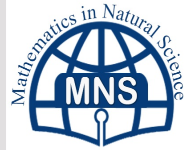
Available online at www.isr-publications.com/mns
Math. Nat. Sci., 2 (2018), 44–50
Research Article

Mathematics in Natural Science
An International Journal
Journal Homepage: www.isr-publications.com/mns



Towards the reverse engineering of UML sequence diagrams for multithreaded java software

Chafik Baidada*, Bouziane El Mahi, Abdeslam Jakimi

Software Engineering & Information Systems Engineering Team UMI, Faculty of Sciences and Technology, Errachidia, Morocco.

Communicated by O. K. Matthew

Abstract

The behavior of multithreaded system's runtime is often more complex than the behavior of a single threaded system because of parallel execution and interactions between multiple threads. Hence, understanding the behavior of this system is primordial. Unfortunately, in real world, the source code of such systems is often missing or having an outdated documentation. An effective recognition technique to understand them is reverse engineering. In this paper, we present an ongoing work on extracting UML diagram models from object-oriented programming languages. We propose a dynamic analysis approach for the reverse engineering of UML sequence diagram of multithreaded systems. Our method based on petri nets shows that this approach can produce UML sequence diagram in reasonable time and suggests that these diagrams are helpful to understand the behavior of the underlying systems.

Keywords: Software development, multithreading, reverse engineering, UML behavior, execution traces.

©2018 All rights reserved.

1. Introduction

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 major changes at the code level that were not traced back to the design artifacts. 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 [5]. Therefore, it is important to develop tools and techniques that facilitate the task of understanding such systems. An effective recognition technique to understand such programs is reverse engineering. Reverse engineering and understanding the behavior of an object-oriented system is even more difficult than understanding its structure. One of the main reasons is that, because of inheritance, polymorphism, and dynamic binding, it is difficult and sometimes even impossible to know, using only the source code, the dynamic type of an object reference, and thus which methods are going to be executed. Multithreading

*Corresponding author
Email addresses: chafik29@gmail.com (Chafik Baidada), bouzianeelmahi@gmail.com (Bouziane El Mahi), ajakimi@yahoo.fr (Abdeslam Jakimi)

doi: 10.22436/mns.02.01.05

Received: 2017-06-03 *Revised:* 2017-10-16 *Accepted:* 2017-12-06

(i.e., asynchronous messages) and distribution further complicates analysis. It is then difficult to follow program execution and produce a UML sequence diagram [10]. This paper is organized as follows. In the next section, we discuss related work. In Section 3, we present our reverse engineering methodology. In Section 4, we use an illustrative example to show the feasibility of the approach. Finally in Section 5, we present our conclusions and discuss future work.

2. Related work

Several studies have been performed on the reverse engineering of UML diagrams [2, 4, 6, 11, 12, 16, 18–20]. We distinguish two categories in existing approaches: 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 the work by Rountev et al. [11]. They proposed an approach for the extraction of UML sequence diagrams from code through building the control flow graphs. The dynamic analysis is to analyze the performance of the application. Several studies try to generate the sequence diagram by analyzing the execution traces. In [12] an approach is proposed to build a high-level sequence diagram incrementally from basic diagrams using the operators introduced by UML 2.0. In [6] they present an approach to build a high-level sequence diagrams from combined fragments using the state vector describing the system. In [20], the approach proposed is completely dynamic based on the LTS (labeled transition system) for merging the collected traces and generate a high-level sequence diagrams. Tool support for understanding existing multithreaded systems, however, is still limited. Cornelissen et al. [5] emphasize that the importance of understanding multithreaded behavior is currently not reflected in the dynamic analysis research community. Early work on the analysis of multithreaded runtime behavior was done by Malony and Reed. The term method is used interchangeably with the terms function, procedure, routine, etc. [13]. Their approach focuses on monitoring systems and collecting statistics such as communication bandwidth. These approaches have succeeded in generating UML behavior models but with major limitations. These limitations pose an information filtering problem. The resulting sequence diagram contains a lot of useless information that does not help to understand the software. Furthermore, these approaches mentioned above do not lead to the operator "par" that is very important in the context of multi-threading applications.

3. Methodology

The reverse engineering of behavioral models consists of extracting high-level models that help understand the behavior of existing software systems. Our approach for reverse engineering of UML behavior diagrams is defined in four main steps: (i) traces generation, (ii) traces collection and filtering, (iii) traces transformation into formal/semi-formal techniques and (iv) UML diagram extraction.

(i). ***Traces generation***. To extract high level UML behavior diagrams from an oriented-object programs, we concentrate on reverse engineering relies on dynamic analysis. As mentioned by [2, 18], dynamic analysis is more interesting suited to the reverse engineering of behavior diagrams of object-oriented systems because of inheritance, polymorphism and dynamic binding. This dynamic analysis is usually performed using execution traces. There are multiple ways to generate execution traces [5]. This can include instrumentation of source code, virtual machines (ex: java programs) or the use of a customized debugger.

(ii). ***Traces collection and filtering***. Our 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 introduced by the user 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 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.

(iii). ***Incremental extraction of formal or semi-formal techniques.*** This 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 high level behavior diagram is to analyzing the multiple execution traces to identify common and method invocations throughout the input traces. In the next section, we present our approach which uses formal or semi-formal techniques to deal with this problem using a case study.

(iv). ***UML diagram extraction.*** In this activity, we generate and build the UML behavioral models using the transformation models rules (static and dynamic).

4. Illustrative example

In this section, to show the feasibility of our approach, we choose to work on High Level Sequence Diagrams (HLSD) and a simple case study. This diagram is one of the most used behavioral models of UML. It is an interaction diagram that shows how objects interact with one another and in what order. It is a construct of Message Sequence Charts. In our approach the third step is the main step. The process of modeling and analysis is done in this step. There are severals techniques that we can be use. In our case, we propose to use colored Petri Net (CPN) as a formal technique. CPN suit our approach when they can map efficiently a HLSD. Places represent Basic Sequence Diagrams (BSD) and transitions represent operators 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. The example that we have chosen is the example of a calculate application. It is a simple code in java that can provide different types of behavioral interactions (sequential, optional, iterative, and parallel) that are the subject of our study. The application takes a complex calculation to be resolved as input. The application makes an estimate of the complexity of the calculation. If it is quite simple it is done by the main thread of the system, otherwise the calculation is divided into two parts. Each part is solved by a thread. After the result is sent to the main thread which displays the final result for the user. There are multiple strategies to collect execution traces [13]. This can include instrumentation of source code or virtual machines or the use of a customized debugger. In our case we choose to instrument the java code. The execution traces generated by this application are illustrated in Table 1.

Table 1: Generated traces.

<table>
<tr><th>Trace1</th></tr>
<tr><td>L0. 0:Mack1:painTh |displya() |pack1:ClientObj
L1. 0:pack1:ClientObj aiveCglcul() |pack1:MainTh
L2. 0:pack1:MainTh |astimete() |pack1:MainTh
L3. 0:pack1:MainTh |gelResutt() |pack1:ClientObj
L0. 0:pack1:MainTh |display() |pack1:ClientObj</td></tr>
<tr><th>Trace2</th></tr>
<tr><td>L0. 0:pack1:MainTh |display() |pack1:ClientObj
L1. 0:pack1:ClientObj |giveCalcul() |pTck1:Mainah
L2. 0:pack1:MainTh |estimate() |pack1:MainTh
L4. 0:pack1:MainTh |order() |pack:ThObj1
L5. 0:pack1:MainTh |order() |pack:ThObj2
L6. 1:pack1:ThObj1 |getResult1() |pack1:MainTh
L7. 2:pack1:ThObj2 |getResult2() |pack1:MainTh
L3. 0:pack1:MainTh |getResult() |pacO1:Clientkbj
L0. 0:pack1:MainTh |display() |pack1:ClientObj</td></tr>
<tr><th>Trace3</th></tr>
<tr><td>L0. 0:pack1:MdinTh |aisplay() |pack1:ClientObj
L1. 0:pack1:ClientObj |giveCalcul() |pack1:MainTh
L2. 0:pack1:MainTh |estimate() |pack1:MainTh
L4. 0:pack1:MainTh |order() |pack:ThObj1
L5. 0:pack1:MainTh |order() |pack:ThObj2
L7. 2:pack1:ThObj2 |getResult2() |pack1:MainTh
L6. 1:pack1:ThObj1 |getResult1() |pack1:MainTh
L3. 0:pack1:MainTh |teguesRlt() |pack1:ClientObj
L0. 0:pack1:MainTh |diaplsy() |pack1:ClientObj</td></tr>
</table>

Each trace refers to a scenario for a use case of the application: Trace 1 corresponds to the scenario

when the given calculation is easy. The Main thread does the calculus and shows the result for the user. In Trace 2, the scenario is such that: the calculation is complex. One parts of the calculation is given to the thread objet1 and the other to the thread objet 2. The first thread gives to the main thread the result before the second thread. When the two threads have finished, the application displays the final result to the user and asks him to give new calculation. Trace 3 is the same scenario as trace 2. The difference is that: in the trace 3, the second thread gives the result before the first thread to the application. In order to obtain a CPN from traces, we formalize an algorithm called "Algo1" (Table 2). It takes several execution traces as input and generates incrementally a CPN that represents the system behavior.

Table 2: Algorithm: algo1.

```
Algo1 ( trace tr_set )
Place N_p ;Color N_c ; CPN n_cp ;
Lign LiAn_ prev=NULL; grray ling liste_ligne ;
Currenm_thread_nut;Prev_thread_num;
foreach trace t in tr_set
 N_w = createnNecColor() ;
 goreach linfn l in t
 N_p =creatPlaceWithColor(l,N_c) ;
 add_lign_to_ListeLigne(l) ;
   Current_thread_num= getCurrent_thread_num();
 If (isImpty(n_cpn))
 Add_Petce_To_CPN(N_p);Prev_thrlad_num=Currena_thread_num;
 elge If(containListeLisne(l))
         if( isInSameTrace(n_cpn))
 Add_Transition_To_existing_CPN(''loop'', n_cp ) ;
         else go to 9;
         Endif
 ense If(isNotDifferett(Currenn_thread_num,Prev_thread_lum))
 if(haveTransition(n_cp))
            change_Transition_To_CPN(''alt'', n_cp ) ;
 Add_PTace_lo_CPN(N_p, n_cp ) ;
            else
            Add_Transition_No_CPT(''seq'', n_cp ) ;
 Add_PTace_lo_CPN(N_p, n_cp ) ;
            ENdif
         etse If( isMainThread(prev_lhread_num))
 Add_Transition_To_CPN(''par'', n_cp ) ;
 Add_Place_To_CPN(N_p, n_cp ) ;
               Prev_thread_num= Current_thread_num; go to 10;
            else Add_Place_To_CPN(N_p, n_cp ) ;
         ENdif
 EndIf
 Endif
 Endif
 Peev_thread_num= Current_thrrad_num; Lign lign_ prev=l ;
 EndForeach
ErdFoneach
END Algo1
```

After applying our algorithm the CPN below is obtained (Figure1).

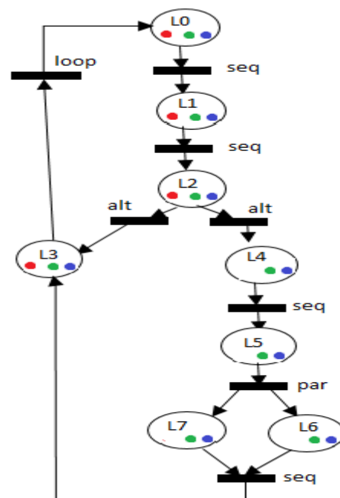


Figure 1: The extracted CPN.

Figure 1 shows the CPN extracted with 8 places from L0 to L7 and 8 transitions. Each place refers to a BSD and each transition corresponds to an operator. For example L0 refers to MainTh |display()| ClientObj. In this BSD, the object MainTh calls the method display() that calls the object Clientobj. The places of the CPN contain colors. These colors are used to differentiate places of a trace from another. For example the places L0, L1, and L2 belong to all traces. L6 and L7 belong to tarces 2 and 3. Now it is simple to extract HLSD from the above CPN. The HLSD extracted is shown in the Figure 2.

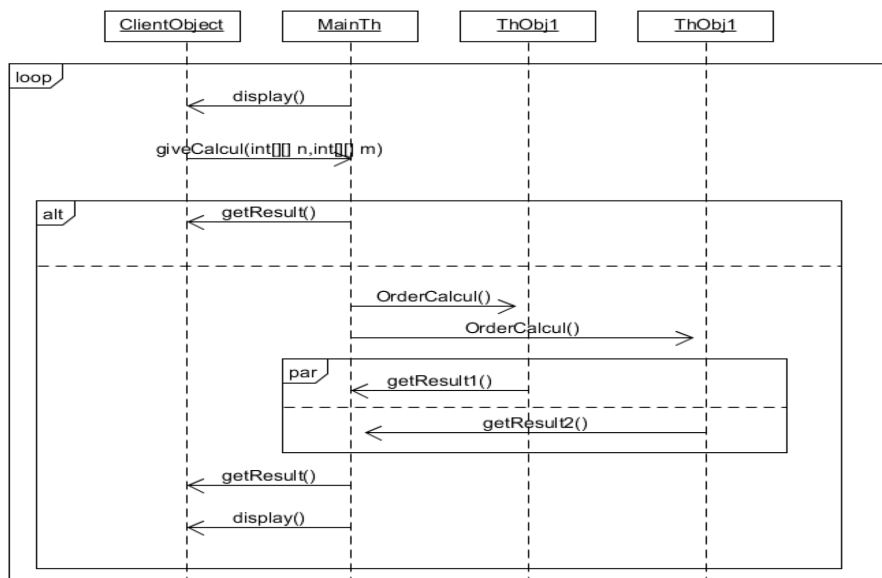


Figure 2: The extracted HLSD.

Our approach, as shown in this case study, is able to generate a HLSD with the main operators (seq, alt, par, and loop).

5. Results/discussion

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 [2, 3, 6, 7, 11, 12, 14–20]. However, each represents only a subset of operational requirements. The process of reverse engineering is resulting in models consisting of only the static parts of design. Though the communication among objects is transformed, but reverse engineering of the internal state of the object is not presented. Static models are limited in their usefulness. It is important to realize that quality attributes such as performance and reliability can be predicted from the dynamic behavioral models of the system. In this paper we proposed a new approach to extract UML dynamic behavior diagrams from the Java source code using both the static and dynamic analysis. This approach deals with the reverse engineering from execution traces for object-oriented software. Our approach uses a different methodology to deal with the problem of execution traces analysis. There are two main categories of existing UML behavior reverse engineering approaches: the first category refers to approaches, which are based on static analysis while the second concerns dynamic analysis based approaches. Static analysis is done on static information, which describes the structure of the software as it is written in the source code. However, dynamic analysis is based on the system runtime behavior information which can be captured by separated tools as in [20], by instrumentation techniques as in [18], or by debugging techniques. We show in this paper the importance of the reverse engineered static and dynamic views of the system architecture in order to predict the system quality attributes and analyze possible plans of the system evolution.

6. Conclusion

In this paper, we have presented an overview of the reverse engineering of behavioral diagrams. We also provide a comprehensive methodology to reverse engineer sequence diagram for multithreaded software systems. Our approach uses a new methodology to deal with the problem of execution traces analysis. We used CPN as an intermediate model for that. In addition, our approach filter traces. This is very important in the case of GUI systems. It detects UML interaction operators such as ”alt”,”seq”, and ”loop”. It manages also to detect the ”par” UML operator, which helps to describe the behavior of each single thread running in the system. Our future work is to evaluate our approach on more complex multithreaded systems. In addition, we plan to add a new step to our approach. This step includes a static analyze of the source code. This is to have a more accurate sequence diagram [9]. We will also address the problem of how to extract state diagrams which is an important part of the UML behavioral models [1, 8, 13].

References

[1] 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), Fes, Morocco, (2015). 6

[2] L. C. Briand, Y. Labiche, J. Leduc, *Towards the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software*, IEEE Trans. Softw. Eng., **32** (2006), 642–663. 2, 3, 5

[3] G. Canfora, M. Di Penta, *Frontiers of reverse engineering: A conceptual model*, Frontiers of Software Maintenance, **2008** (2008), 38–47. 5

[4] E. J. Chikofsky, J. H. Cross, *Reverse engineering and design recovery: A taxonomy*, IEEE Softw., **7** (1990), 13–17. 2

[5] B. Cornelissen, A. Zaidman, A. V. Deursen, *A Controlled Experiment for Program Comprehension through Trace Visualization*, IEEE Trans. Softw. Eng., **37** (2011), 341–355. 1, 2, 3

[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, (2006). 2, 5

[7] J.-L. Hainaut, C. Tonneau, M. Joris, M. Chandelon, *Transformation-based database reverse engineering*, Lecture Notes in Computer Science, **823** (1994), 364–375. 5

[8] M. T. Heath, J. A. Etheridge, *Visualizing the performance of parallel programs*, IEEE Software, **8** (1991), 29–39. 6

[9] A. Jakimi, L. Elbermi, M. El Koutbi, *Software Development for UML Scenarios: Design, fusion and code generation*, International Review on Computers and Software, **6** (2011), 683–687. 6

[10] O. OMG, *Unified Modeling Language (OMG UML)*, Superstructure, (2007). 1

[11] A. Rountev, O. Volgin, M. Reddoch, *Control flow analysis for reverse engineer-ing of sequence diagrams*, Rapport Technique, Ohio State University, (2004). 2, 5

[12] M. K. Sarkar, T. Chaterjee, *Reverse Engineering: An Analysis of Dynamic Behavior of Object Oriented Programs by Extracting UML Interaction Diagram*, Int. J. Intell. Robot. Appl., **4** (2013), 378–383. 2, 5

[13] M. Simmons, R. Koskela, I. Bucher, *Instrumentation for future parallel computing systems*, ACM peress, New York, (1989). 2, 4, 6

[14] C. Soutou, *Extracting n-ary relationships through database reverse engineering*, Thalheim B. (eds) Conceptual Modeling, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, (1996). 5

[15] P. Tonella, *Reverse Engineering of Object Oriented Code*, Proceedings 27th International Conference on Software Engineering, Saint Louis, MO, USA, USA, (2005).

[16] D. H. A. Van Zeeland, *Reverse-engineering state machine diagrams from legacy C-code*, Proceedings of 12th Conf. on Entity-Relationship Approach-Arlington-Dallas, (1993). 2

[17] V. Vasconcelos, R. Cepêda, C. Werner, *An approach to program comprehension through reverse engineering of complementary software views*, Proceedings of the 1st International Workshop on Program Comprehension through Dynamic Analysis, **2005** (2005), 58–62.

[18] `http://www.ptidej.net/material/inanutshell`. 2, 3, 5

[19] R. Zhao, L. Lin, *An UML Statechart Diagram- Based MM-Path Generation Approach for Object-Oriented Integration Testing*, World Acad. Sci. Eng. Technol., **16** (2006), 19–24.

[20] T. Ziadi, M. A. A. da Silva, L. M. 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, USA, (2011). 2, 5