Towards New Hybrid Approach of the Reverse Engineering of UML Sequence Diagram

Chafik Baidada[1], Abdeslam Jakimi[1]
Software Engineering & Information Systems Engineering Team
Computer Sciences Department, Moulay Ismaïl University, FST Errachidia, Morocco
{chafik29@gmail.com; ajakimi@yahoo.fr}

***Abstract*— To fully understand the behavior of a program, it is important to have automatic techniques that generate UML (Unified Modeling Language) models representing the behavior of the system. Reverse engineering techniques, either through dynamic analysis of the running application or static analysis of the source code, are used to help gain this understanding. Each type of technique has its limitations. The major limitation of dynamic analysis is the need of a system expert user who knows the different behaviors of the system. Static analysis has also limits especially with dynamic links and polymorphism states. In this paper, we propose an approach to automatically extract UML sequence diagrams from object-oriented programming languages. Our approach combines dynamic and static analyses to get the best of both approaches. Our method uses static analysis to identify all different system entries values that can be used to guide the dynamic analysis.**

I. INTRODUCTION

Recently, software is becoming increasingly complex. Traditional software engineering research and development focuses on increasing the productivity and quality of systems under development or being planned. Without diminishing the importance of software engineering activities focusing on initial design and development, empirical evidence suggests that significant resources are devoted to reversing the effects of poorly designed or neglected software systems. In a perfect world, all software systems, past and present, would be developed and maintained with the benefit of well-structured software engineering guidelines. In the reality, many systems are not or have had their structured design negated. The understanding of these programs is an essential part of maintenance, reuse, validation and other activities of software engineering. An important part of maintenance time is often devoted to reading the code to understand the functionality of the program. According to some studies, up to 60% of the maintenance is devoted to understanding the software [1]. Therefore, it is important to develop tools and techniques that facilitate the task of understanding such systems because the documentation is often absent, outdated or incomplete. An effective recognition technique to understand such programs is reverse engineering. Many works investigate the reverse-engineering of UML static models, such as class diagrams. However, there is little work on reverse-engineering dynamic models, although, in addition to static models, the UML includes notations to specify the dynamic behavior of programs, such as sequence diagrams and state charts.

Dynamic models of programs are as important as static models because they allow maintainers to identify complex interactions among objects and to disambiguate message sends when inheritance, delegation, polymorphism, dynamic binding, reflection are used intensively (for example, when using design patterns such as the Abstract Factory, Observer).

This paper is organized as follows. The next section we define UML and specify the problem of reverse engineering dynamic UML models. In section III, we discuss related work. In section IV, we present our reverse engineering methodology. Finally in section V, we present our conclusions and discuss future work.

II. UML AND REVERSE ENGINEERING

UML

In the world of object-oriented, target language most used for reverse engineering is UML [2] due to its significant presence in the industry.

UML (Unified Modeling Language) is a graphical modeling language developed in response to the call for proposals of the OMG (Object Management Group) to define the standard notation for modeling object-oriented applications. It is the result of the fusion of several languages such as OMT (Object Modeling Technique) and OOSE (object-oriented software engineering) and Booch. The main authors of the UML notation are Grady Booch, Ivar Jacobson, and Jim Rumbaugh. UML, in its version 2.x offers fourteen diagrams to model a computer application throughout its lifecycle. These diagrams are divided into two categories: static diagrams (structural) and dynamic diagrams (behavioral). The diagrams in the first category are used to describe the architecture and composition of the system while the second category shows the system behavior, interactions of objects and their changes over time. One of the most used diagrams from the second category is the sequence diagram. Graphically, a sequence diagram has two dimensions: a horizontal dimension representing the instances participating in the scenario, and a vertical dimension representing time. Sequence diagrams are typically associated with use case realizations in the logical view of the system under development.

Sequence Diagrams (SD) have been significantly changed in UML2. Notable improvements include the ability to define what is called combined SD. A Combined SD is a sequence diagram that refers to a set of SD and composes them using a set of interaction operator. The main operators are: seq, alt, opt, loop, and par. The seq operator specifies a weak sequence between the behaviors of two SD. The alt operator defines a choice between a set of SD. The opt operator specifies the

978-1-5090-0751-6/16/$31.00 ©2016 IEEE

option, while the loop operator define an iteration of a SD. The par operator allows specifying concurrency between SD.
Figure 1 shows two basic Sequence Diagrams (BSD): BSD1 and BSD2. The basic SD BSD1 describes the interactions between two instances a (instance of the A class) and b (instance of the B class).
The behavior specified in the high level sequence diagram (HLSD) of figure 1 is equivalent to the expression loop (SD2 alt SD3).

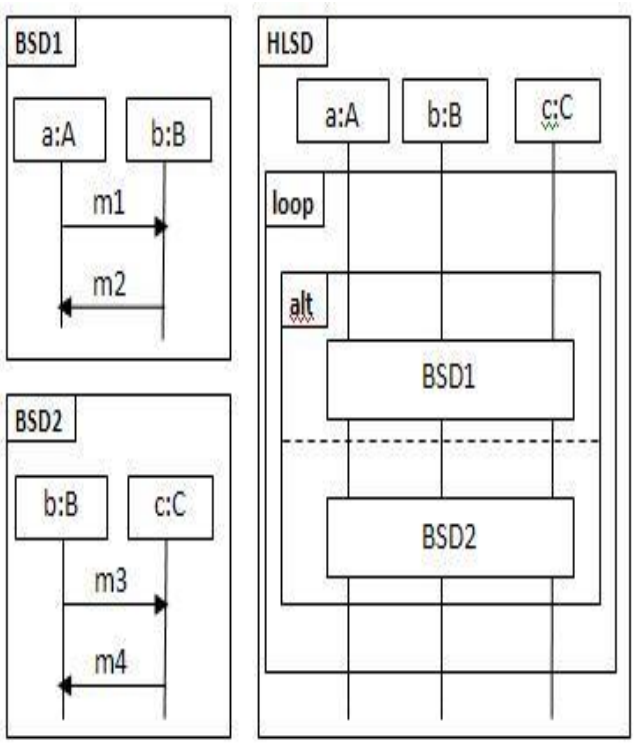

Fig 1. Example of UML2 Sequence diagrams.

Reverse engineering

Forward engineering is opposite of reverse engineering and it distinguish the traditional software engineering process from reverse engineering. Forward engineering takes sequences from feasibility study through designing its implementation.

Chikofsky and Cross [3] made a very successful attempt at providing some precise and long standing definitions for much of the terminology used to this day in the field of Reverse Engineering.

Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at higher levels of abstraction [3].

This technique is widely used in several disciplines including new technologies such as mechanics, electronics, etc. In software engineering, the process of reverse engineering code to produce software dynamic models is to convert the code files to design models such as UML sequence diagram, state diagram, etc. the goal is to increase the level of abstraction to better understand the dynamic behavior of software. This is due to the fact that quality attributes such as performance and reliability are mainly dependent on the software dynamic behavior.

Reverse engineering dynamic UML models is a very efficient way to understand complex software whose source code is absent or documentation is outdated. This can be achieved either by looking at the code (static analysis), or by analyzing the running application (dynamic analysis). Static analysis faces problems related to the highly dynamic nature of some application. Even the structure of the some objects of the application might be defined dynamically at runtime, with only a basic skeleton defined statically in the code.

Dynamic analysis solves some of these issues by analyzing the actual running systems. However, it also faces problems. The behavior of the applications depends on both the internal state of the interactive computing system, and on the inputs provided. This can be avoided by executing the system several times. This also needs a system expert user who knows the different system entries values and environment states.

In this paper, we focus on the reverse engineering of UML sequence diagram.

III. Related Work

Several studies have been performed on the reverse engineering of dynamic UML diagram [4,5,6,7,8,9,10,11]. We distinguish two types of analysis: static and dynamic.

Static analysis is to use the code structure to generate the sequence diagram. One of the main works based on static analysis is that Rountev et al. [4]. They describe a first algorithm to reverse-engineer UML v2.0 sequence diagrams by control flow analysis of a program source code. They map control-flow graphs (CFG) to the control flow primitives of UML v2.0 sequence diagrams. They also introduce behavior preserving transformations to reduce the size (number of basic sequence diagrams, number of message sends. . .) of the reverse-engineering sequence diagrams. Their approach does not consider data obtained from dynamic analyses and thus is limited to the accuracy of the control flow analysis. Also, they do not attempt to perform higher-level analyses on the reverse-engineered diagrams.

A control flow graph is a graph whose nodes are the foundation blocks of the program and arcs represent the connections of the program (conditional or not). Figure 2 shows an example of a control flow graph of a Java method. The control flow graph can be constructed easily and statically from the source code of a program. An instruction without connection block is represented by a single node of the graph. Conditions are also represented by a base block, with two outgoing edges: a in the case where the condition is met (labeled "T") and the other in the case where the condition is false (labeled "F").

The dynamic analysis is to analyze the performance of the application using execution traces. The traces are modeled by a sequence of events occurring during the execution.

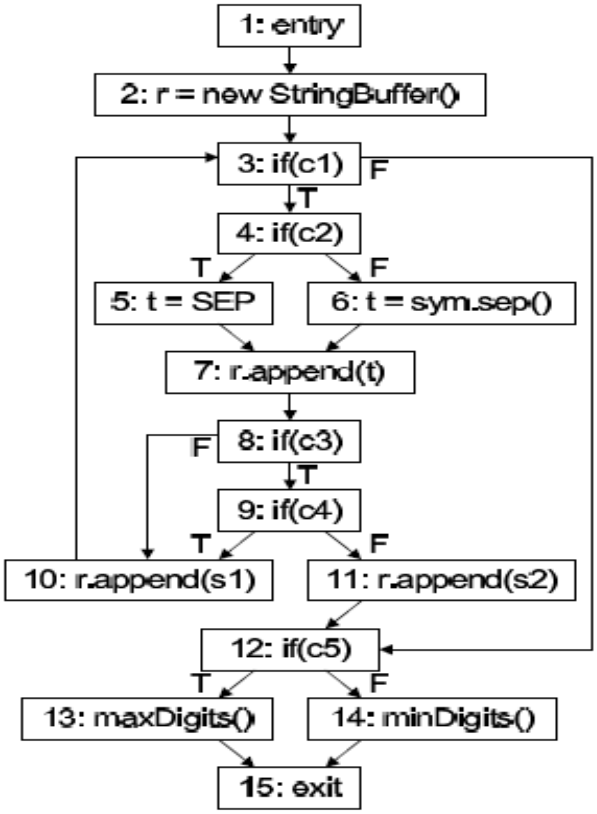

Fig 2. An example of CFG

Several studies try to generate the sequence diagram by analyzing the execution traces. In [5] is proposed an approach to build a high-level sequence diagram incrementally from basic diagram using the operators introduced by UML 2.0. In [6] they try to build a high-level sequence diagram from combined fragment using the state vector describing the system. In [7], it is proposed an approach completely dynamic based on the LTS (labeled transition system) for merger traces collected and generate a high-level sequence diagram.

These approaches have succeeded in generating representative UML behavior (sequence diagrams). But they recognize some limitations. These limitations include the information filtering problem. Thus, the resulting sequence diagram contains a lot of useless information that does not help to understand the software.

IV. METHODOLOGY

The reverse engineering of sequence diagram consists of extracting high-level models that help understand the behavior of existing software systems. The proposed approach for reverse engineering of UML sequence diagrams is defined in four main steps (Fig 1.): (i) extracting system entries values from source code (ii) traces collection and filtering, (iii) traces transformation into colored Petri nets and (iv) UML diagram extraction.

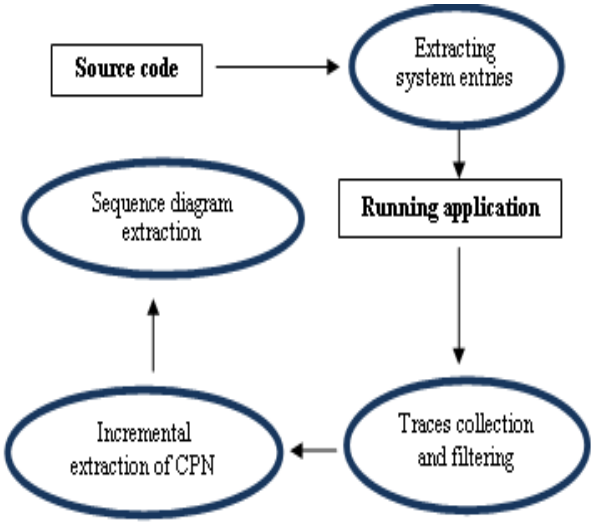

Figure 3. , Overview of the proposed process

(i). extracting system entries values from source code.

The major limitation as mentioned above in dynamic analysis is the need of a system expert who knows all the possible behaviors of the system. This prevents full automation of dynamic approaches. To overcome this problem, we choose to add a static analysis step.

This step is very important. It consists of reading the source code of the application and building CFG. This can be done easily as mentioned in [4].The purpose is to identify relevant input values by analyzing the conditions present in CFG. This input values which will be used and the next step must guarantee that all possible behaviors of the system will be executed. These input values are grouped according to the different use cases of the system

(ii). Traces collection and filtering.

In this step, we concentrate on reverse engineering relies on dynamic analysis. This dynamic analysis is usually performed using execution traces. There are multiple ways to generate execution traces [1]. This can include instrumentation of source code, virtual machines (ex: java programs) or the use of a customized debugger.

The aim at this stage is to collect the major events occurring during the system executions. The system behavior is related to the environment entry data, in particular, values generated by the previous step to initialize specific system variables. Thus, one execution session is not enough to identify all system behaviors. So we chose to run the system several times with different input values to generate different executions traces. Each execution trace corresponds to a particular scenario of a given service (use case) of the system. After that, a filtering process is applied for traces. This process is based on the package of the object present in the line trace. All lines traces that contain a package that is not interesting will be removed. For example all objects on the package relatives of the GUI of the system such as swing for java will be ignored. This process enables us to concentrate on the main behavior of the system.

(iii). Incremental extraction of formal or semi-formal techniques.

This is the main step of the approach. It deals with the known

problem of analyzing traces. Indeed, one of the major challenges to reverse engineering high level sequence diagram is to analyzing the multiple execution traces to identify common and method invocations throughout the input traces.
We use colored petri nets (CPN) to deal with this problem.
CPN suit our approach when they can map efficiently a high level sequence diagram. Places represent basic sequence diagram and transitions represent operator such as **alt, loop, seq, par**. Colors are used to distinguish between places. All places from the same trace have the same color. That is very helpful to distinguish between scenarios in a HLSD.
We formalize an algorithm that takes several execution traces as input and generate incrementally a colored petri net that represents the system behavior. First the algorithm scans the execution traces line by line. Each line is analyzed if it is a new one, a new place is created. All places that represent lines of a trace have the same color. The algorithm creates also transition. It associates for each place the correspondent transition by analyzing previous and current trace.
The traces analysis focuses on the number of thread to detect the "par" operator. If the lines belong to the same thread the operators "seq" and "alt" can be identified easily by simple comparison between previous and current line. The operators "loop" is obtained if the current line already exists and belong to the same trace.

(iv). Sequence diagram extraction

In this activity, we generate and build the UML behavioral models using the transformation models rules.

Figure 4 show that a CPN can be mapped easily into sequence. Places represent basic sequence diagram and transitions represent operator.

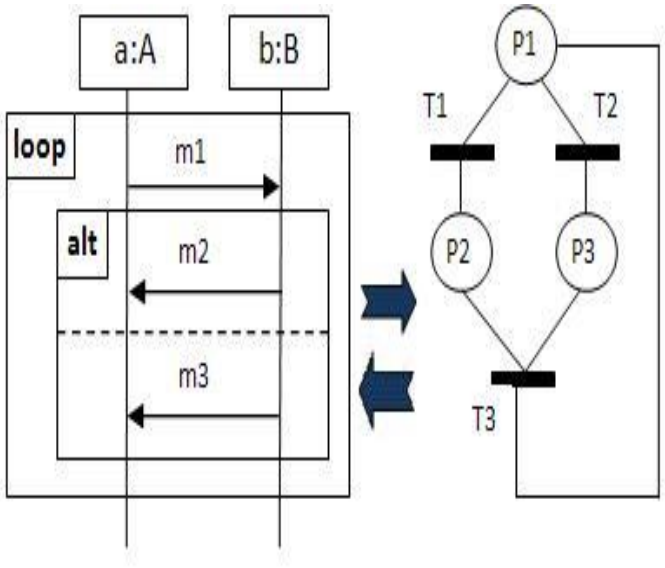

Figure 4. CPN mapped in sequence diagram

V. CONCLUSION

Reverse engineering is one the essential processes of software maintenance. It involves high risk especially when the maintenance and development teams are different. Extracting the static model from the source code may not be motivating factor for software maintenance. Many reverse engineering tools and approaches are proposed in literature. However each represents only a subset of operational requirements.

Reverse-engineering aims to provide design models from existing software. Hence, this can facilitate program comprehension and by consequence maintenance and evolution of systems.

In this paper, we have presented an overview of the reverse engineering of behavioral diagrams. Our approach use static analysis to identify all the possible behavior of the system and generate relevant input values that can be used by dynamic analysis. It also deals with the reverse engineering from execution traces for object-oriented software.

The approach proposed is fully automatic and uses a different methodology to deal with the problem of execution traces analysis. We start by a static analysis to define the all possible behaviors of the system. This guides the dynamic analysis. We used CPN as an intermediate model to analyze execution traces. In addition, our approach filter traces. This is very important in the case of GUI systems. It manages also to detect the "par" operator which is very important in the context of multi-threading system.

Our future work is to evaluate the approach proposed on complex systems. We will also address the problem of how to extract state diagrams which is an important part of the UML behavioral models.

REFERENCES

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

[2] OMG. Unified Modeling Language (OMG UML), Superstructure. V2.1.2. Nov.2007.

[3] Elliot j. Chikofsky and James H. Cross II. Reverse engineering and design recovery : a taxonomy. IEEE Software, 1990

[4] A. Rountev, O. Volgin, and Miriam Red-doch. Control flow analysis for reverse engineer-ing of sequence diagrams. Technical Report OSU-CISRC-3/04-TR12, Ohio State University, March 2004.

[5] Wahab Rouagat, Tewfik ZIADI "Towards a New Approach for Reverse Engineering of UML 2 Sequence Diagrams", STIC09,

[6] R. Delamare, B. Baudry, Y.L/ Traon "Reverse-engineering of UML 2.0 Sequence Diagrams from Execution Traces" Workshop on Object-Oriented Reengineering at ECOOP 06, Nantes, France, July 2006.

[7] T. Ziadi, M. Aurélio A. da Silva, L. Hillah, M. Ziane, "A Fully Dynamic Approach to the Reverse Engineering of UML Sequence Diagrams", 16th IEEE International Conference on Engineering of Complex

Computer Systems, 2011, pp. 107-116, (IEEE Computer Society).
[8] http://www.ptidej.net/material/inanutshell. August 2009.
[9] 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 (9) , 2006 , pp. 642-663.
[10] R. Zhao, and L. Lin, "An UML Statechart Diagram-Based MM-Path Generation Approach for Object-Oriented Integration Testing",Enformatika; 2006, Vol. 16, p259.
[11] D.H.A. van Zeeland, "Reverse-engineering state machine diagrams from legacy C-code", proceedings of 12th Conf. on Entity-Relationship Approach - Arlington-Dallas, Dec. 1993.
[12] A. Jakimi, L. Elbermi and M. El Koutbi, "Software Development for UML Scenarios: Design, fusion and code generation". International Review on Computers and Software, Vol. 6. n. 5, pp. 683-687, 2011.
[13] M. H. Abidi, A. Jakimi, E. H. El Kinani. A New Approach the Reverse Engineering UML State Machine from Java Code. International Conference on Intelligent Systems and Computer Vision (ISCV'2015), March 25-26 2015, Fes, Morocco.
[14] A.P.V. Vasconcelos, R.S.V. Cepêda, C.M.L. Werner, "An approach to program comprehension through reverse engineering of complementary software views", 1st Int. Workshop on Program Comprehension through Dynamic Analysis, Pittsburgh, PA, USA, 2005, pp. 58–62. Nov 2005.
[15] P. Tonella and A. Potrich, "Reverse Engineering of Object Oriented Code".Springer, New York, USA. 2005. ISBN: 0-387-40295-0.