Recovering UML2 Sequence Diagrams from Execution Traces

EL Mahi BOUZIANE[1], Abdeslam JAKIMI[3]
Software Engineering and Information Systems Team
Faculty of Sciences and Technics, My Ismail University
Errachidia, Morocco

Chafik BAIDADA[2]
Laboratory of Information Technologies
ENSA, Chouaib Doukkali University
El Jadida, Morocco

***Abstract*—Reverse engineering is a proven and efficient technique for automatically generating UML2 models from object-oriented legacy systems with missing or obsolete documentation. To perform reverse engineering, two techniques are used: dynamic and static analysis. Dynamic analysis refers to collecting information when the system is running while static analysis corresponds to inspecting the source code. Dynamic analysis is preferred than static one in order to extract dynamic models that represents the behavior of a systems because of polymorphism and dynamic binding. In this paper, we present new different methodology that use Colored Petri Nets (CPNs) to recover UML2 Sequence Diagram (SD). First, it generates execution traces corresponding to the different scenarios representing the system behavior. Then, CPNs are used to model and analyze these execution traces to extract UML2 sequence diagram. Our case study illustrates the process of our approach and show that sequence diagram can be extracted with a good accuracy.**

***Keywords*—Execution traces; Reverse engineering; UML2; Sequence Diagram; Colored Petri Nets**

I. Introduction

Today object-oriented systems, are becoming increasingly larger and more complex. This increases the cost of their development and maintenance. According to [1], the cost of software maintenance represents 50% to 75% of the total cost. Despite the progress made in software engineering and development methods, several legacy systems still suffer from many problems such as unavailability of developers, obsolete development methods used to code the software, outdated documentation and non-compliance with the design when coding the software. In the software lifecycle, understanding its architecture and behavior is the main task in the maintenance phases. It is a tedious and time-consuming task that requires the mobilization of a large number of human resources. As mentioned in [2], up to 60% of maintenance time is spent on understanding the software. Therefore, it is important to develop techniques to obtain an abstract representation that facilitate the understanding of these systems.

A proven and effective technique to face this problem is reverse engineering of UML2 models. It can be defined as a process of analyzing the source code of systems and representing it in models with a higher level of abstraction. Reverse engineering is mostly used to extract high level abstraction models or semantics from the source code [3]. Reverse engineering is used to help understanding existing systems. The IEEE-1219 [4] standard considers reverse engineering as a technological solution to deal with legacy system. For the object-oriented software, the most used modeling language is UML (Unified Modeling Language) [5]. Dynamic models are as important as static models because they allow to understand the behavior of the system. One of the major UML dynamic model is SD. Indeed, it allows to represent complex interactions between its objects [6]. As described in [7], dynamic analysis allows to remove the ambiguity of message sending when inheritance, delegation, polymorphism, dynamic links, reflection are used intensively. For this, we will give more importance to this type of analysis.

This paper draws on our previous work [8, 9] to propose a new, more coherent and precise approach for reverse engineering the UML2 SD. This new approach allows to extract the conditions for combined fragment operator alt, opt and loop. For this purpose, improvements in the generation of execution traces and modeling with CPNs have been made. Indeed, the CPNs used have a smaller size and are more coherent. However, this approach does not currently apply to multi-threaded systems.

The remainder of this paper is organized as follows. Section II includes related works. Section III introduces a background in reverse engineering of UML2 SDs using CPNs. Section IV outlines the proposed approach. Section V presents a case study. Finally, Section VI concludes and points out some of our future works.

II. Related Work

Reverse engineering is defined as “the process of identifying and analysis of software’s system components, their interrelationships, and the representation of their entities at a higher level of abstraction” [10]. Reverse engineering aims to discover the technological principles of a system through the analysis of its structure and behavior.

In the literature, depending on the type of analysis used, there are two main categories in existing approaches: static and dynamic. Static analysis consists in performing the analysis of the source code or the binaries to generate UML dynamic diagrams. This is done without running the system. There are several approaches that perform reverse engineering through static analysis [11, 12, 13, and 14]. One of the main works based on static analysis is [14]. In this work, the authors present an algorithm that builds UML 2.0 sequence diagrams from Java code, using control flow graphs. They create an

algorithm that transform this graph on SD. This is done on three phases. First, the algorithm associates subgraphs with sequence diagram fragments operator alt, opt, break, and loop. After that, a series of transformations are applied to the obtained SD to in order to make it more understandable.

Dynamic analysis consists of running the program to obtain the necessary information in the form of an execution trace for the creation of sequence diagrams. These traces represent the values of the program variables, the state of the execution stack, the occurrences of objects created, the signatures of the methods called, the information about threads or any other execution information considered useful. As a result, objects under execution can be observed. There are several works that use dynamic analysis. In [15], an approach to extract SD from dynamic information of object-oriented programs is presented. In order to reduce repetitions in the execution trace, four rules are used to optimize the size of the execution traces by detecting similarity between sub-trees and replace merging them. The author in [16] propose an approach that allows extracting UML High Level Sequence Diagrams (HLSD) from java code by constructing control flow graphs. They proposed a method for switching between the general control flow graph (FCG) and UML sequence diagrams. The combination process is done by analyzing the different states of the system. In [17], it is proposed an approach based on dynamic analysis using Labeled Transition System (LTS). These LTSs are used for modeling execution traces in order to facilitate there analyzing. For each trace a corresponding LTS is generated. After that come the step to merge these LTSs in order to have one LTS modeling the behavior of the system. Finally, an HLSD is generated from the obtained LTS using regular expressions.

The approaches listed before were able to extract SDs that represent the system behavior. However, the diagrams obtained are incomplete and suffer from several problems. These problems include information filtering problems. As listed in the catalog of abstractions and filtering in the context of reverse engineering of sequence diagrams [18]. In addition, these approaches fail to extract the conditions in the combined fragment operators like loop, opt and alt.

III. Background of the Approach

In this section, we first explain what a sequence diagram in UML 2.x is. Second, we give some definitions regarding execution traces and how they are obtained. Finally, we introduce CPN and how we used it to represent an HLSD.

A. UML2 Sequence Diagrams

The SD is a form of behavioral diagram that allows to specify in a chronological way the interactions that exist between a group of objects from the temporal point of view. It has been significantly changed in UML 2.0 [5]. Indeed, the sequence diagram in UML2 is considered as partially ordered collections of events, which introduces new concepts such as combined fragment, parallelism and a synchronism and allows the definition of more complex behaviors.

An HLSD is obtained by combining Basic Sequence Diagram (BSD) using interaction operators. The most commonly used combined fragment operators in the UML2 sequence diagram are *seq* to express sequence, *opt* for optional, *alt* for alternatives and *loop* for iterative actions.

B. Execution Traces

Dynamic analysis, starts by generating traces. These traces are then analyzed to extract a HLSD. In our approach, for each scenario a trace execution is generated. In what follows, we introduce a set of definitions that are necessary to understand the approach.

Def. 1: A trace line is a method invocation or a control structure.

Def. 2: A method invocation is a triplet T1=<Sender, Message, Receiver> where:

Sender is the caller object, expressed in the form package:class:object.

Message is the invoked method of the receiver object, expressed in the form methodName (par1, par2, …).

Receiver is the called object, expressed in the form package:class:object.

Def. 3: A control structure is a triplet T2 = <controlType, status, condition> where:

ControlType has one of the following values: IF, ELSE, SWITCH, CASE or DEFAULT.

Status expresses the start or the end of the control structure.

Condition (optional) is the condition expression associated with IF, CASE, FOR, or WHILE.

Def. 4: (Equivalence between method invocations): The method invocations l1 = <s1, m1, r1> and l2 = <s2, m2, r2> are equivalent if and only if:

- The objects s1 and s2 (respectively, r1 and r2) are equivalent if they are instances of the same class.
- The messages m1 and m2 concern the same method and have the same arguments.

Def. 5: An execution trace is a set of trace lines.

An example of execution traces in the format described before are shown in Table 1. For each Scenario correspond a trace (ex: Trace1 refers to Scenario1). The trace Trace1 is composed of lines from L0 to L5. Lines L6 to L10 belongs to Trace2. Pack1 represent the packages to which classes A and B belong. m1() to m5() refers to the methods invocation (messages) of objects a, b, c, and d.

C. CPN

Petri nets is a formal modeling language used to represent the dynamic behavior of different systems (computer, industrial, telecommunications ...) [19]. It was first introduced in 1962 by the German mathematician and computer scientist Carl Adam. CPN is an extension of Petri nets. CPN is an extension of petri nets. This extension considerably reduces the size of the network when extending the modeling with Net Petri. It allows the distinction between places by attaching a color to them.

TABLE I. AN EXAMPLE OF TRACES

Trace1 L6. Pack1:B:b|m3()|Pack1:A:a L7. WHILE |BEGIN |condition1 L8. Pack1:B:b|m4()|Pack1:A:a L9. Pack1:B:b|m5()|Pack1:A:a L10. WHILE|END L6. Pack1:B:b|m3()|Pack1:A:a	**Scenario1**
Trace2 *L0. IF* | BEGIN | condition2 L1. Pack1:A:a |m1()|Pack1:B:b L2. ELSE | BEGIN L3. Pack2:C:c |m2()|Pack2:D:d L4. ELSE| END L5. IF | END L1. Pack1:A:a |m1()|Pack1:B:b	**Scenario2**

A Petri Net block is a subnet of the Petri Net with one initial place and one final place. Those places refer respectively to the precondition and the post-condition of the subnet. In [20], CPN is used to integrate scenarios represented in the form of SDs. They use four combined fragment operators (conditional, sequential, iterative and concurrent) to combine scenarios. CPNs are suitable for our approach as they can be transformed easily into an HLSD (see Fig. 1).

Transitions represent BSD or the operator such as seq, opt or control type as defined in Def 3. Places can represent a state of the system or the beginning or the end of the operator alt and loop. Colors are used to distinguish between traces.

Fig. 1 shows how a CPN can be transformed easily into a HLSD and vice-versa. Pi and Pf represent respectively the initial and the final place of the Petri Net block. The first transition a | m1() |b corresponds to the first BSD. In this BSD the object a of class A call to the object b of the class B with the message m1(). The place LOOP | BEGIN represents the state before the start of the operator loop that leads to two transitions. The transition IF | BEGIN| C1 allows entering in the loop statement. This is done when the condition C1 of loop is equal to the value true.

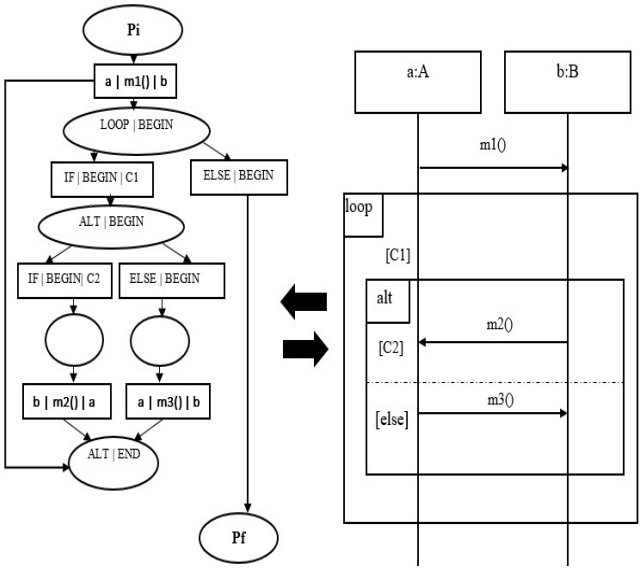

Fig. 1. A HLSD Mapped onto CPN with Operators Loop and Alt.

After that, comes the place ALT | BEGIN which represents the state that also leads to two transitions. The first transition labeled IF | BEGIN| C2 refers to the case when the condition of *alt* is satisfied and leads to the transition b | m2() | a. This transition describes that the message m2() is sent by the object a of the class A to the object b of class B. The transition ELSE | BEGIN| C2 represents the second transition of *alt* and consequently occurs when its condition is not verified. The transition a| m3() | b refers to the BSD which describes that the object a calls the object b using the message m3(). The second transition of *loop* is ELSE | BEGIN refers to its end and thus occurs when its condition is not verified.

In this section, we have presented and explained the most important concepts about SD, trace and CPN. This is the background on which our approach is based.

IV. PROPOSED APPROACH

Our objective is to extract SD from execution traces for an object oriented system using CPNs.

In this section, we present our approach for reverse engineering of the HLSD. As illustrated in Fig. 2, the approach is divided in four main steps. First, the step trace collection. Second, the trace filtering step. Third the step of trace merging. Finally, the step of HLSD extraction. In the next subsections, each step is described in details.

A. Traces Collection

In order to generate an accurate HLSD, our approach use dynamic analysis technic. In [6], it's described that dynamic analysis is more efficient than static one in the context of the reverse engineering of UML dynamic models such as SDs. This analysis is based on analyzing traces execution. These traces can be generated using several technics [2]. These technics includes the instrumentation of the source code, bytes code or the use of a customized debugger. In our approach we use the byte code instrumentation.

We choose to use AspectJ [21] as trace collection tools. This tool concerns java software systems. It allows to report all information created during the execution of the program. This includes methods invocations, occurrence of objects, sending and receiving messages between objects, loops and conditions.

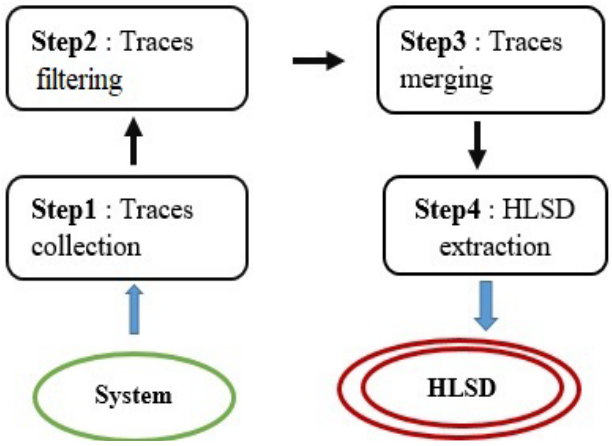

Fig. 2. Overview of our Approach.

The behavior of the system is highly dependent on the input data entered by the user. Therefore, it is necessary to identify the majority of input variable values in order to specify all system behavior. This can be done with the help of a system functional expert. This can be done with the help of a system functional expert. After running the system with different input data values, execution traces are generated, each corresponding to a given scenario. Since execution trace formats differ from one tool to another, we have developed a tool that will allow to standardize the format of execution traces. This format is defined in the definitions 1, 2, and 3. The objective of this tool is to format the trace execution to facilitate the processing of merging traces.

B. Trace Filtering

The generated execution traces contain a lot of information about all classes composing the system. For example, these classes can be divided into three types: data access classes, business classes and presentation classes. The business classes are the classes that describe the behavior of the business logic of the system. Our objective in this step is to concentrate on traces lines that describe this behavior and ignore other traces lines. This is the objective of the trace filtering step. We have developed an algorithm that allows us to delete execution traces which belong to data access or presentation classes.

C. Traces Merging

As mentioned before, one execution of the system doesn't allow an accurate description of all the system behavior. Therefore, the system must be run several times and thus generate different traces. The challenge is to be able to merge these different traces to identify the behavior of the system as a whole. In [22], several well-defined merging techniques were listed.

For merging execution traces, we choose to use CPNs. The process is done in two successive sub-steps: first CPN initialization, then CPN merging.

1) CPN Initialization: In this sub-step, a basic CPN is generated for each execution trace. All the trace lines are transformed into transitions in CPNs except those which refers to the start or the end of iterative control structure like LOOP | START and LOOP | END. These line traces are transformed into places. This reduces the size of CPN and makes it more consistent. A same color is assigned to all places that belong to the same execution trace. We use these colors to differentiate between scenarios. Each color refers to specific scenario. This allows subdividing an HLSD into less complex HLSDs to facilitate understanding the behavior of the system.

2) CPN Merging: In this sub-step, all the CPNs corresponding to execution traces are merged to obtain a single CPN. The algorithm kBehavior [23] is used for this reason. This algorithm has points in common with the known Ktail algorithm. [24]. These algorithms are used to construct finite-state automaton (FSA) that abstracts execution traces. The algorithms iteratively merge the equivalent states in order to generalize the resulting FSA. kBehavior can reuse already learned path to adapt it in the FSA with newly generated traces. This is not the case for Ktail. In our approach, we have developed an algorithm called adapted kBehavior which is totally inspired by kBehavior. It's a new version adapted to deal with CPNs. Adapted kBehavior does not need to pre-process the new traces it receives. However, it needs to explore the already generated CPN when it tries to learn again. When a new sequence of transition and places needs to be added to the CPN, it must be ensured that this sequence is not already present in the CPN. Since the generated CPN is generally not non-deterministic, the path of the CPN is quite inexpensive and the additional cost generated by this method remains reasonable.

To make the CPN more coherent, a final transformation is carried out. This transformation concerns the processing of an iterative behavior. This processing includes adding two test transitions after the place LOOP | BEGIN | CONDITION. The first transition labeled IF | BEGIN | CONDITION is executed when the condition of Loop is satisfied. The second transition labeled LOOP | END is executed in the other case. This transition leads to the place labeled LOOP | END and consequently indicating the end of Loop. The output place of the last transition inside *loop* does not refer anymore its end but to its beginning. The labeling of this place is changed by removing the indication of its condition in order to avoid redundancy as illustrated in Fig. 3.

D. HLSD Extraction

In this step, we can easily build an HLSD by mapping the resulting CPN using the following transformation rules.

- **Rule 1:** all names of objects in the CPN are transformed into lifelines in SD.
- **Rule 2:** a transition T1 with the method invocation 0:a:B | m1 ()| b:B is transformed into a BSD where object a:A sends message m1() to object b:B
- **Rule 3:** A Place P1 that contains the operator ALT | BEGIN **or** OPT |BEGIN **or** LOOP **|** BEGIN refers respectively to BSD with the operators alt, opt and loop.
- **Rule 4:** the CPN paths coming after the place ALT | BEGIN and ending on the transition ALT | END are transformed into combined fragments with the operators ALT.
- **Rule 5:** the CPN paths coming after the place OPT | BEGIN and ending on the transition OPT | END are transformed into combined fragments with the operators OPT.
- **Rule 6:** The cyclic CPN paths coming after the transition IF | BEGIN | CONDITION which comes after the place LOOP | BEGIN is transformed into combined fragments with the operator loop.
- **Rule 7:** The CPN paths coming after the transition ELSE | BEGIN | CONDITION which comes after the place LOOP | BEGIN is transformed into BSD after the fragment corresponding to the operator loop.

The rules listed below can be combined to map a CPN to an HLSD representing system behavior (Fig. 2).

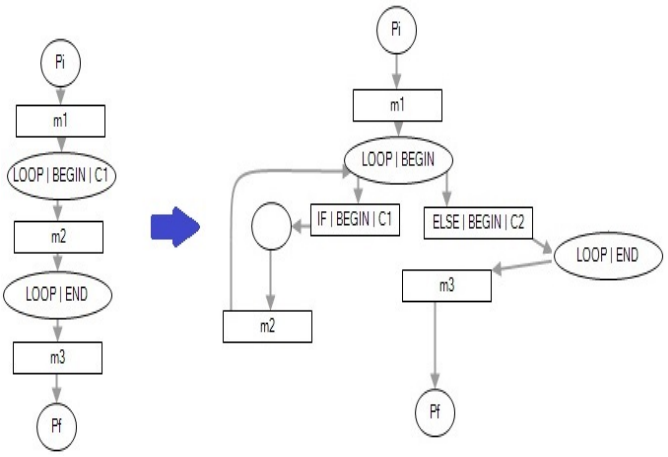

Fig. 3. CPN Corresponding to Scenario1.

V. Case Study

To test and illustrate the different steps of our approach, we have developed an application called Sales. It allows vendors to create sales of several articles. It gives the possibility to print an invoice, delivery or a payslip. All these operations are saved in a database. The application developed in Java provides different types of behavior (iterative, optional, sequential and alternative) which are the objective of our case study. The application has a layered architecture with two layers: business logic and data access layer and therefore is structured in two packages BLL (for business logic) and DAL (for data access layer). The BLL package contains six business classes (see Listing 1): Vendor, Sale, Calculation, Invoice, Payslip, and Delivery. The DAL package is composed of the following classes (see Listing 2): VendorDAL, SaleDAL, InvoiceDAL, PayslipDAL, and DeliveryDAL.

To create sales, the vendor makes an order to start a new sale. The vendor can add articles repetitively and calculate the total amount of the sales (repetitive behavior). When the vendor completes the sale, he chooses to print a delivery slip or an invoice in order to be signed (alternative behavior). Finally, if the customer wants it, a pay slip must be printed (optional behavior).

Listing1 and listing2 shows the source code of some classes of the application. Listing1 refers to the BLL package and Listing2 correspond to the DAL package.

Listing 1

```
1   package BLL;
2   import DAL.*;
3   class Vendor {
4    public  Vendor(){
5    }
6     public static void signInvoice(){
7     System.out.println("Invoice signed");
8   aVendorDAL.saveUpdate(this);
9    }
10   public static void signDelivery(){
11    System.out.println("Delivery signed");
12  aVendorDAL.saveUpdate(this);
13   }
14   public static void signPayslip(){
15    System.out.println("Payslip signed");
16  aVendorDAL.saveUpdate(this);
17   }
18  }
19  class Sale {
20   public void newSales(Vendor V, int nbr_article, boolean
21  isInvoice,   Boolean isPayslip){
22    float oldValue=0;
23    System.out.println("New sale created");
24    for(int i=1;i<=nbr_article;i++){
25     float newValue=addArticle();
26     Calculation calcul= new Calculation();
27     oldValue=calcul.calculAmount(newValue, oldValue);
28    }
29    if(isInvoice){
30    Invoice invoice= new Invoice();
31    invoice.waitMessage();
32    invoice.start();
33   }
34   else {
35   Delivery delivery= new Delivery();
36   delivery.getDelivery();
37   }
38   if(isPayslip){
39    Payslip payslip=new Payslip();
40    payslip.getPayslip();
41   }
42   }
43   public  float addArticle(){
44    SalesDAL saleDAL =new SalesDAL();
45    saleDAL.saveUpdate(this);
46    System.out.println("New article added");
47    return 1000;
48   }
49  }
50  class Invoice extends Thread {
51    public void preparingInvoice(){
52     System.out.println("Preparing Invoice ");
53    }
54     public void waitMessage(){
55      System.out.println("waiting for Invoice ");
56    }
57    public void getInvoice(){
58    System.out.println("Invoice printed");
59    Vendor.signInvoice();InvoiceDAL Inv1= new
60  InvoiceDAL();
61    Inv1.saveUpdate(this);
62    }
63    @Override
64      public void run() {
65       try {
66            preparingInvoice();
67            Thread.sleep(1000);
69            getInvoice();
70       }
71       catch (InterruptedException ex) { }
72    }
73   }
74  class Payslip {
75   public void getPayslip(){
76    System.out.println("Payslip printed");
77    Vendor.signPayslip();
78    PayslipDAL Pay1 = new PayslipDAL();
79    Pay1.saveUpdate(this);
80    }
90   }
100
101
```

Listing 2

```
1  package DAL;
2  import BLL.*;
3
4  class DeliveryDAL  {
5  public  void saveUpdate (Delivery d){
6     System.out.println
7  ("Delivery updated in database  ");
8        /*
9  ...sending sql query to database using jdbc */
10  }
11 }
12
13 class SaleDAL {
14 // if new one a sale is created and updated
15 // else the existing sale is updated
16  public  void saveUpdate (Sales s){
17    System.out.println
18 ("sales updated in database  ");
19       /*
20  ... sending sql query to database using jdbc */
21  }
22
23 public class PayslipDAL {
24  public  void saveUpdate (Payslip s){
25  System.out.println
26 ("Payslip updated in database  ");
27       /*
28 ... sending sql query to database using jdbc */
29  }
30 }
31
32 class VendorDAL {
33  public   void saveUpdate (Vendor s){
34    System.out.println("Vendor updated in database  ");
35     /*
36   ... sending sql query to database using jdb */
37  }
38 }
39
```

A. Trace Collection

At this stage, the generated execution traces that represent the behavior of the systems are organized in text files according to the format proposed by our approach. To do that, we use a java program that we have developed. It takes as input the traces generated after instrumentation with AspectJ. Then, it adapts them according to the adequate format as it shown in Table II, Table III and Table IV.

B. Trace Filtering

As input for this step, we have formatted traces that each correspond to a given scenario. These traces include calls of all objects that belong to packages BLL and DAL. In this case study, we consider that the main behavior of our application is illustrated in the business logic layer. So, we will ignore all trace lines that refers to the data access layer (trace lines with red color). For that, the algorithm will delete all lines traces that includes the package: DAL. Therefore, the line traces in red in Table II will be deleted. The final traces will contain only traces that include the BLL package.

C. Trace Merging

This step consist in generating for every filtered trace a corresponding CPN (Fig. 4, 5 and 6). These CPNs include as transitions the events generated by the system like the invocation of methods and performing tests. The places of CPN contains only the stars and the end of structure controls: LOOP | BEGIN |, LOOP | END, ALT | BEGIN |, ALT | END. To do this, we have developed an algorithm that transforms every trace into a CPN.

First, our algorithm creates the initial place that represents the start of the CPN. Then, if it finds a method invocation, a correspondent transition is created and attached to the CPN. If a loop control structure is found, it creates places to indicate the start and the end of the iteration and create transitions corresponding to methods invocations between them. When, an alternative structure control is found, the algorithm checks if there is trace line with IF and ELSE. Then, it creates two places labeled ALT | BEGIN | CONDITION and ALT | END that indicate the start and the end of the if else test. After that, it checks if there is a method invocation after the trace line IF | BEGIN, a transition with the same label is created then another transition with the method invocation. Otherwise, a transition with the label ELSE | BEGIN is created. In the case when only IF is found without ELSE the algorithm creates two places labeled OPT | BEGIN | CONDITION and OPT | END.

TABLE II. EXECUTION TRACE CORRESPONDING TO SCENARIO 1

Trace1 (nbr_article = 3, isInvoice = false, isPayslip = false): L0. 1:BLL:Vendor:vendor | Vendor () | DAL:VendorDAL:vendorDAL L1. 1:BLL:Vendor:vendor | newSales (nbr_article, isInvoice, isPayslip) | BLL:Sale:sale L2. FOR | BEGIN | i<=nbr_article L3. 1:BLL:Sale:sale | addArticle () | DAL:SaleDAL:saleDAL L4. 1:BLL:Sale:sale | addArticle () | BLL:Sale: sale L5. 1:BLL:Sale:sale | calculAmount(newValue, oldValue) | BLL:Calcul:calcul L3. 1:BLL:Sale:sale | addArticle () | DAL:SaleDAL:saleDAL L4. 1:BLL:Sale:sale | addArticle () | BLL:Sale:sale L5. 1:BLL:Sale:sale | calculAmount(newValue, oldValue) | BLL:Calcul:calcul L3. 1:BLL:Sale:sale | addArticle () | DAL:SaleDAL:saleDAL L4. 1:BLL:Sale:sale | addArticle () | BLL:Sale:sale L5. 1:BLL:Sale:sale | calculAmount(newValue, oldValue) | BLL:Calcul:calcul L6. FOR | END L7. IF | BEGIN | isInvoice L8. ELSE | BEGIN L9. 1:BLL:Sale:sale | getDelivery () | BLL:Delivery:delivery L10. 1:BLL:Delivery:delivery | signDelivery () | BLL:Vendor:vendor L11. 1:BLL:Delivery:delivery | getDelivery () | DAL: DeliveryDAL:deliveryDAL L12. IF | END L13 IF | BEGIN | *isPayslip* L14. IF | END	**Scenario1**

TABLE III. EXECUTION TRACE CORRESPONDING TO SCENARIO 2

Trace2 (nbr_article = 1, isInvoice = true, isPayslip = true): L0. 1:BLL:Vendor:vendor | Vendor () | DAL:VendorDAL:vendorDAL L1. 1:BLL:Vendor:vendor | newSales (nbr_article, isInvoice, isPayslip) | BLL:Sale:sale L2. FOR | BEGIN | i<=nbr_article L3. 1:BLL:Sale:sale | addArticle () | DAL:SaleDAL:saleDAL L4. 1:BLL:Sale:sale | addArticle () | BLL:Sale:sale L5. 1:BLL:Sale:sale | calculAmount(newValue, oldValue) | BLL:Calcul:calcul L6. FOR | END L7. IF | BEGIN | isInvoice L15. 1:BLL:Sale:sale | getInvoice() | BLL:Invoice:invoice L16. 1:BLL:Sale:sale | waitMessage () | BLL:Vendor:vendor L17 PAR | BEGIN L18. 10:BLL:Invoice:invoice | preparingInvoice () | BLL:Invoice:invoice L19. 10:BLL:Invoice:invoice | signInvoice () | BLL:Vendor:vendor L20. 10: BLL:Invoice:invoice | addArticle () | DAL:InvoiceDAL:invoiceDAL L21. PAR | END L8. ELSE | BEGIN L12. IF | END L13 IF | BEGIN | *isPayslip* L22. 1:BLL:Sale:sale | getPayslip () | BLL: Payslip: paySlip L23. 1:BLL: Payslip: paySlip | signPayslip () | BLL:Vendor:vendor L24. 1:BLL: Payslip: paySlip | getPayslip () | DAL: PayslipDAL: paySlipDAL L14. IF | END	**Scenario 2**

TABLE IV. EXECUTION TRACE CORRESPONDING TO SCENARIO 3

Trace3 (nbr_article = 4, isInvoice = false, isPayslip = true): L0. 1:BLL:Vendor:vendor | Vendor () | DAL:VendorDAL:vendorDAL L1. 1:BLL:Vendor:vendor | newSales (nbr_article, isInvoice, isPayslip) | BLL:Sale:sale L2. FOR | BEGIN | i<=nbr_article L3. 1:BLL:Sale:sale | addArticle () | DAL:SaleDAL:saleDAL L4. 1:BLL:Sale:sale | addArticle () | BLL:Sale:sale L5. 1:BLL:Sale:sale | calculAmount(newValue, oldValue) | BLL:Calcul:calcul L3. 1:BLL:Sale:sale | addArticle () | DAL:SaleDAL:saleDAL L4. 1:BLL:Sale:sale | addArticle () | v:Sale:sale L5. 1:BLL:Sale:sale | calculAmount(newValue, oldValue) | BLL:Calcul:calcul L3. 1:BLL:Sale:sale | addArticle () | DAL:SaleDAL:saleDAL L4. 1:BLL:Sale:sale | addArticle () | BLL :Sale:sale L5. 1:BLL:Sale:sale | calculAmount(newValue, oldValue) | BLL:Calcul:calcul L3. 1:BLL:Sale:sale | addArticle () | DAL:SaleDAL:saleDAL L4. 1:BLL:Sale:sale | addArticle () | BLL :Sale:sale L5. 1:BLL:Sale:sale | calculAmount(newValue, oldValue) | BLL:Calcul:calcul L6. FOR | END L7. IF | BEGIN | isInvoice L8. ELSE | START L9. 1:BLL:Sale:sale | getDelivery () | BLL:Delivery:delivery L10. 1:BLL:Delivery:delivery | signDelivery () | BLL:Vendor:vendor L11. 1:BLL:Delivery:delivery | getDelivery () | DAL: DeliveryDAL:deliveryDAL L12. IF | END L13 IF | BEGIN | *isPayslip* L22. 1:BLL:Sale:sale | getPayslip () | BLL: Payslip: paySlip L23. 1:BLL: Payslip: paySlip | signPayslip () | BLL:Vendor:vendor L24. 1:BLL: Payslip: paySlip | getPayslip () | DAL: PayslipDAL: paySlipDAL L14. IF | END	**Scenario 3**

The Places Pi and Pf are added to the CPN to indicate respectively the initial place and the final trace. To simplify the CPNs, all repeated method invocation between places "LOOP | BEGIN" and "LOOP | END" is deleted. All places representing trace lines are colored with the same color. These colors are used to diferentiat between the different scenarios.

After merging CPNs that refers to scenario1, scenario2 and scenario3, using the adapted Kbehavior, a new CPN is generated (Fig. 7). This CPN includes diferent paths with differents colors. Scenario1 has a yellow color, scenario2 has the green color while scenario3 has the red color.

The condition C1 refers to when the variable i is less than nbr_article while the condition C2 corresponds to if the variable isinvoice is true. Now, we apply our last transformation on loop places to make the obtained CPN more coherent (Fig. 8).

D. HLSD Extraction

The objective of this step is to extract the HLSD that represent the system behavior. For that, we use the transformation rules described in Section 4.4 to transform the final CPN into HLSD (Fig. 9).

The approach, as shown in following figure, is able to extract HLSD with the main UML2 fragment operators (seq,opt, alt and loop). Unlike other approaches using dynamic analysis, our approach succeeds in extracting the conditions corresponding to the combined fragment operators loop, opt and alt.

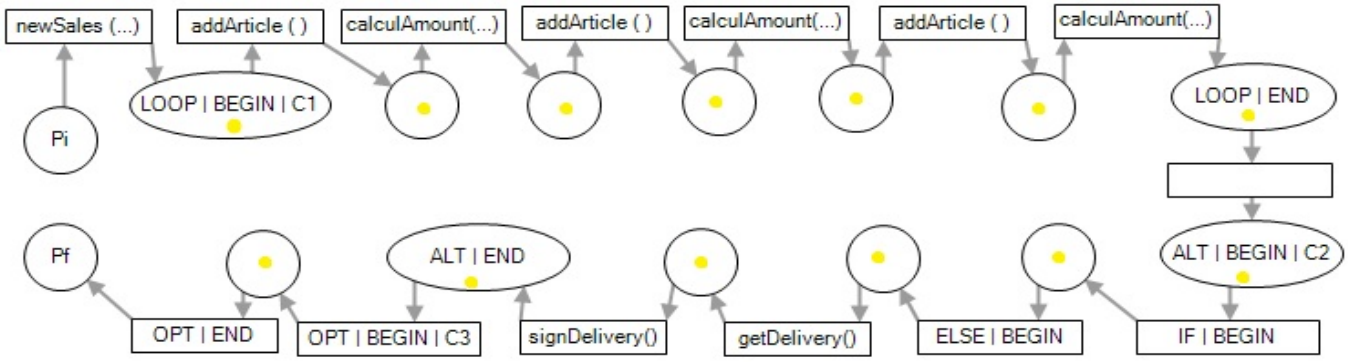

Fig. 4. CPN Corresponding to Scenario1.

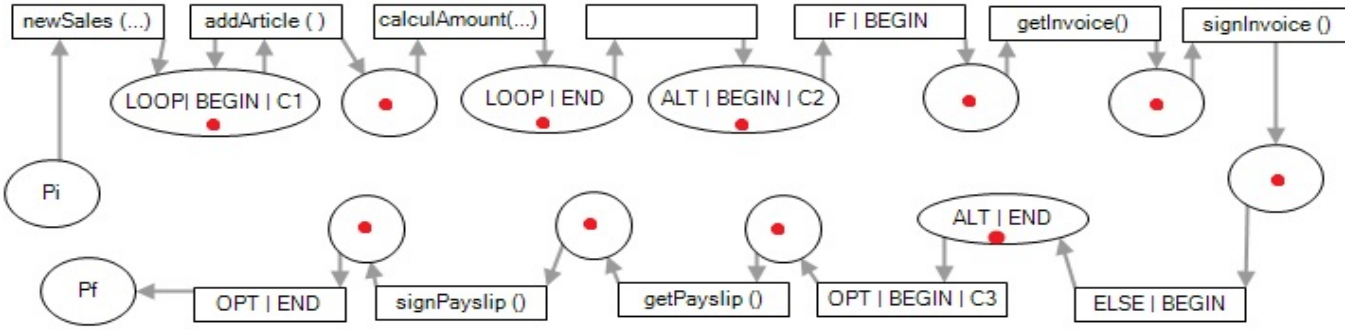

Fig. 5. CPN Corresponding to Scenario2.

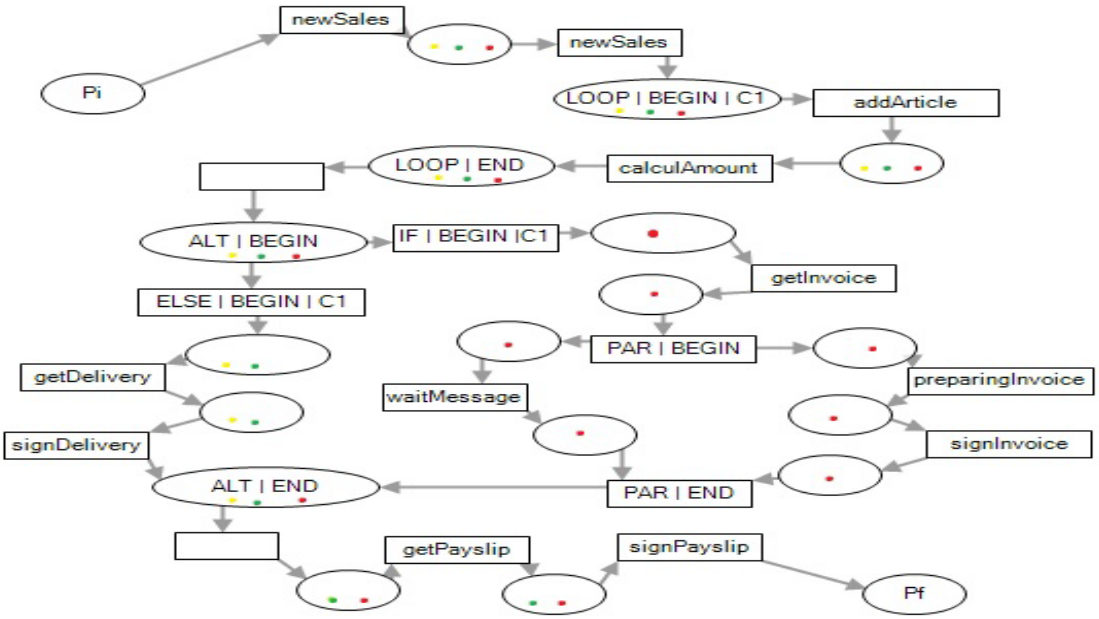

Fig. 6. CPN Corresponding to Scenario3.

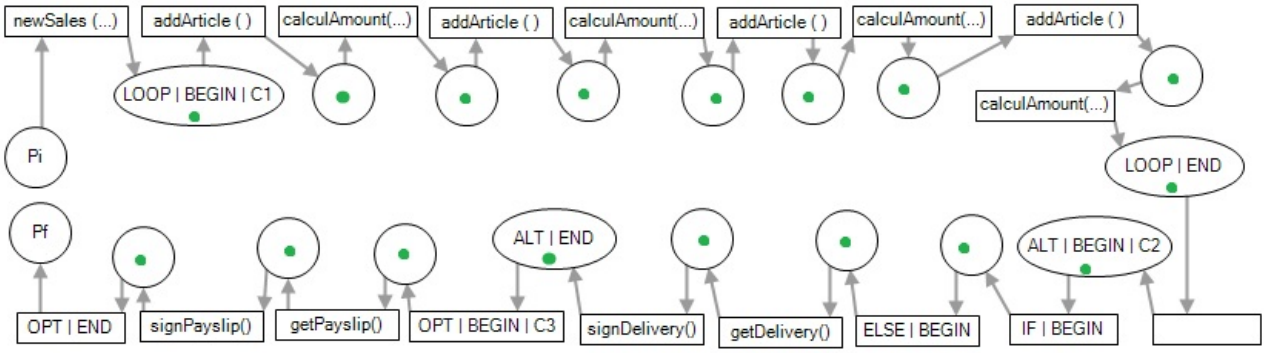

Fig. 7. The Merged CPN.

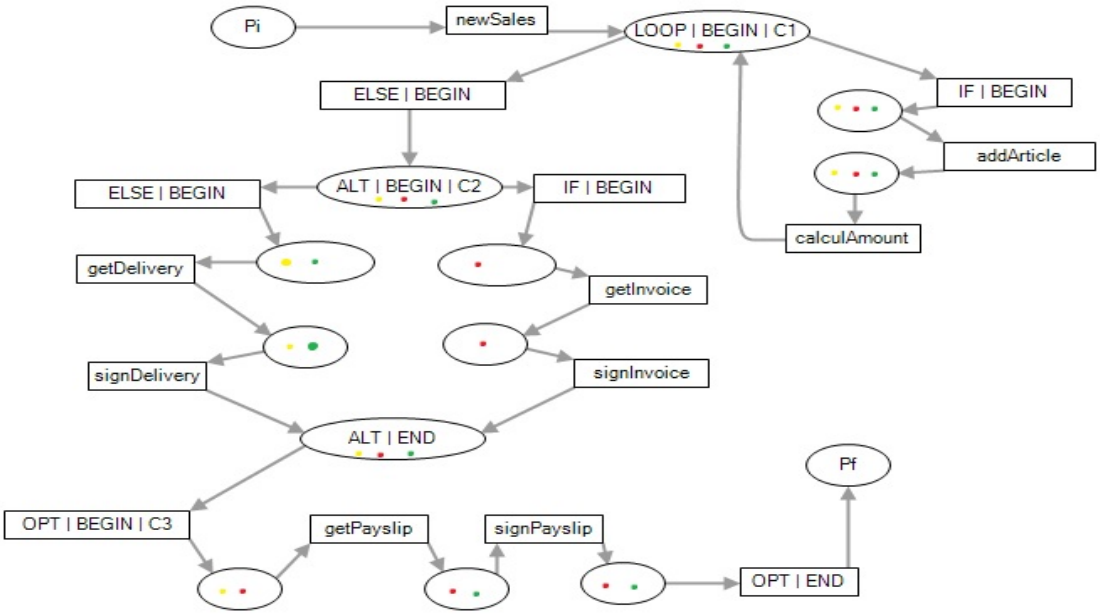

Fig. 8. The Finale CPN.

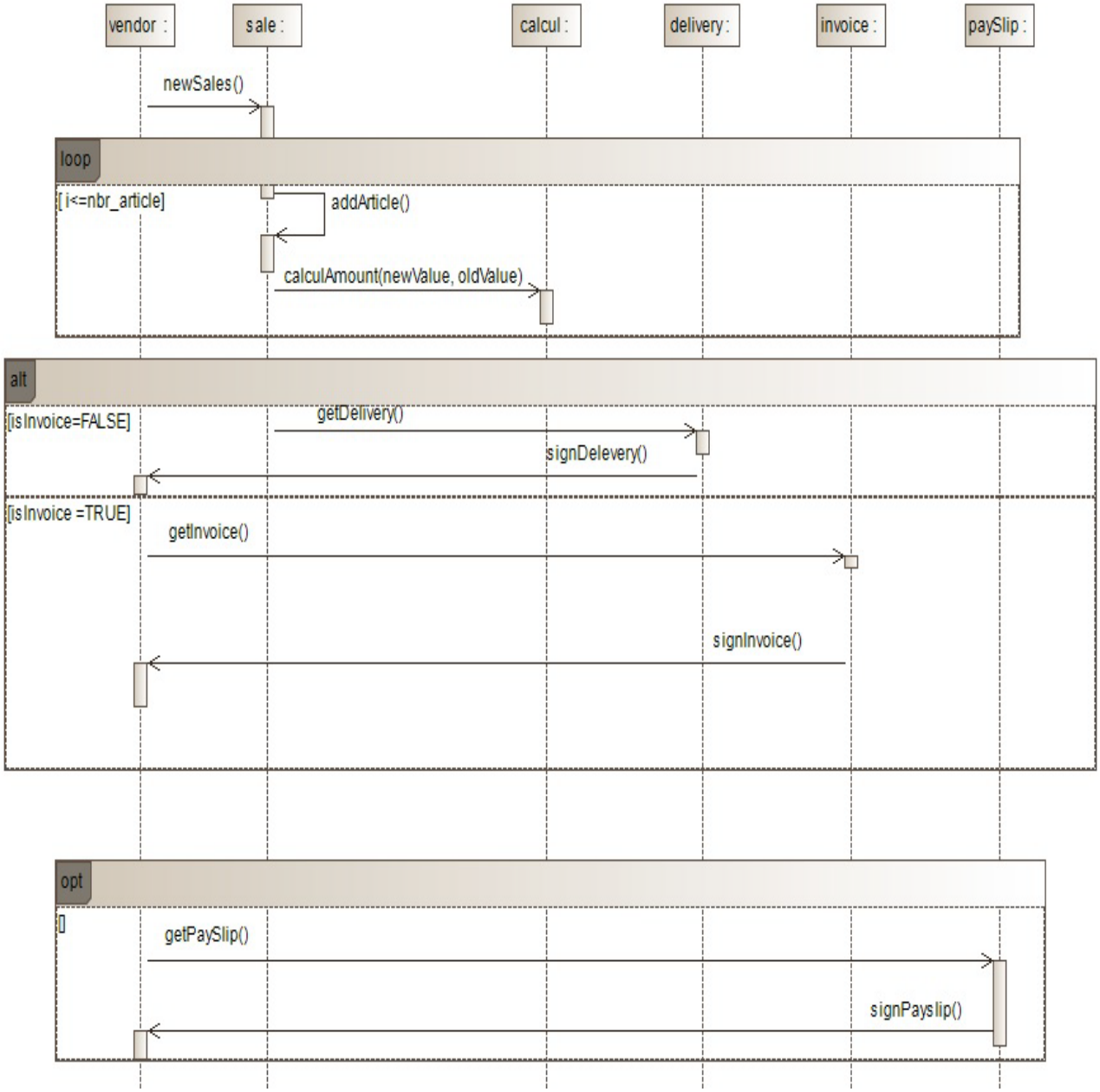

Fig. 9. Extracted HLSD.

In addition, our approach can be generalized to all object-oriented languages since it only uses text files for execution traces.

The colors are used to facilitate understinding the behavior of the system by subdividing it into several HLSD.

VI. Conclusion

Our work consists on proposing a new methodology for recovering an UML2 HLSD from execution trace using CPNs. For this, we first present a background of SD, CPN and reverse engineering. Then, we define several concepts which are essential for the understanding of our new approach. The approach starts by generating and collecting traces. Then, these traces are filtered and represented on CPNs in order to merge them. This merging is performed using an adapted version of the kBehavior algorithm that we have created. These CPNs are less complexes and more coherent than CPN in [8, 9]. The final obtained CPN use colors to differentiate between paths that represents different scenarios of the behavior of the system. This facilitates the understanding of the system. The approach succeeds to extract SD fragments operators such as seq, loop, alt and opt. It's also extracts UML2 operator conditions relating on alt, opt and loop which is not the case in [8, 9].

Our future work is to extract the fragment operator *par* which is important for multi-threading systems. Besides, we will try to handle the problem of extracting others UML2 diagrams like a state diagram and activity diagram.

REFERENCES

[1] Sommerville, I., "Software Engineering" Addison Wesley, 2000.

[2] B. Cornelissen, A. Zaidman, et A. Deursen, "A Controlled Experiment for Program Comprehension Through Trace Visualization," pp 2. IEEE Trans. on Software Engineering, 2011.

[3] K.-K. Lau and R. Arshad, "A Concise Classification of Reverse Engineering Approaches for Software Product Lines",vol. 4, 2016.

[4] IEEE. std 1219: Standard for Software Maintenance. IEEE Computer Society Press, Los Alamitos, CA, USA, 1998.

[5] OMG. Unified Modeling Language (OMG UML), Superstructure, Vol. 2, 2007.

[6] L. C. Briand, Y. Labiche, J. Leduc, "Towards the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software," IEEE Transactions on Software Engineering, vol. 32, no. 9, pp. 642-663, 2006.

[7] C. Bennett, D. Myers, M.-A. Storey, D. M. German, D. Ouellet, M. Salois, and P. Charland, "A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams," J. Softw. Maint. E vol., vol. 20, no. 4, pp. 291–315, 2008.

[8] Chafik B., El Mahi B. and Abdeslam J.: A "Dynamic Analysis for Reverse Engineering of Sequence Diagram Using CPN," Lecture Notes in Computer Science (ISSN: 0302-9743), 2018.

[9] Chafik B., El Mahi B., Abdeslam J. "A New Approach for Recovering High-Level Sequence Diagrams from Object-Oriented Applications," Elsevier Procedia Computer Science Journal (ISSN: 1877-0509), 2019.

[10] E. J. Chikofsky and J. H. Cross, II, "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software, vol. 7, no. 1, pp. 13-17, 1990.

[11] R. Kollmann and M. Gogolla, "Capturing Dynamic Program Behaviour with UML Collaboration Diagrams," in Proceedings of the 5th Conference on Software Maintenance and Reengineering (CSMR'01), pp 58-67. IEEE Computer Society, 2001.

[12] R. Kollmann, P. Selonen, E. Stroulia, T. Syst¨a, and A. Z¨undorf. "A Study on the Current State of the Art in Tool-Supported UML-based Static Reverse Engineering," in Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02), pp 22-32. IEEE Computer Society, 2002.

[13] A.Rountev, O. Volgin, and M. Reddoch, "Static Control-Flow Analysis for Reverse Engineering of UML Sequence Diagrams," in ACM SIGSOFT Software Engineering Notes, ACM, vol.31, no.1, pp. 96-102, 2005.

[14] A. Rountev and B.H. Connell, "Object Naming Analysis for Reverse-Engineered Sequence Diagrams," in Proceedings of the 27th International Conference on Software Engineering (ICSE'05), pp 254-263. ACM, 2005.

[15] Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue, "Extracting Sequence Diagram from Execution Trace of Java Program," International Workshop on Principles of Software Evolution (IWPSE'2005), pp. 148-151, 2005.

[16] Romain Delamare, Benoit Baudry, Yves Le Traon, "Reverse-engineering of UML 2.0 Sequence Diagrams from Execution Traces", in Proceedings of the workshop on Object-Oriented Reengineering at ECOOP 06, 2006.

[17] Tewfik Ziadi, Marcos Aur'elio Almeida da Silva, Lom Messan Hillah, Mikal Ziane. "A Fully Dynamic Approach to the Reverse Engineering of UML Sequence Diagrams," 16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS, Las Vegas, United States, 2011.

[18] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman, "Visualizing Test suites to Aid in Software Understanding," In Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07), pages 213-222. IEEE Computer Society, 2007.

[19] K. Jensen, "A brief introduction to coloured Petri nets," in Proceeding of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97) Workshop, LNCS, Springer-Verlag, vol. 1217. pp. 203–208, 1997.

[20] A. Jakimi, A. Sabraoui, E. Badidi, A. Salah, and M. El Koutbi, "Using UML Scenarios in B2b Systems," IIUM Engineering Journal, 2010

[21] AspectJ: The AspectJ project at Eclipse.org, http://www.eclipse.org/aspectj/.

[22] J. A. Brzozowski, "Derivatives of regular expressions," J.ACM, vol. 11, no. 4, pp. 481–494, 1964.

[23] L. Mariani, F. Pastore and M. Pezze. "Dynamic Analysis for Diagnosing Integration Faults," in IEEE Transactions on Software Engineering, vol. 37, no 4, pp. 486-508, 2011.

[24] A. Biermann and J. Feldmann,. "On the synthesis of finite state machines from samples of their behavior," IEEE Transactions on Computer, vol. 21, pp. 592–597, 1972.