Journal of Theoretical and Applied Information Technology
30th June 2020. Vol.98. No 12
© 2005 – ongoing JATIT & LLS

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

TOWARDS A DYNAMIC ANALYSIS OF LEGACY SYSTEMS FOR REVERSE-ENGINEERING INTERACTION DIAGRAMS

[1]EL MAHI BOUZIANE, [2]CHAFIK BAIDADA, [3]ABDESLAM JAKIMI

[1] GLISI Team, Department of Computer Science, Faculty of Sciences and Technics, Errachidia, Morocco
[2]Department of Computer Sciences, ENSA El Jadida,, Morocco
[2] GLISI Team, Department of Computer Science, Faculty of Sciences and Technics, Errachidia, Morocco

[1]bouzianeelmahi@gmail.com , [2]chafik29@gmail.com@abc.com, [3]ajakimi@yahoo.fr

ABSTRACT

Recently, reverse engineering has become widely recognized as a valuable process for extracting system abstractions and design information from existing software. Reverse engineering for legacy systems is used to retrieve missing design documentation from existing source code in an abstract model UML format for studying both the static structure and dynamic behavior of the system and for expanding the new features to the product. To help engineers to understand the behavior of these systems, a dynamic analysis technic is used to recover the UML sequence diagram of an object-oriented program. In this context, most existing approaches in addition to not filter execution traces, don't allow to extract properties of control structure corresponding to combined fragments operators such as loop, alt and opt. They can't also detect the operator par which is important in the case of multi-threading systems. In this paper, we propose a novel approach based on Colored Petri Nets (CPNs). This approach allows to generate UML2 sequence diagram with main combined fragment operators: seq, loop, alt, opt and par. It consists of four steps: trace collection, trace filtering, trace merging, and high level sequence diagram (HLSD) extraction. CPNs are used to abstract execution traces in order to facilitate their analysis.

Keywords: *Reverse Engineering, Legacy Systems, Sequence Diagram, Colored Petri Nets, Dynamic Analysis, Execution Traces*

1. INTRODUCTION

Reverse Engineering is the important building block in understanding and maintaining the code. Maintainability increases when the dynamic behavior of the object is translated into design from the source code. Recently, new software engineering methods aim to increase the productivity and quality of systems under development. However, in the reality of the software industry, these methods are not always respected. Indeed, several existing systems suffer from problems such as missing or incomplete documentation and non-compliance with the design when coding the software.

Software engineering activities like maintenance, testing, and integration deal with legacy systems. A legacy system, is a system where is not possible to understand all the fundamental concepts that shaped it as they could be neither available nor existent for understanding.

The most important aspect of all these processes is the comprehension of the components of existing systems and the relationships existing between them. According to [1] up to 60% of maintenance time is spent on understanding software. Especially since most of these systems generally suffer from several problems, such as unavailability of developers, obsolete development methods used to code the software and missing documentation. Therefore, it is important to develop techniques to obtain an abstract representation to facilitate the understanding of these systems.

A proven and effective technique to address this problem is reverse engineering of UML models. It can be defined as a means of analyzing the source code of these systems and representing it in a form with a higher level of abstraction to make it easier to understand. Reverse engineering can help to understand existing systems by retrieving models from their available artifacts. The IEEE-1219 [3] standard recommends reverse engineering as a technological solution to deal with legacy systems without updated documentation. In the object-oriented world, the target modeling language most used for reverse engineering is UML (Unified Modeling Language) [4] due to its significant presence in the industry. To better understand the behavior of these systems, dynamic models are needed, such as Sequence Diagrams (SDs). UML SDs take an important place in software engineering. They help software engineers to understand the source code of existing object-oriented software systems through the visualization of interactions between their objects [5]. To extract SDs describing the behavior of a system, we concentrate on reverse engineering relying on dynamic analysis. As mentioned in [6], dynamic analysis is more adapted to the reverse engineering of SDs due to inheritance, polymorphism and dynamic binding.

Section 2 of this paper giveses related works. Section 3 introduces a background in reverse engineering of UML SDs using CPNs. Section 4 outlines the proposed methology and approach. Finally, section 5 provides some concluding remarks and points out some future works.

2. RELATED WORK

Reverse engineering as opposite of forward engineering is 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 [7].

In reverse engineering, program analysis usually takes place either through two kinds of analyses: static analysis and dynamic analysis. Static analysis concerns analyzing the source code of a system by building an abstracted model of it. Various approaches have been developed to capture a system's behavior through static analysis [8, 9, 10, 11]. One of the main objectives of these works is that of Rountev et al. [11]. They proposed an approach for the extraction of SDs from the source code of a system through building control flow graphs. In this study, the nodes represent the basic blocks of a program, and the links represent all kinds of interactions between these blocks.

The dynamic analysis, on the other hand, is to analyze a software system under execution. 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. This dynamic analysis supports polymorphism and late binding, unlike static analysis. Several works try to generate SDs by analyzing the execution traces. Taniguchi et al. [12] propose an automatic approach for the reverse engineering of SDs from the execution traces of an object-oriented program. They use four additional rules to optimize the size of the execution traces by detecting similarity between sub-trees and replace merging them. In [13], they try to build a High-Level Sequence Diagram (HLSD) from combined fragments using the different states of the system. This approach consists of two phases. During the first phase, a simple SD is generated containing just the method calls. The second phase enables to draw HLSD by combining the diagrams generated in the first step. The combination process is done by analyzing the different states of the system. In [14], it is proposed an approach based on dynamic analysis. They use LTS (Labeled Transition System) for modeling execution traces. Then an HLSD is generated from this LTS.

These approaches have succeeded in generating representative SD. However, they recognize some limitations. These limitations include information filtering problems. For this reason, in [15] Cornelissen et al. defined a catalog of abstractions and filtering in the context of reverse engineering of sequence diagrams. The approaches mentioned above do not use these filtering technics.

3. UML SEQUENCE DIAGRAM AND CPN

3.1. UML Sequence Diagrams (SD)

In this work, we chose to use excrat sequence diagrams (SD) because of their wide use in different domains. A SD shows interactions among a set of objects in temporal order, which is good for understanding timing and interaction issues. An SD is the most used diagram for capturing inter-object behavior. Graphically, an SD has two dimensions: a horizontal dimension representing the instances participating in the scenario, and a vertical dimension representing time. SD is typically associated with use case realizations in the logical view of the system under development. It has been significantly changed in UML 2.0 [4].

Notable improvements include the ability to define HLSDs. An HLSD is an SD that refers to a set of Basic SD (BSD) and composes them using a set of interaction operators. The main operators are seq for sequence, alt for alternatives, loop for iterative actions, and par for parallelism. Figure 1 shows an example of an HLSD composed of two BSDs using the operators *loop* and *alt.* For example, the basic SD BSD1 describes the interactions between two instances a1 (instance of the class A) and b1 (instance of the B class). The behavior specified in the HLSD is then equivalent to the expression while (C1) (if (C2) then BSD1 else BSD2).

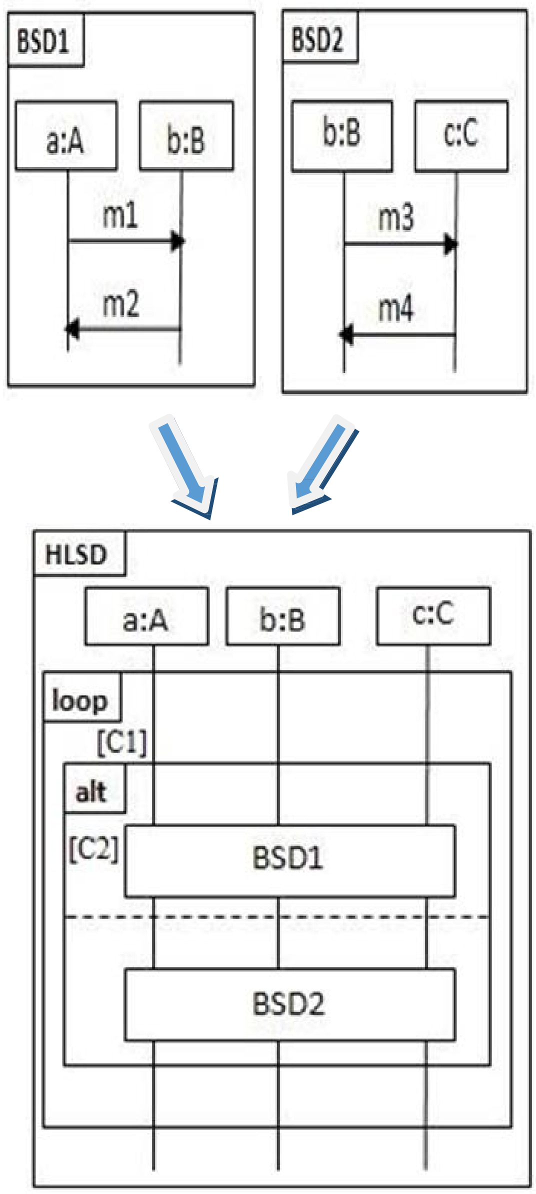

Figure 1. Example of an HLSD

3.2 Execution Traces

Mining specifications from logs of execution traces has attracted much research effort in recent years since the mined specifications, such as program invariants, temporal rules, association patterns, or various behavioral models, may be used to improve program documentation, comprehension, and verification.

To build an HLSD using dynamic analysis, we have to generate traces of program executions. Each trace corresponds to a scenario of a given use case. In what follows, we introduce a set of definitions that are necessary to understand the approach.

Definition 1: A trace line is a method invocation, control structure or parallelism operator.

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

- Sender is the caller object, expressed in the form threadNumber: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.

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

- controlType has one of the following values: IF, ELSE, SWITCH, CASE, DEFAULT, FOR, or WHILE.
- status expresses the start or the end of the control structure.
- condition (optional) is the condition expression associated with IF, CASE, FOR, or WHILE.

Definition 4: A parallelism operator is a tuple T3=<parallelismOperator, status> where:

- parallelismOperator is the operator: PAR.
- status expresses the start or the end of the parallel invocations.

Definition 5: (Equivalence between method invocations): The method invocations I1 = <s1, m1, r1> and I2 = <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 and are created in the same thread.

- the messages m1 and m2 concern the same method and have the same arguments.

Definition 6: An execution trace is a set of trace lines.

Table 1 shows an example of generated execution traces where each trace corresponds to a given scenario of a use case. Trace1 describes Scenario1 and Trace2 describes Scenario2.

Table 1. An example of traces

Scenario1 : Trace 1
L7. 0:Pack1:B:b|m4()|Pack1:A:a
L8. WHILE |BEGIN |condition1
L9. 0:Pack1:B:b|m5()|Pack1:A:a
L10. 0:Pack1:B:b|m6()|Pack1:A:a
L11. WHILE|END
L12. PAR | BEGIN
L7. 1:Pack1:B:b|m4()|Pack1:A:a
L13. PAR |END
Scenario2: Trace 2
L0. IF | BEGIN | condition2
L1. 0:Pack1:A:a |m1()|Pack1:B:b
L2. ELSE | BEGIN
L3. 0:Pack2:C:c |m2()|Pack2:D:d
L4. ELSE| END
L5. IF | END
L6. 0:Pack2:D:d |m3()|Pack2:D:c
L1. 0:Pack1:A:a |m1()|Pack1:B:b

These traces are composed of several lines. L0 to L13 refers to the number of each line. Pack1 and Pack2 represent the packages to which classes A, B, C, and D belong. m1() to m6() correspond to the methods calls of objects a, b, c, and d. The numbers 0 and 1 correspond to the IDs of the threads.

3.3. Colored Petri Nets (CPN)

Petri nets [16] are well-known and developed formalism with a rich theory, practical applications ranging from communication networks to healthcare systems and are supported by a wide range of commercial and non-commercial tools. CPN is a backward-compatible extension of Petri nets. CPN preserves useful properties of Petri nets and at the same time extends the initial formalism to allow the distinction between tokens by attaching a data value to them. This distinction is expressed graphically by having tokens with different colors.

A Petri Net block is a subnet of the Petri Net that with one initial place and one final place. Those places refer respectively to the precondition and the post-condition of the subnet. From the many existing variants of Petri nets, CPN is used in composing and integrating scenarios that are represented in the form of SDs [17].

Four operators for composing scenarios have been implemented: sequential, conditional, iterative and concurrent. CPNs suit our approach as they can map an HLSD efficiently (figure 2, 3). Transitions can represent BSD or operators such as “alt”, “loop” (figure 2), and par (figure 3). Colors are used to distinguish between traces. All places from the same trace have the same color.

From what precedes, we can conclude that, for an HLSD, we can generate a CPN that can represent all major UML SD operators such as alt, par, and loop. We can also do the reverse transformation by mapping a CPN into an HLSD.

The problem that arises is how we can reverse this process, i.e., how, from execution traces, can we generate a CPN that can be mapped onto an HLSD? In the next section, we propose an approach that deals with this problem.

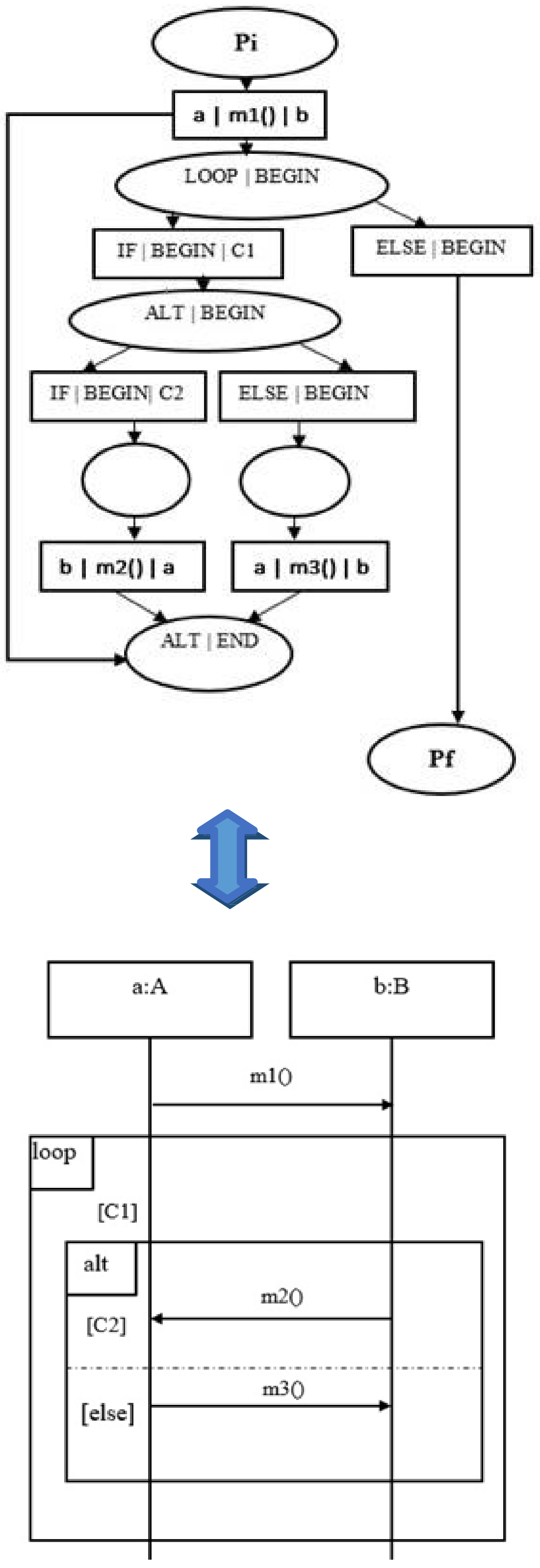

Figure 2. HLSD mapped onto CPN with operators « loop » and « alt »

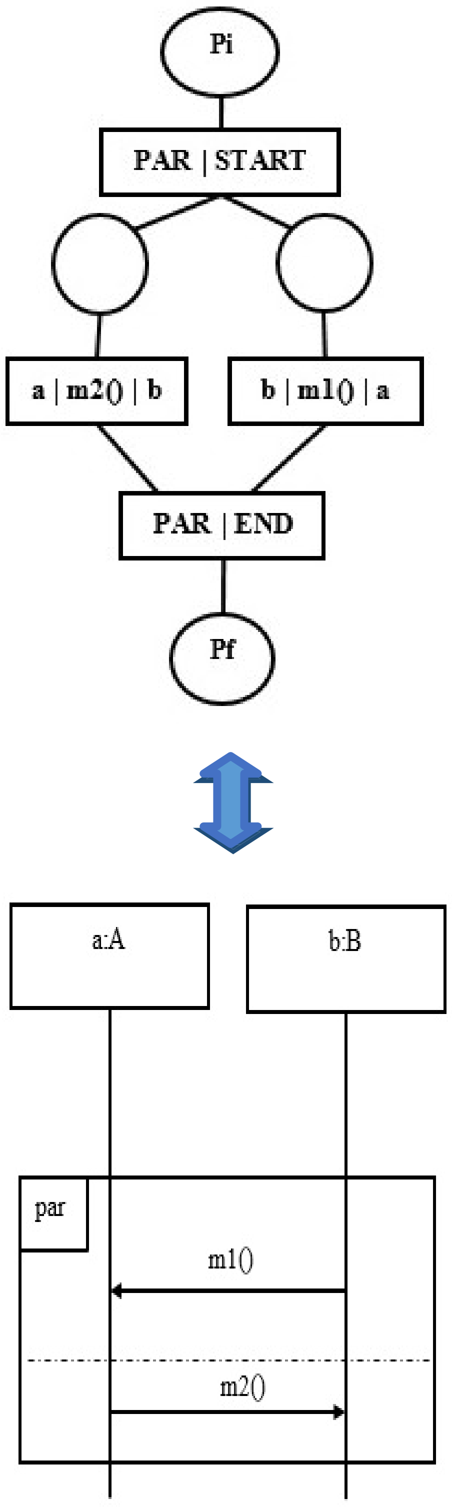

Figure 3. HLSD mapped onto CPN with operator “par”

4. DESCRIPTION OF THE APPROACH

In this section, we give an overview of the reverse engineering of UML High-Level sequence diagram for the system.

The approach is defined in four main steps (figure 4) : trace collection, trace filtering, trace merging, and HLSD extraction.

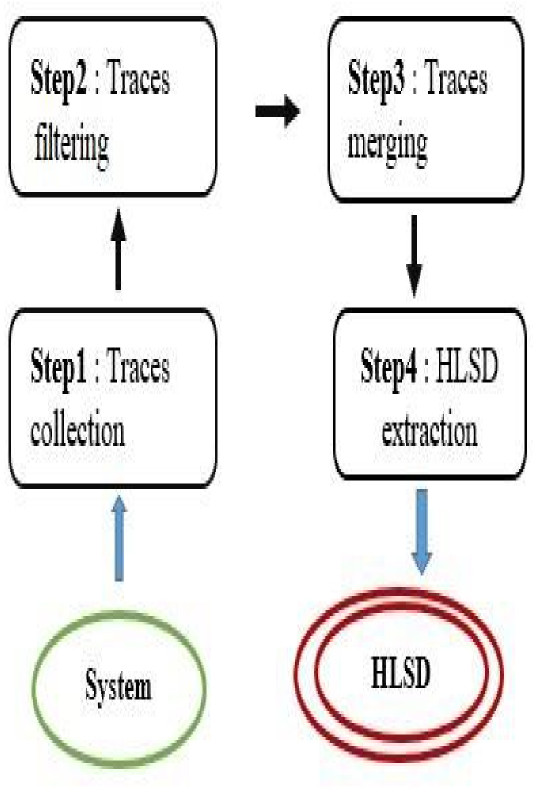

Figure 4. Overview of our approach.

4.1 Trace collection

To extract an HLSD from an object-oriented program, we concentrate on reverse engineering relying on dynamic analysis. As mentioned in [5], dynamic analysis is more suited to the reverse engineering of SDs of object-oriented systems. This dynamic analysis is usually performed using execution traces. There are multiple ways to generate execution traces [1]. These ways can include instrumentation of the source code, bytes code, virtual machines (for java programs for instance) or the use of a customized debugger. From these technics, we choose to use code instrumentation. Java software systems, we chose AspectJ [18]. This one can be used as a Java intermediate code instrumentation tool. It allows the retrieval of the following information created during the execution of the program: occurrence of objects, messages that circulate between them, loops, conditions and threads.

The system behavior is related to the environment entry data, in particular, values introduced by the user to initialize specific system variables. Thus, one execution session is not enough to identify all system behaviors. Therefore, we chose to run the system several times to generate different executions traces. Each execution trace corresponds to a particular scenario of a given use case of the system. The form of collected traces can differ from one tool to another, which has forced us to develop an adapter that reorganizes the traces into a new adapted form as described in the definitions 1, 2, and 3. The role of the adapter is to restructure the trace into a form appropriate to the processing of merging traces.

4.2. Trace filtering

The generated execution traces contain a lot of information about all classes composing the system. 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.

4.3. Trace merging

Trace merging is the main step of our approach. It deals with the known problem of analyzing traces. Indeed, one of the major challenges to reverse engineering an HLSD is analyzing the multiple execution traces to identify operators and method invocations throughout the input traces. Independently from the reverse engineering of SDs, the challenge of merging traces is well identified in the grammar inference domain where several well-defined techniques were proposed [19].

In this subsection, we chose to use CPNs to merge these execution traces. The process is done in two sub-steps: CPN initialization and

Merging.

a) CPN initialization

In this sub-step, one CPN for each execution trace is generated. All the trace lines are transformed into transitions in CPNs except those which express the start or the end of iterative control structure like LOOP | START and LOOP | END. These line traces are transformed into places.

The objective of this sub-step is also to extract the operator par. The generated child threads events are delimited by the trace lines PAR | BEGIN and PAR | END. The algorithm creates the transition labeled PAR | BEGIN with two or more outgoing edges corresponding to the number of the created threads to indicate the beginning of a concurrent behavior. It also creates the transition PAR | END to indicate its end. As shown in subsection 3.2, every trace line has a thread number. The algorithm that compares between threads numbers to create for each trace line the correspondent CPN. It focuses on the threads number creates a CPN path for all trace lines that have the same thread number. These paths are attached to the transition "par". All places that represent trace lines have the same color. These colors allow us to distinguish between the scenarios and give the possibility of subdividing an HLSD into several HLSDs to facilitate the task of understanding the system.

b) CPN Merging

In the previous sub-step, every trace has a correspondent CPN and includes as transitions only method invocations or the operator par. In the second sub-step, the CPNs of the different traces are synthesized to obtain a single CPN that merges the initial traces. This is done by using the algorithm kBehavior [20]. This algorithm is inspired by the kTail algorithm [21,22]. Both are used to build an automaton from execution traces. These techniques allow learning a regular target grammar from a set of sequences. For this, a generalization procedure of the automaton is applied iteratively by successive fusion of equivalent states. kTail has a major limitation: it is not able to reuse already learned knowledge to adapt to newly generated traces, which is not the case for kBehavior. In our case, we took the main CPNs. When a new trace is given to the algorithm, adapted kBehavior first identifies sub-traces of the input trace that are accepted by a sub-CPN in the current CPN (the sub-traces must have a minimum length of k; otherwise they are considered too short to be relevant). Then our adapted kBehavior algorithm extends the CPN with the addition of new branches that suitably connect the identified sub-CPN, producing a new version of the CPN that accepts the entire input trace. An example of traces merging is illustrated in Figure 5.

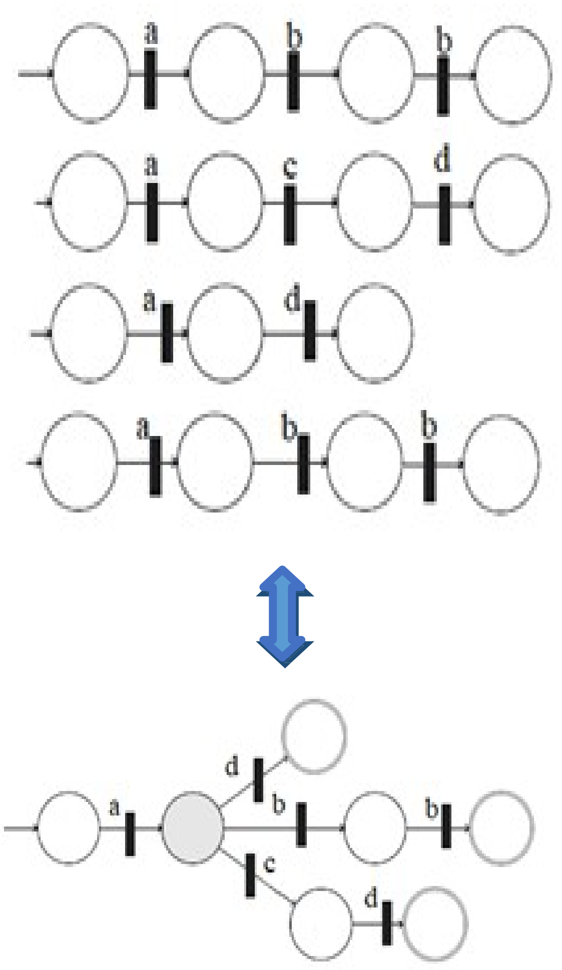

Figure 5. Example of merging traces using our adapted kBehavior algorithm with k=2

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 any more 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 Figure 6.

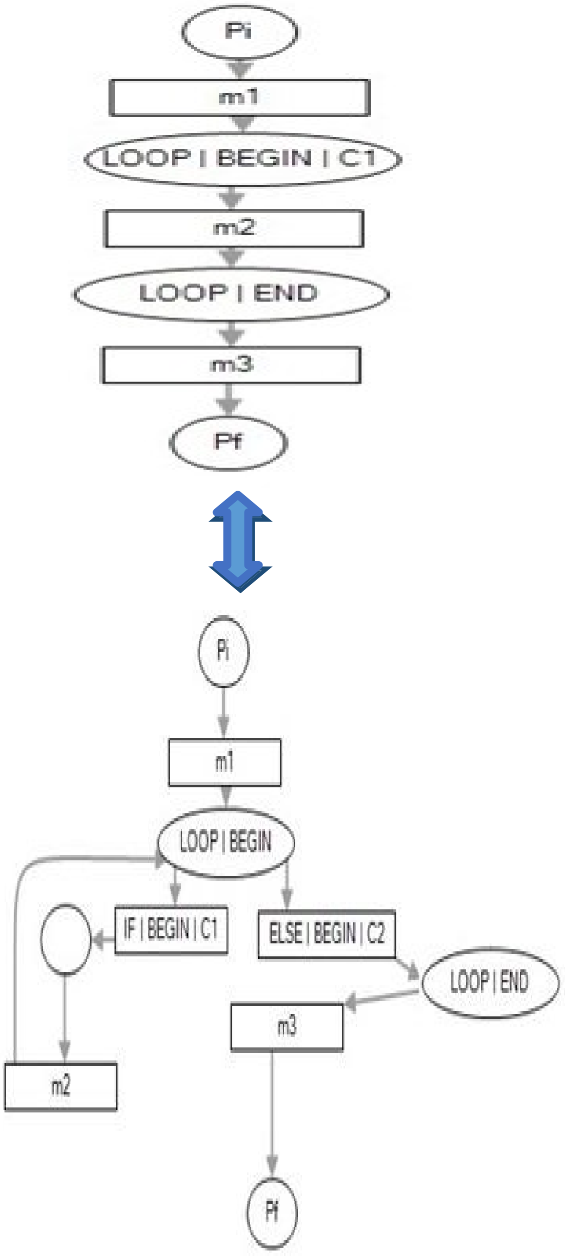

Figure 6. CPN corresponding to scenario1

4.4. HLSD extraction

In this step, we can easily build an HLSD (Figure 7) 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.
- **Rule 8:** A Transition T1 that contains the operator "PAR | BEGIN" **or** refers to BSD with the operators par.
- **Rule 9**: The CPN paths coming after the transition "PAR | BEGIN" and ending on the transition "PAR | END" are transformed into combined fragments with the operators par.

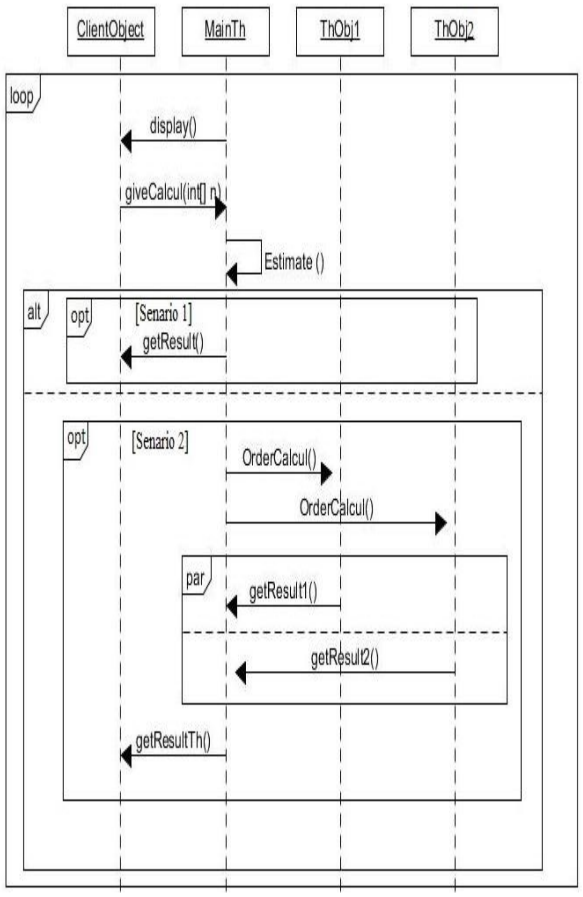

Figure 7. HDLS with operators (Loop, Par, Alt)

A HDLS shows a dynamic collaboration between a number of objects. It is widely used by businessmen and software developers to document and understand requirements for new and existing systems.

Combined fragment is an interaction fragment which defines a combination (expression) of interaction fragments. A combined fragment is defined by an interaction operator and corresponding interaction operands. A combined fragment consists of one or more interaction operands, and each of these encloses one or more messages, interaction uses, or combined fragments.

Through the use of combined fragments the user will be able to describe a number of traces in a compact and concise manner. In this HDLS, combined fragments let you show loops, branches, and other alternatives.

5. CONCLUSION

Organizations are highly dependent on their software in carrying out their daily activities. Unfortunately, the repeated changes that are applied to these systems make their evolution difficult. It is difficult for developers to modify or change the source code when they do not understand the original system. In software engineering, developers generally base code development on design documents to build software that matches the design requirements. UML sequence diagrams are commonly used to represent object interactions in software systems. This work considers the problem of extracting UML sequence diagrams from existing code for the purposes of software understanding and testing.

Reverse Engineering is focused on the challenging task of understanding legacy program code without having suitable documentation. In this paper, we presented an overview of our approach for the reverse engineering of sequence diagrams of an object-oriented software system. The approach is based on dynamic analysis of legacy systems. We use CPNs to model execution traces. Then these CPNs are merged into a single CPN using the adaptive Kbehavior. Finally, the result CPN is translated into a HLSD by applying transformation rules. The colors of CPN are used to distinguish between scenarios and therefore enables subdividing an HLSD into several HLSDs to facilitate the task of understanding the system. Our approach has also been successful in detecting the operator par and conditions in alt and loop operators.

This study presented a transformation method that converts the source code (execution traces) a UML sequence diagram as an aid in analyzing and understanding legacy systems. The future works of this research include the following areas:

- Evaluate and validate our approach to more simple and complex systems and try to handle the problem of extracting other types of UML diagrams and modernization of legacy systems [23,24,25].

- Merging our study and many works [26,27,28] will enable the visualization of object-oriented software behavior and algorithmic structure and thereby enhance the development, maintenance practices and communications in scientific and engineering software [26,27, 28].

REFERENCES

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

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

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

[4] OMG. Unified Modeling Language (OMG UML), *Superstructure. 2018.*

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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

[17] A. Jakimi, A. Sabraoui, E. Badidi, A. Salah, and M. El Koutbi, "Using UML Scenario in B2B Systems," *IIUM Engineering Journal, 2010.*

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

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

[20] 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.

[21] 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.

[22] 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.

[23] G. Chénard, I. Khriss and A. Salah, "Towards the Discovery of Implementation Platform

Description Models of Legacy Object-Oriented Systems," *in Workshop on Processes for Software Evolution and Maintenance (WoPSEM 2010)* IEEE, 2010.

[24] G. Chénard, I. Khriss and A. Salah, "Chénard, G., Khriss, I. and Salah, A. Towards the Automatic Discovery of Platform Transformation Templates of Legacy Object-Oriented Systems," *in Models and Evolution (ME) 2012 workshop a satellite event at MoDELS 2012*, Insbrusck, Austria, 2012.

[25] H. Abdelmalek, G. Chénard, I. Khriss and A. Jakimi, A Bimodal Approach for the Discovery of a View of the Implementation Platform of Legacy Object-Oriented Systems under Modernization Process, *In Proceedings of 35th International Conference on Computers and Their Applications CATA'20*, vol 69, pages 98—111. 2020.

[26] Aziz Nanthaamornphong and Anawat Leatongkam, Extended ForUML for Automatic Generation of UML Sequence Diagrams from Object-Oriented Fortran, Hindawi Scientific Programming Volume 2019, ID 2542686, 22 pages. 2019.

[27] Sabine Wolny , Alexandra Mazak , Manuel Wimmer, Automatic Reverse Engineering of Interaction Models from System Logs, *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Zaragoza, Spain, 2019

[28] Lina Čeponienė, Vaidotas Drungilas, Mantas Jurgelaitis, Jonas Čeponis, A Method for Reverse Engineering UML Use Case Model for Websites, *Journal of Information Technology and Control* Vol. 47 / No. 4 / 2018 pp. 623-638, 2018.