawn progressively to encapsulate messages according to the conditional/loop stacks associated with their corresponding method calls.
- Return messages are inserted in the graph according to the tree structure of the combined trace. These messages connect the receiver and called participants of the associated methods.

B. *Interaction Views*

At each step of the SD extraction process, the analyst is able to visualize the already processed fragment of the combined trace (herein called *global view*) and its corresponding SD fragment (called *diagram view*). In the following subsections, we detail these two views.

1) Global View: The size of execution traces makes their visualization particularly difficult. We believe however that the analyst makes better decisions when he can view the traces at least partially. Consequently, we decided to visualize the combined trace progressively as it is traversed.

Representing methods and method calls: Methods are represented by cylinders laid out in a plane in a grid-like structure. Each cell in the grid could host a cylinder. Methods are colored in gray. The method in control of the execution is colored in blue.

We represent a method call with an arc from the calling-method cylinder to the invoked-method cylinder as shown in Figure 6(a). The arc representation avoids crossing links, as the height of the arc depends on the distance between cylinders and the height of the other arcs.

The representation of a call indicates also the direction of the call and the traces where it appears. The directions are represented by long isosceles triangles whose base is attached to the calling method and opposite tip vertex to the invoked method. The triangle is drawn as strips colored according to the traces (one color per trace). In Figure 6(a), the coloration of the arc indicates that it is present in three traces corresponding to the colors.

Although we represent the combined trace progressively, the view could still become cluttered after traversing a significant part of the trace. To help the analyst focus on the most recent calls while keeping the global context, we reduce the width of an arc as the age of the corresponding call grows. We determine the age of a call by the number of calls that separate it from the current call in the traversing process. The width reduction is illustrated in Figure 6(c).

Method returns are represented by similar arcs whose width is reduced with the age (Figure 6(b)). The only difference is that we use a dashed triangle with a unique color to help the analyst distinguish between calls and

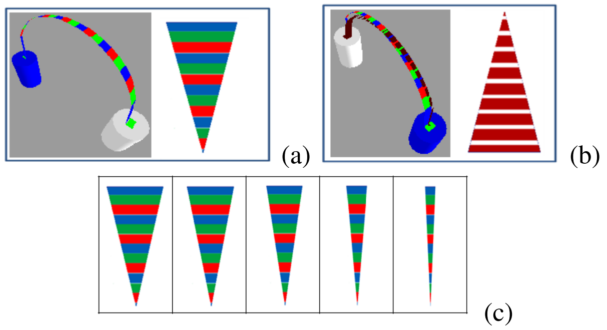


Figure 6. (a) Method-call representation, (b) method-return representation, and (c) width reduction according to age.

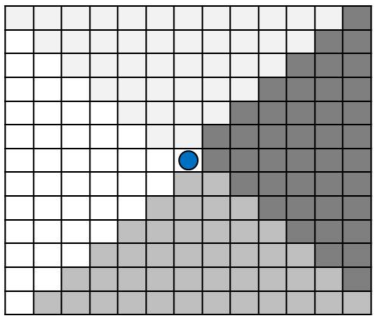


Figure 7. First step of the layout algorithm.

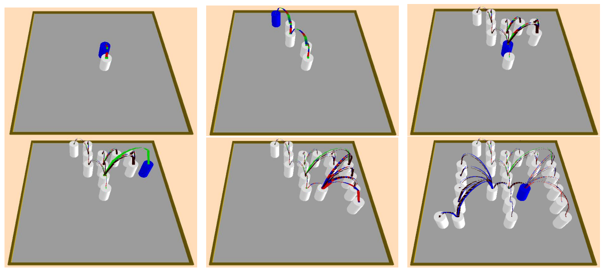


Figure 8. Snapshots of the evolution of the global view.

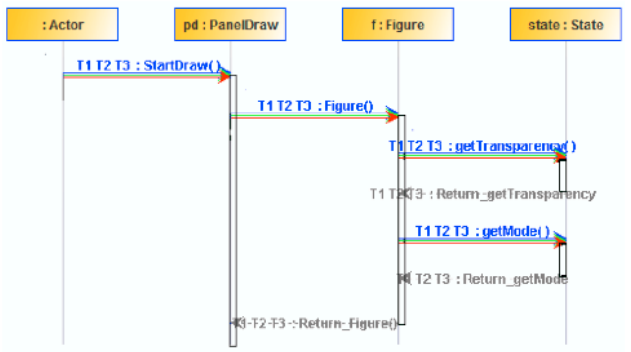

: Actor
pd : PanelDraw
f : Figure
state : State
T1 T2 T3 : StartDraw( )
T1 T2 T3 : Figure()
T1 T2 T3 : getTransparency( )
T1 T2 T3 : Return_getTransparency
T1 T2 T3 : getMode( )
T1 T2 T3 : Return_getMode
T1 T2 T3 : Return_Figure()

Figure 9. A portion of an SD view.

returns. As a call and its corresponding return are between the same methods (but in opposite directions), it is not necessary to indicate the traces where the return happens.

Placement algorithm: As stated earlier, the global view allows visualizing the combined trace progressively. This trace is pruned during the traversing process. Indeed, the analyst is able to remove messages and objects that are useless for the abstract behavior desired in the sequence diagram. The interactive pruning is detailed in Section IV-A.

The pruned trace is a tree. To place the nodes of this tree, i.e., cylinders that correspond to method executions, we first place the root node in the center of the grid. Then the grid is divided into four portions using the diagonals as shown in Figure 7. The children of the root node are assigned each one of the four infinite triangular regions. If after the pruning, the root node has more than four children, the triangular regions are recursively divided into narrower triangular regions. When each child is assigned to a region, the sub-tree defined by each of the child nodes are placed according to the positioning algorithm of Walker [10]. Walkers algorithm allows recursive positioning of tree nodes using the classical tree structure in two steps. As we use a grid in our case, we place the nodes of a level in a layer of the region assigned to the sub-tree. If the layer does not contain enough cells, we jump to the next layers until all the nodes are placed. The size of our grid is not fixed a priori. Layers can be added if we need more space.

Figure 8 shows snapshots of the positioning during the traversing of the combined trace.

2) Diagram View: As we traverse the combined trace, we draw the sequence diagram according to the automated transformations of Section IV-A and to the actions performed by the analyst.

We use the standard notations of UML 2.0 [11] for drawing the SD. To help the analyst make better decisions, we add information about the traces from which the messages are extracted. This is done by duplicating the message arrows as many times as the number of traces. Each arrow is colored according to its trace. The identifiers of the traces are also added before the message label.

Figure 9 shows a portion of the SD corresponding to our running example. The first participant is added to indicate the actor that triggers the scenario of drawing a geometric shape. The other three participants are the objects created during the execution. The four displayed messages exist in the three traces T1, T2, and T3.

C. Interactions

The extraction of the SD could be seen as a progressive pruning of the combined trace. In the following subsections, we present how the analyst navigates into the combined trace and how he modifies the traces.

1) Navigation in the Combined Trace: In default mode, the traversal of a trace is done according to the execution events. As the trace corresponds to a tree, this is equivalent

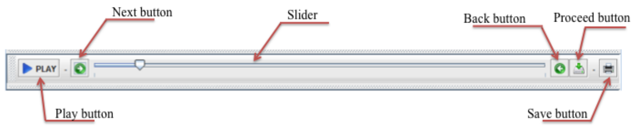


Figure 10. Navigation bar.

Table I
NAVIGATION BAR COMMANDS.

Button	Description
Play	Initialize and launch the navigation and the creation of the sequence diagram.
Next	Show the next method call in the global view and add the corresponding elements in the SD view.
Back	"Undo" the last navigation step by removing in the two views elements related to the corresponding call.
Proceed	Switch to the automatic mode by transforming all the remaining parts of the trace without giving control to the analyst.
Save	Save the obtained sequence diagram.

to a depth-first search. To allow the analyst navigation, we implemented a navigation bar with a time slider (Figure 10). The different navigation actions are detailed in Table I.

After each navigation action, the analyst regains control. After looking at the global view and the elements added in the SD for the last visited method call, he can modify the SD and possibly the combined trace. Modifications correspond to renaming objects and messages, deleting objects and messages, and merging SD fragments.

2) Renaming Objects and Messages: The objects participating in the sequence diagram are identified by their memory addresses. When the sequence diagram is used for documentation purpose, the name should help to better understand the behavior. This is particularly true when more than one object are created from the same class. In this context, the analyst could rename objects with meaningful names according to the use-case documentation. The analyst could also inspect the code corresponding to the method where the call occurs. Finally, to help the analyst find names, our environment also proposes a list of names derived by analysis of the corresponding code (variations of the class name and names of variables where the object is stored after its creation). For example, the participant initially created with label "1035198: Circle" could be renamed "circle: Circle".

Similarly, message names could be renamed by the analyst. In our example of the drawing tool, there is a method named *m_cc()* that modifies the color in the class Circle, The message corresponding to the call of *m_cc()* could be renamed to "*modify_object_color()*" for example to ease the comprehension of the sequence diagram.

When renaming messages, name equivalences are stored to ease future maintenance tasks.

3) Removing Objects and Messages: Execution traces contain many implementation details that should not appear in a SD. Examples of these details are call-to-library objects, graphical-interface objects and methods, temporary and working objects and methods, etc. The presence of these implementation details is one of the major limitations pointed out by state-of-the-art work (see [3] for example). Just by looking at class and message names and by inspecting code, the analyst could decide that an object or a message is not useful for the targeted abstract level.

By default when removing an object from the combined trace, all the incoming and outgoing messages are removed. Messages that result from these messages are also recursively removed. In many cases, this is impractical because an object to remove could be used as an intermediate object for implementation reasons. Thus we do not want to lose the rest of the interaction after removing it. To avoid these situations, we introduced three object-removing operations: (1) default recursive removal, (2) removal with message redirection (for example, when removing object b, if message $m1$ from object a to b triggers message $m2$ from b to object c, then we can choose to add $m1$ from a to c or add $m2$ from a to c), and (3) apply one of the two previous "removal" but to all objects in the trace having the same type as the removed object.

A message could be removed without deleting the sender or the receiver object. A message could be removed similarly to the three ways of removing objects: (1) default recursive removal (all messages triggered by the removed message are recursively deleted), (2) removal with redirection (for example, if we have chain $m1{>}m2{>}m3$, we could delete $m2$ and add the chain $m1{>}m3$), and (3) apply one of the two previous "removal" but to all messages of the same method in the trace.

4) Recommending Fragment Merges: During the combination of the execution traces (Section III), we also align sub-trees with common method calls and whose objects are not from the same type but having a common superclass (constructors are also aligned). In the combined trace, we did not merge the aligned sub-trees, but we inject a recommendation action. This action is triggered when all the aligned sub-trees are traversed and their elements mapped to the SD. The analyst could look to the global and SD views (and possibly the code) and accept/reject the recommendation. In case of acceptance, the associated message sequences are merged in the SD and the participant objects are replaced by an object that has the type of the common ancestor.

In our running example, parts of the three traces are dedicated to the creation and drawing of respectively circles, rectangles, and segments. Events 9 and 10 in Figure 2, and 8 and 9 in Figure 3, are the method calls corresponding to this common behavior. In both cases (plus the case of segments), the constructor is called followed by the method *draw*. The portion of the SD that corresponds to these parts is given in Figure 11(a). After the merge, the three objects c:Circle,

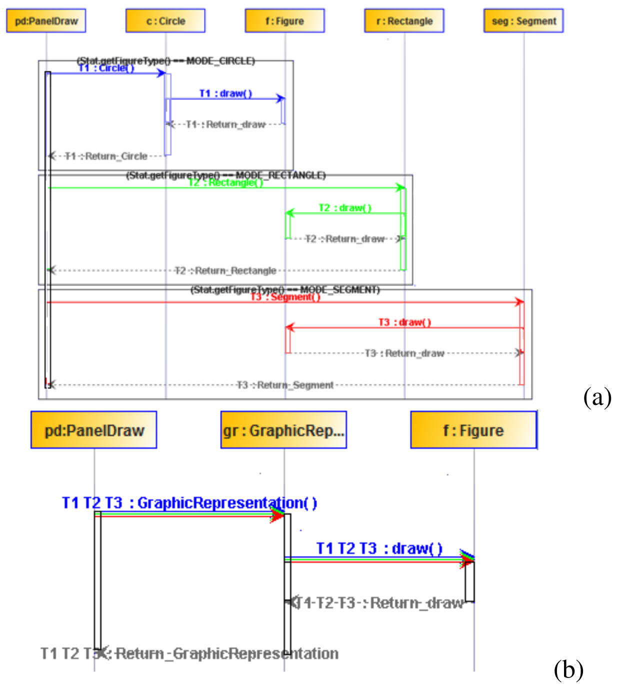


Figure 11. Merging SD fragments.

r:Rectangle, and seg:Segment are replaced by the object g:GraphicRepresentation because Circle, Rectangle, and Segment have GraphicRepresentation as common superclass. The constructors are replaced by the constructor of the new object. Finally, the message *draw()* with the three colored arrows between the new object and *f:Figure* replaces the three *draw()* messages (Figure 11(b)).

V. Case Study

A. Setting

The objective of this case study is to illustrate how interactive visualization complements automated reverse engineering of sequence diagrams. To perform our study, we selected an ATM simulation system[5]. This Java system, contains 24 classes. A similar implementation in C++ of the same system carried out by the same team was used to validate the automated approach presented in [2]. In addition to the fact that this system was previously used for the same problem, the ATM simulation system has the advantage that analysis and design documents are available. These documents include use cases that are necessary to determine the scenarios for which we reverse engineer the sequence diagrams. They also include the sequence diagrams corresponding to the different scenarios. This allows us to compare with our results objectively.

We selected three use-case scenarios: *Session*, *Deposit*, and *Withdraw*. For each scenario, we asked a subject to use our environment to extract the sequence diagram. During the generation process, the subject had access only to the source code and the Javadoc of the ATM simulation system. Our subject is a Ph.D. student in software engineering with a good experience in Java and UML.

To evaluate our approach, we compared three sequence diagrams for each scenario. The first diagram is the design diagram (DD) provided with the ATM system documentation. This constitutes our reference for comparison. The second diagram (ATD) is produced using an automated transformation according to the transformation rules of [2]. The third diagram (IVD) is produced by our subject using our interactive visualization environment. For each diagram, we counted the number of messages and the number of actors. For ATD and IVD, we counted the number of messages (respectively participants) that exist in the design diagram (DD).

In the remainder of this section, we first present the quantitative results for the three scenarios. Then, we detail the results of the scenario *Session*.

B. Quantitative Results

Table II summarizes the contents of design and generated diagrams. ATD contains around three times the number of messages of DD and between 2 and 6 additional participants. The recall for both messages and participants is, however, 100%. This is because all the interactions that occur in the execution are reported in the diagram.

IVD contains significantly fewer irrelevant messages (reduction between 30% and 60%). The recall is, however, less than the automated approach. Indeed, two messages are missing in the diagrams, corresponding to scenarios *Session* and *Withdraw*. In both cases, the missing messages were removed because our subject did not judge them important for the documentation of the scenario. All the participants of DD where found in IVD except for one in *Withdraw* and in *Session*. In both cases, our subject removed the object :Display corresponding to the participant :ConsumerConsole in the design diagram. This was done because it was considered as an interface object. Surprisingly, Display was not removed in *Deposit*. An irrelevant participant was added in *Withdraw* and in *Deposit*. It corresponds to object :Session. In DD, this participant was not included because there is already one participant representing transaction *Withdraw* (respectively *Deposit*).

These results should be contrasted by performance considerations. For all the scenarios, the process of extracting each SD took less than 20 minutes including consultation of code and Javadoc. This execution time is largely acceptable for reverse-engineering tasks and could be easily justified by the improvement in the precision of the generated diagrams.

C. Session Use-case Scenario

We detail scenario *Session* to illustrate the contribution of interactive visualization. This use-case scenario is used

[5]www.math-cs.gordon.edu/local/courses/cs211/ATMExample/

Table II
QUANTITATIVE RESULTS FOR DESIGN AND GENERATED DIAGRAMS.

Use-case scenarios			*Session*	*Deposit*	*Withdraw*
Design Diagram	Number of Messages		10	13	15
	Number of Participants		5	6	6
Automated Transformation Diagram	Number of Messages	All	39	39	41
		Relevant	10	13	15
	Number of Participants	All	11	8	8
		Relevant	5	6	6
Interactive Visualization Diagram	Number of Messages	All	16	27	29
		Relevant	8	13	13
	Number of Participants	All	4	7	6
		Relevant	4	6	5

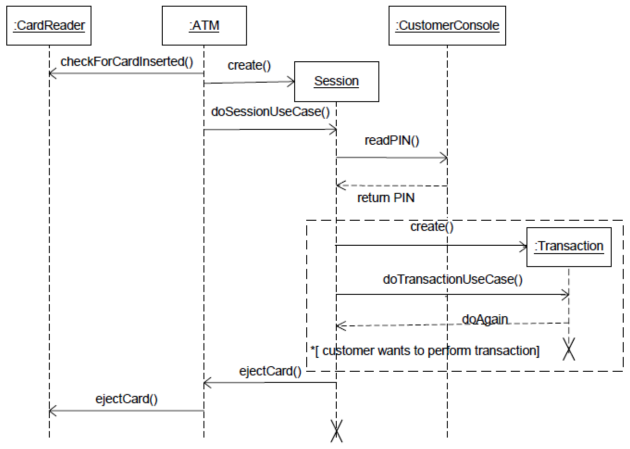


Figure 12. Sequence diagram for scenario *Session* in the documentation.

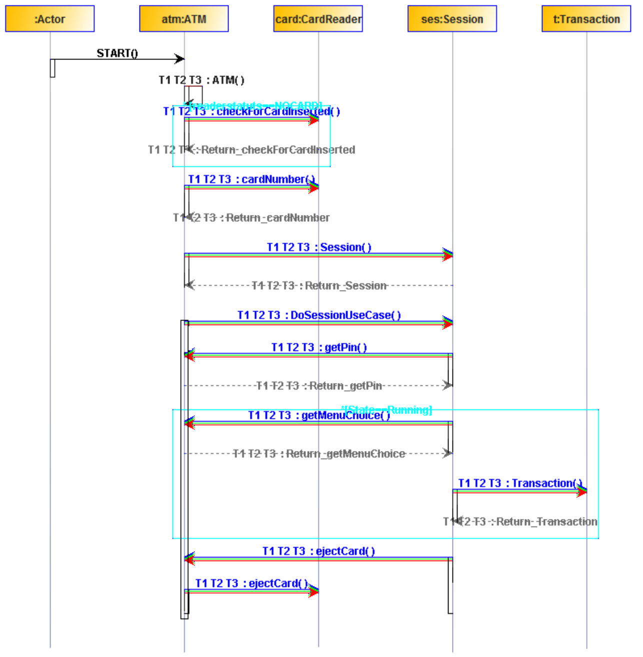

:Actor
atm:ATM
card:CardReader
ses:Session
t:Transaction
START()
T1 T2 T3 : ATM( )
[readerstatuts==NOCARD]
T1 T2 T3 : checkForCardInserted( )
T1 T2 T3 : Return_checkForCardInserted
T1 T2 T3 : cardNumber( )
T1 T2 T3 : Return_cardNumber
T1 T2 T3 : Session( )
T1 T2 T3 : Return_Session
T1 T2 T3 : DoSessionUseCase( )
T1 T2 T3 : getPin( )
T1 T2 T3 : Return_getPin
*[State==Running]
T1 T2 T3 : getMenuChoice( )
T1 T2 T3 : Return_getMenuChoice
T1 T2 T3 : Transaction( )
T1 T2 T3 : Return_Transaction
T1 T2 T3 : ejectCard( )
T1 T2 T3 : ejectCard( )

Figure 13. Sequence diagram for scenario *Session* generated by interactive visualization.

by all the scenarios corresponding to specific transactions that could be performed using the ATM system. To produce enough variations for this scenario, we executed the system with three different transactions: *Withdraw*, *Deposit*, and *Transfer*. This scenario is described as follows. First, the user is prompted to insert the card and enter its PIN number. The card and PIN number are then checked for validity. Once validated, the user starts performing a specific transaction.

The design diagram DD of this scenario is provided in Figure 12. The diagrams generated by interactive visualization IVD and automated transformation ATD are respectively given in Figures 13 and 14. The first observation we made is that many participant objects are added in ATD because of the initialization phase that does not exist in DD (top of ATD). Messages corresponding to this phase, such as *Bank()* and *CardReader()*, were removed by our subject in IVD, considering them as implementation details. This causes the removal of the corresponding participants (e.g. :Bank and :CardReader).

The second observation is that two participants considered as interface objects were removed by our subject. In the case of :Keyboard, this was a good decision (not present in DD). However, in the case of :Display, the designer estimated that this participant (called :CustomerConsole) is important for the documentation. In both cases, the deletion of participants resulted in the removal of corresponding messages such as *requestCard()* between :ATM and :Display, or *readPin()* between :ATM and :Keyboard.

Another interesting observation is related to the merge of execution fragments in IVD. Messages belonging to the specific transactions (two per transaction) are replaced by two messages. Specific participants are also replaced by :Transaction that corresponds to an object of the abstract class Transaction that is the common super-class of transaction classes. In ATD of Figure 14, there is one participant labeled with the four transactions. However, from our understanding, there is a specific participant for each execution and the merge between the four executions is not done automatically.

VI. RELATED WORK

In this section, we present an overview of existing work on reverse engineering of sequence diagrams. This work can be classified into two major categories, corresponding to the nature of the used information: static vs. dynamic.

For the first category, the sequence diagram is generated by analyzing source code. In the approach presented by Rountev et al. [1], sequence diagrams are derived from control flow graphs. In addition to the classical limits of static analysis to capture the behavior, this method produces sequence diagrams that are limited to a single function/method. In the same category, Tonella et al. [7] present a static

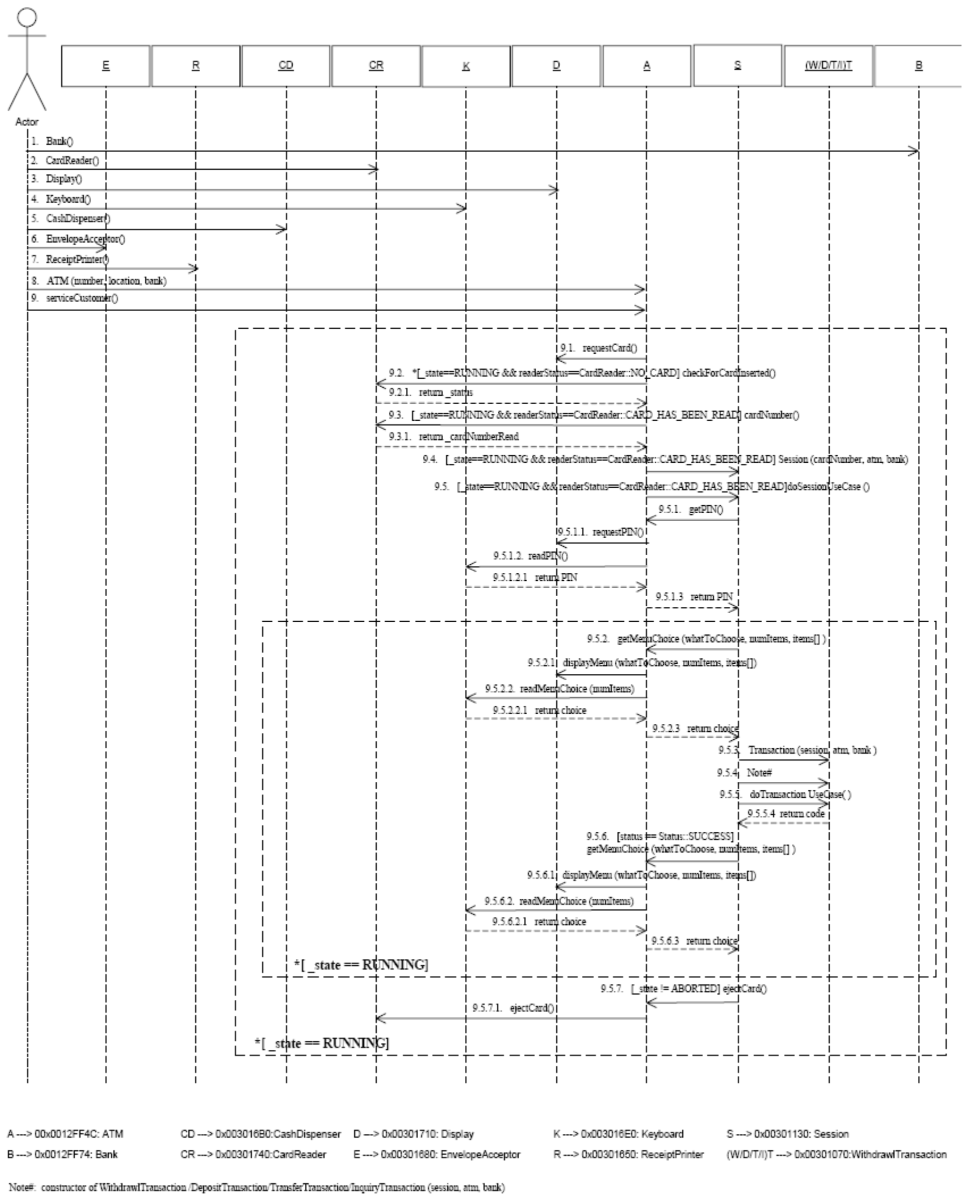

Actor
E
R
CD
CR
K
D
A
S
(W/D/T/I)T
B
1. Bank()
2. CardReader()
3. Display()
4. Keyboard()
5. CashDispenser()
6. EnvelopeAcceptor()
7. ReceiptPrinter()
8. ATM (number, location, bank)
9. serviceCustomer()
9.1. requestCard()
9.2. *[_state==RUNNING && readerStatus==CardReader::NO_CARD] checkForCardInserted()
9.2.1. return _status
9.3. [_state==RUNNING && readerStatus==CardReader::CARD_HAS_BEEN_READ] cardNumber()
9.3.1. return _cardNumberRead
9.4. [_state==RUNNING && readerStatus==CardReader::CARD_HAS_BEEN_READ] Session (cardNumber, atm, bank)
9.5. [_state==RUNNING && readerStatus==CardReader::CARD_HAS_BEEN_READ]doSessionUseCase ()
9.5.1. getPIN()
9.5.1.1. requestPIN()
9.5.1.2. readPIN()
9.5.1.2.1 return PIN
9.5.1.3 return PIN
9.5.2. getMenuChoice (whatToChoose, numItems, items[])
9.5.2.1 displayMenu (whatToChoose, numItems, items[])
9.5.2.2. readMenuChoice (numItems)
9.5.2.2.1 return choice
9.5.2.3 return choice
9.5.3. Transaction (session, atm, bank)
9.5.4. Note#
9.5.5. doTransaction UseCase( )
9.5.5.4 return code
9.5.6. [status == Status::SUCCESS] getMenuChoice (whatToChoose, numItems, items[])
9.5.6.1. displayMenu (whatToChoose, numItems, items[])
9.5.6.2. readMenuChoice (numItems)
9.5.6.2.1 return choice
9.5.6.3 return choice
*[_state == RUNNING]
9.5.7. [_state != ABORTED] ejectCard()
9.5.7.1. ejectCard()
*[_state == RUNNING]
A ---> 00x0012FF4C: ATM
B ---> 0x0012FF74: Bank
CD ---> 0x003016B0:CashDispenser
CR ---> 0x00301740:CardReader
D ---> 0x00301710: Display
E ---> 0x00301680: EnvelopeAcceptor
K ---> 0x003016E0: Keyboard
R ---> 0x00301650: ReceiptPrinter
S ---> 0x00301130: Session
(W/D/T/I)T ---> 0x00301070:WithdrawlTransaction
Note#: constructor of WithdrawlTransaction /DepositTransaction/TransferTransaction/InquiryTransaction (session, atm, bank)

Figure 14. Sequence diagram for scenario *Session* generated by automated transformation [2].

approach for reverse engineering of sequence diagrams and collaboration diagrams from C++. Here again, the analysis suffers from the limitation of static analysis, such as the difficulty to resolve polymorphic method calls.

To circumvent the above-mentioned limitations, many research teams use dynamic analysis to extract sequence diagrams by analyzing traces of executions. For example, Briand et al. [2], [3] use a model-driven approach to transform execution traces into sequence diagrams. In addition to generating participants and messages, this approach considers also alternative, optional, and loop structures. Similarly, Taniguchi et al. [4] propose an automated approach for extracting sequence diagrams from execution traces. In addition to use transformation rules, they add a simplification phase on the execution traces. Delamare et al. [5] propose also a generation with a simplification phase. Unlike the previous work where the simplification is done a priori on the execution traces, in their work, it is done on the generated sequence diagrams. Simplifications take into account object states to identify alternatives and loops.

Approaches based on dynamic analysis lack from contextual knowledge to discriminate between relevant information and implementation details. The resulting diagrams are generally very large and do not help much the understanding of the behavior. These diagrams could be interesting when reverse engineering is applied for purposes other than documentation. This is the case of the work of Ng et al. [13], where the extracted sequence diagrams are used to identify behavioral design patterns. Our approach allows analysts to use their contextual knowledge to filter unnecessary details. Interactive visualization combined with recommendation capabilities reduces the complexity of dealing with execution traces.

Behavior visualization is another domain that is related to our contribution. Many visualization metaphors have been proposed to understand program behavior (see for example [8] and [12]). The main difference with our work is the interactivity. Indeed, we use visualization not only to display the results of a process, but also to allow an analyst to contribute to this process.

VII. Conclusion and Future Work

Reverse engineering of sequence diagrams is an important task for the understanding of the behavior of object-oriented programs. This paper presented a semi-automated approach based on dynamic analysis and interactive visualization. Our approach follows three steps. First, execution traces are generated starting from use-case scenarios. These execution traces are then aligned to produce a combined trace. The third step is dedicated to the extraction of a sequence diagram. The extraction combines automated transformation with analyst actions through the interactive visualization environment. Our environment includes a recommendation module that suggests abstraction possibilities for behavior that is shared by different executions.

We evaluated our approach on a benchmark used in [2]. Our results showed that interactive visualization allows having more concise diagrams with more abstract behavior. The time dedicated to the interaction is acceptable considering the nature of the reverse-engineering task and the limitations of automated approaches.

Although, the first results are very promising, there is room for improvement. From the visualization point of view, the global view could significantly benefit from improved perception of the execution events. The recommendation module could also consider more subtle cases. In this context, we plan to explore incremental learning with feedback from the analyst. Our future work includes also the reverse engineering of other dynamic diagrams such as state diagrams. We believe that interactive visualization could help defining abstract states that are difficult to determine automatically.

Acknowledgment

The authors thank FQRNT and MDEIE for financial support, the GEODES graduate students for their participation, and the reviewers for their suggestions.

References

[1] A. Rountev, O. Volgin, M. Reddoch, "Static Control Flow Analysis for Reverse Engineering of UML Sequence Diagrams", Workshop on Program Analysis for Software Tools Engineering, 2005.

[2] L. Briand, Y. Labiche, Y. Miao, "Towards the Revese Engineering of UML Sequence Diagrams", WCRE 2003.

[3] L. Briand, Y. Labiche, Y. Miao, "Towards the Reverse Engineering of UML Sequence Diagrams for Distributed, Multithreaded Java Software", IEEE Trans. on Software Engineering, 2006.

[4] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, K. Inoue, "Extracting Sequence Diagram from Execution Trace of Java Program", International Workshop on Principles of Software Evolution (IWPSE2005), 2005, pp. 148-151.

[5] R. Delamare, B. Baudry, Y. Le Traon, "Reverse Engineering of UML 2.0 Sequence Diagrams from Execution Traces", Workshop on Object-Oriented Reengineering, 2006.

[6] W. De Pauw, D. Kimelman, J. Vlissides. "Modeling Object-oriented Program Execution", European Conference on Object- Oriented Programming, volume 821, pages 63-182, 1994.

[7] P. Tonella and A. Potrich. "Reverse Engineering of the Interaction Diagrams from C++ Code". Conf. Software Maintenance, pages 159-168, 2003.

[8] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, J. Yang, "Visualizing the Execution of Java Programs". Software Visualization, LNCS 2269, pages 151-162, 2002.

[9] T.F. Smith and M.S. Waterman, "Identification of Common Molecular Sub-sequences", Journal of Molecular Biology 147 (1981), 195-197.

[10] J. Walker, "A Node Positioning Algorithm for General Trees". Software Practice and Experience, 20 (7) : 685-705, 1990.

[11] G. Booch, J. Rumbaugh, I. Jacobson, "The Unified Modeling Language User Guide", Addison Wesley, 1998.

[12] B. Cornelissen, A. Zaidman, A. van Deursen, "A Controlled Experiment for Program Comprehension through Trace Visualization", IEEE Trans. on Software Engineering, 2010.

[13] J. Ka-Yee Ng, Y. Guéhéneuc, G. Antoniol, "Identification of Behavioural and Creational Design Motifs through Dynamic Analysis". J. Softw. Maint. Evol., 2009.

[14] A. Zaidman and S. Demeyer, "Automatic Identification of Key Classes in a Software System using Webmining Techniques", J. Softw. Maint. Evol., vol. 20, no. 6, pp. 387-417, 2008.