A Fully Dynamic Approach to the Reverse Engineering of UML Sequence Diagrams

Tewfik Ziadi*, Marcos Aurélio Almeida da Silva*, Lom Messan Hillah*†, Mikal Ziane*‡
*UMR CNRS 7606, LIP6-MoVe
Université Pierre et Marie Curie, Paris, France
†Université Paris Ouest Nanterre La Défense, Nanterre, France
‡Université Paris Descartes, Paris, France
Email: Tewfik.Ziadi@lip6.fr, Marcos.Almeida@lip6.fr, Lom-Messan.Hillah@lip6.fr, Mikal.Ziane@lip6.fr

***Abstract*—The reverse engineering of behavioral models consists in extracting high-level models that help understand the behavior of existing software systems. In the context of reverse engineering of sequence diagrams, most approaches strongly depend on the static analysis and instrumentation of the source code to produce correct diagrams that take into account control flow structures such as alternative blocks ("if"s) and repeated blocks ("loop"s). This approach is not possible with systems for which no source code is available anymore (e.g. some legacy systems). In this paper, we propose an approach for the reverse engineering of sequence diagrams from the analysis of execution traces produced dynamically by an object-oriented application. Our approach is fully based on dynamic analysis and reuses the k-tail merging algorithm to produce a Labeled Transition System (LTS) that merges the collected traces. This LTS is then translated into a sequence diagram which contains alternatives and loops. A prototype of this approach has been tested with a real world application that has been developed independently from the present work. Our results show that this approach can produce sequence diagrams in reasonable time and suggest that these diagrams are helpful in understanding the behavior of the underlying application.**

***Keywords*-reverse engineering; UML sequence diagrams; execution traces**

I. Introduction

Scenario formalisms, such as UML Sequence Diagrams (SDs), play an important role in software engineering. They help software engineers understand existing software through the visualisation of interactions between its objects [1]. They can also be used in testing activities [2].

When sequence diagrams are either absent or inconsistent with the current code, as it is the case for many legacy systems, reverse engineering can be used to extract more accurate models. Reverse engineering can be done statically by analyzing the system's source code or dynamically by running the program and then analyzing the obtained execution traces to extract sequence diagrams. As underlined by [3], dynamic analysis is better suited to the reverse engineering of sequence diagrams of object-oriented (OO) systems because of inheritance, polymorphism and dynamic binding. Indeed, it is difficult to know the dynamic type of an object reference and which methods are executed by only relying on the source code.

In this paper, we consider the reverse engineering of sequence diagrams from execution traces of object-oriented systems. An execution trace of an object-oriented system is defined as a sequence of method invocations where each method invocation represents a communication between two objects. Figure 1 shows simple traces related to the well known ATM (Automatic Teller Machine) [4] example. While the mapping between method invocations in traces and messages in sequence diagrams is straightforward, two major challenges can be identified:

- *Control flow detection*. The first challenge concerns the detection of control structures in traces and their mapping to interaction operators in sequence diagrams. This mainly concerns the detection of the two main interaction operators included in UML sequence diagrams: alt and loop.
- *Multiple execution traces merging*. The second challenge concerns merging execution traces. Indeed, the behavior of a system is often described by multiple execution traces that correspond to different scenarios.

Several approaches have been proposed for the reverse engineering of sequence diagrams regarding these two challenges (e.g., [3], [5], [6]). In [3] a complete review of these approaches is discussed. However, existing approaches often use static analysis by instrumenting the source code to identify if method invocations in traces are related to loop blocks in the source code. Although this solution ensures that the detected loop conforms to the source code, it can not be reused in the context where the source code is not available. Another limit related to existing work on the reverse engineering of sequence diagrams concerns the second challenge. Indeed these approaches [5]–[7] only extract a sequence diagram from a single execution trace and it is not clear how they deal with multiple execution traces.

In this paper, we revisit the problem of reverse engineering sequence diagrams by a new approach which is completely based on dynamic analysis and which does not use any source code analysis. To tackle the two main challenges previously presented, our approach proposes to represent execution traces as Labeled Transition Systems (LTS). This enables us to define merging execution traces as merging

LTSes using well founded algorithms such as k-tail [8]. To extract a sequence diagram from the obtained merged LTS, we propose to identify sequence diagrams as regular expressions. This enables us to generate them from the obtained LTS and ensure that they are trace equivalent [9], [10].

In order to validate our approach, a prototype implementation was developed for Java systems. This prototype is able to observe a real Java system at runtime and to build a set of execution traces from it. From this set of execution traces it builds an LTS that merges the traces. From this LTS, it extracts a sequence diagram that depicts the interactions among the objects.

The paper is organized as follows. Section II introduces a background on reverse engineering of UML SD from execution traces and gives an overview of our approach. Section III deals with the extraction of LTS from execution traces. Section IV introduces our approach for sequence diagram extraction. Section V reports on first experiments using our prototype. Finally, Section VI discusses related work and Section VII concludes this paper.

II. Background and Approach

Our goal in this work is to extract UML sequence diagrams from multiple execution traces for an object-oriented system using only dynamic analysis. As underlined by [3], dynamic analysis is better suited to the reverse engineering of sequence diagrams of object-oriented systems because of inheritance, polymorphism and dynamic binding. Before presenting an overview of our approach, we present in this section models that we used to formalize the reverse engineering of UML sequence diagrams. First, we define execution traces for object-oriented systems. Then we formalize sequence diagrams as regular expressions on method invocations.

A. Execution Traces for Object-Oriented Systems

Performing dynamic analysis of a object-oriented system starts by collecting an *execution trace* from its execution. An execution trace is defined as a sequence of method invocations. In the following, we formally define method invocations and traces.

Definition 1 (Method invocation): A Method invocation is a triplet $\langle caller, method, callee\rangle$ where:

- $caller$ is the caller object, expressed in the form "$object : class$"
- $method$ is the invoked method of the callee object, expressed in the form "$methodName()$".[1]
- $callee$ is the callee object, expressed in the form "$object : class$"

A method invocation is displayed in the trace as: `label. caller | method | callee`. Labels are only used to simplify further references in the rest of the text.

[1]In this paper we do not consider method parameters in our invocations.

In what follows, we introduce a definition of equivalence of method invocations which is necessary to formalize traces merging.

Definition 2 (Equivalence between method invocations): The method invocations $inv1 = \langle caller1, method1, callee1\rangle$ and $inv2 = \langle caller2, method2, callee2\rangle$ are equivalent if and only if:

- The two objects $caller1$ and $caller2$ are equivalent. Two objects are equivalent if they are instances of the same class and are created (using the constructor invocation) with the same values of parameters.
- $method1$ and $method2$ concern the same method and have the same signature. [2]
- $callee1$ and $callee2$ are two equivalent objects.

To define equivalence between method invocations in our approach, we implemented in the *Traces Collection* step described below a component collecting all invocations of constructors in all execution traces. This allows us to check equivalence between objects using the rule defined in the definition above.

Definition 3 (Trace): A Trace is a sequence of method invocations $inv_1, inv_2, ..inv_n$.

Figure 1 illustrates an example of two execution traces for the well known ATM (Automatic Teller Machine) example. These traces show method invocations between four objects, instances of the classes: `UserIHM`, `ATM`, `Consortium`, and `Bank`. The first trace illustrates the execution of the ATM system where the user entered a bad password in two attempts and then decides to cancel the operation. The second trace shows a sequence of method invocations where the user cancels the operation after the password request. Note that these two traces also illustrate equivalence between method invocations as we defined above. Indeed, all method invocations labeled by the same label are equivalent events if they are related to different traces. For instance, all invocations labeled "inv3" in Trace 1 and Trace 2 are equivalent because the objects involved in these invocations are equivalent (they are created with the same values of parameters).

Note that in this paper we focus on sequential traces. If a concurrent system is considered, the traces collected from different threads are sequentially collected in a unique execution trace as described in [11]. In addition, in the presented example, we only presented synchronous invocations. However, our approach also supports asynchronous ones.

B. Sequence diagrams (SD)

A SD shows how a set of objects interact with each other. The diagrams considered in this paper follow the

[2]In this paper we do not consider method parameters in our invocations. However we only deal with parameters of constructors, i.e., the *new()* method for Java systems in the perspective of defining equivalence between objects.

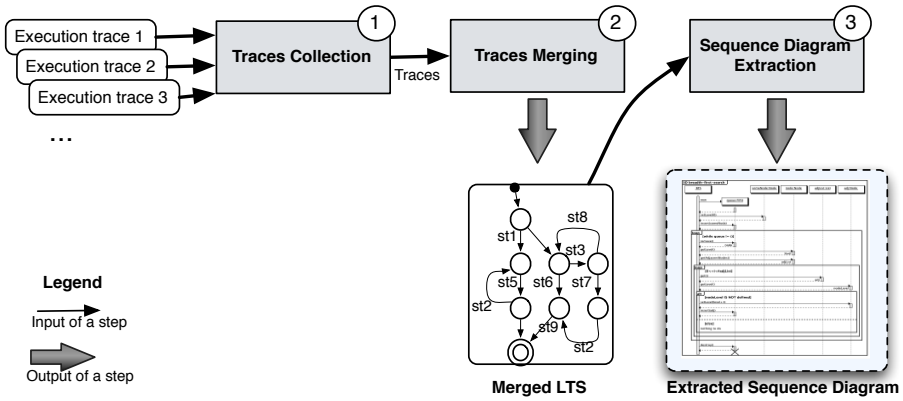

Figure 2. Overview of our approach.

Trace 1

inv1. atm:ATM | displayMainScreen() | user:UserIHM
inv2. user:UserIHM | insertCard() | atm:ATM
inv3. atm:ATM | requestPassword() | user:UserIHM
inv4. user:UserIHM | enterPassword() | atm:ATM
inv5. atm:ATM | verifyAccount() | cons:Consortium
inv6. cons:Consortium | verifyAccountWithBank() | bank:Bank
inv7. bank:Bank | badBankPassword() | cons:Consortium
inv8. cons:Consortium | badPassword() | atm:ATM
inv3. atm:ATM | requestPassword() | user:UserIHM
inv4. user:UserIHM | enterPassword() | atm:ATM
inv5. atm:ATM | verifyAccount() | cons:Consortium
inv6. cons:Consortium | verifyAccountWithBank() | bank:Bank
inv7. bank:Bank | badBankPassword() | cons:Consortium
inv8. cons:Consortium | badPassword() | atm:ATM
inv9. user:UserIHM | cancel() | atm:ATM
inv10. atm:ATM | cancelledMessage() | user:UserIHM
inv11. atm:ATM | ejectCard() | user:UserIHM
inv12. atm:ATM | requestTakeCard() | user:UserIHM

Trace 2

inv1. atm:ATM | displayMainScreen() | user:UserIHM
inv2. user:UserIHM | insertCard() | atm:ATM
inv3. atm:ATM | requestPassword() | user:UserIHM
inv9. user:UserIHM | cancel() | atm:ATM
inv10. atm:ATM | cancelledMessage() | user:UserIHM
inv11. atm:ATM | ejectCard() | user:UserIHM
inv12. atm:ATM | requestTakeCard() | user:UserIHM

Figure 1. Sample execution traces for the ATM example.

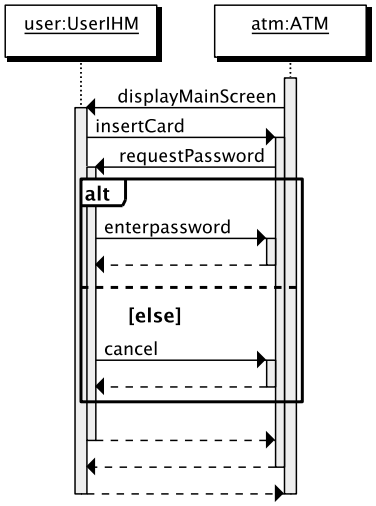

Figure 3. Sequence Diagram of checking account in the ATM example.

UML2 metamodel [12]. Figure 3 shows an example of a sequence diagram which describes the interactions of two objects, instances respectively of classes `UserIHM`, and `ATM`. The vertical lines represent lifelines for the given objects. Interactions between objects, displayed as horizontal arrows, are called messages in the UML2 specification. Each message corresponds to a method invocation. Messages located on the same lifeline are ordered from top to bottom.

Interactions in sequence diagrams can be composed using operators. UML2 considers several operators among which we only kept the main ones, which also allows us to identify sequence diagrams as regular expressions: `seq` (for sequential composition), `alt` (for alternative) and `loop` (for iteration). Figure 3 illustrates the use of the `alt` operator. Notice that a sequential composition can also be implicitly given by the relative order of two messages in a diagram. For example, in Figure 3, the message `displayMainScreen` is specified before the message `insertCard`. This is equivalent to a sequential composition between these two messages[3].

More formally, inspired by our precedent work [4], SDs can be defined as algebraic expressions where atomic terms are method invocations and operators are the three operators mentioned above.

Definition 4 (Sequence Diagram (SD)): A sequence diagram is an expression of the form:

$$D ::= M \mid (D \,\mathbf{alt}\, D) \mid (D \,\mathbf{seq}\, D) \mid \mathbf{loop}\,(D)$$

[3]Note that, this interpretation is correct because the **seq** operator specifies a weak sequential which is different from the strict sequential composition [12].

where M is a method invocation.

For instance, let us consider the sequence diagram in Figure 3. It can be represented by the following expression:

D = inv1 **seq** inv2 **seq** inv3 **seq** (inv4 **alt** inv5)

where the inv$i$s are defined as:

- inv1 = (inv1. atm:ATM:|displayMainScreen() | user:userIHM),
- inv2 = (inv2. user:UserIHM|insertCard() | atm:ATM),
- inv3 = (inv3. atm:ATM|requestPassword | user:UserIHM),
- inv4 = (inv4. user:UserIHM|enterPassword() | atm:ATM),
- inv5 = (inv5. user:UserIHM|cancel() | atm:ATM).

This definition of sequence diagrams is isomorphic to regular expressions (RE) as shown below and will be used in section IV. Indeed, the seq operator in SDs is equivalent to the classical *concatenation* operator in REs, alt is equivalent to *choice* operator, and loop is equivalent to the Kleene *star* operator in REs. The alphabet of the corresponding regular expression is the set of method invocations of the sequence diagram.

Definition 5 (Isomorphism between REs and SDs): The mapping from REs into SDs $[\cdot] : RE \mapsto SD$ is defined as follows:

- $[S] = S$, iff S is a method invocation
- $[(S_1 + S_2)] = ([S_1]\ \mathbf{alt}\ [S_2])$
- $[(S_1 \cdot S_2)] = ([S_1]\ \mathbf{seq}\ [S_2])$
- $[(S)*] = \mathbf{loop}\ [S]$

The set of traces associated with a sequence diagram is defined straightforwardly as the set of traces recognized by the corresponding regular expression. This morphism is also obviously an isomorphism as all the equations are symmetrical.

C. Overview of Our Approach

The proposed approach consists of three steps outlined in Figure 2: the collection of the execution traces from a running system, the generation of a LTS that represents the merge of the input execution traces and finally the extraction of the sequence diagram.

Step 1: Traces Collection: This step consists in observing the interaction of a set of known objects in various scenarios. For each scenario, an execution trace is captured by creating a method invocation for each method call from one object to another. There are multiple strategies to collect execution traces [13]. This can include instrumentation of virtual machines or the use of a customized debugger. In Section V, we present the strategy we used to collect execution traces for Java systems.

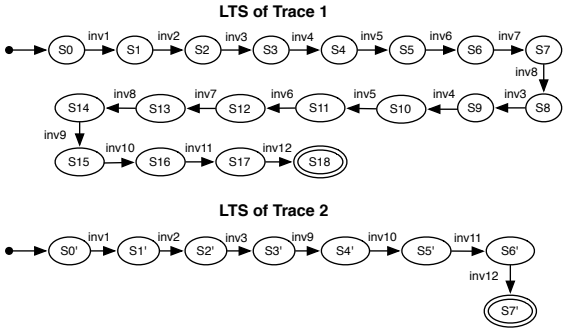

Figure 4. LTSes generated from traces of Figure 1.

Step 2: Traces Merging: In the second step of our approach we propose a technique, based on merging Labeled Transition Systems, to merge the traces collected in the previous step. This step is detailed in Section III.

Step 3: Sequence Diagram Extraction: This final step generates a sequence diagram using the results of Step 2. This step is detailed in Section IV.

III. Merging Traces

The second step of our approach deals with merging traces. Indeed, as mentioned in the previous section, one of the major challenges to reverse engineering sequence diagrams is to merge the multiple execution traces to identify common and variable method invocations throughout the input traces. To the best of our knowledge, most of existing work [5]–[7] only extracts a sequence diagram from a single execution trace and consequently this challenge of merging traces is not reported. For instance, *Briand et al.*'s approach only generates what the authors called partial sequence diagrams which depict the method invocations within a specific execution trace [3].

Independently from the reverse engineering of sequence diagrams, the challenge of merging traces is well identified in the grammar inference domain where several well defined techniques were proposed [14]. The main idea of these techniques is to represent each input trace using a labeled transition system and then define trace merging as LTS merging. However, the grammar inference techniques are often used to infer only LTSes that specify protocols of components. In this section we propose to reuse and adapt one of these grammar inference techniques (the k-tail algorithm [8]) to merge execution traces in the perspective of extracting sequence diagrams. This includes two steps: *Initialization* and *Merging*.

A. Initialization

In the first step, one LTS for each captured execution trace is generated. The LTS that we generate is a variant of classical finite automata where transitions are labeled by method invocations. Figure 4 illustrates two examples

of LTSes in which final states are represented by double circled states and the initial state is represented as a full small circle. Below we formally define our LTSes:

Definition 6 (LTS): An LTS is a 4-tuple $\langle S, T, s_0, s_F \rangle$, where S is a set of states, T is a finite set of transitions between states in S, s_0 is the initial state, and s_F is the set of final states. A transition $t \in T$ is a 3-tuple $\langle s, inv, s' \rangle$, where $s, s' \in S$ are the source and destination of the transition respectively, and inv is the method invocation labelling the transition.

This transformation from one execution trace to an LTS is straightforward. For each method invocation in the trace, a transition and a state are created in the LTS. The generated LTS will be a sequence of states, and will contain a single final state, which corresponds to the state reached when all method invocations in the trace have proceeded.

B. Merging

In this second step, the LTSes of the different traces are merged to obtain a single LTS that merges the initial traces. This is done by using the k-tail algorithm [8]. The algorithm starts by initializing a new LTS which has a new common initial state and merges the initial state of all input LTSes.

The k-tail algorithm takes as input the new initial LTS, then, iteratively merges "k-equivalent" states. Two states `s1` and `s2` are "k-equivalent" if and only if they are defined by the same set of paths of method invocations with length *k*. Before defining k-equivalence between states, we define the notion of k-paths.

Definition 7 (k-Paths): Given a state *s* in a LTS `M`, a set of paths with length *k*, called *k-paths(s)*, is defined as a set $\{path_1, ...path_r\}$, where $path_i$ is a sequence of method invocations in `M`, i.e., $path_i = st_1 st_2 ... st_k$ such that there exists a sequence of transitions $(s, inv_1, s_1)(s_1, inv_2, s_2)...(s_{k-1}, inv_k, s_k)$ in the LTS `M`.

The notion of *k-equivalence* between states is defined as follows:

Definition 8 (k-equivalence): Two states s1 and s2 in the LTS `M` are k-equivalent if and only if `k-paths(s1) = k-paths(s2)`.

For instance, the states `s14` in the LTS of Trace 1 and `s3'` in the LTS of Trace 2 (see Figure 4) are *2-equivalent* because:

```
2-paths(s14) = 2-paths(s3') = { inv9inv10 }
```

The k-tail algorithm iteratively identifies sets of k-equivalent states, i.e., states with the same `k-paths`, to be merged. Merging *k-equivalent* states `s1` and `s2` for example is realized by removing `s1` and adding all transitions entering or exiting `s1` to `s2`. The process of merging k-equivalent states is repeated until there are no more such states. The obtained LTS being an automaton, classical determinization and minimization techniques are applied to obtain a rigorous deterministic finite state machine.

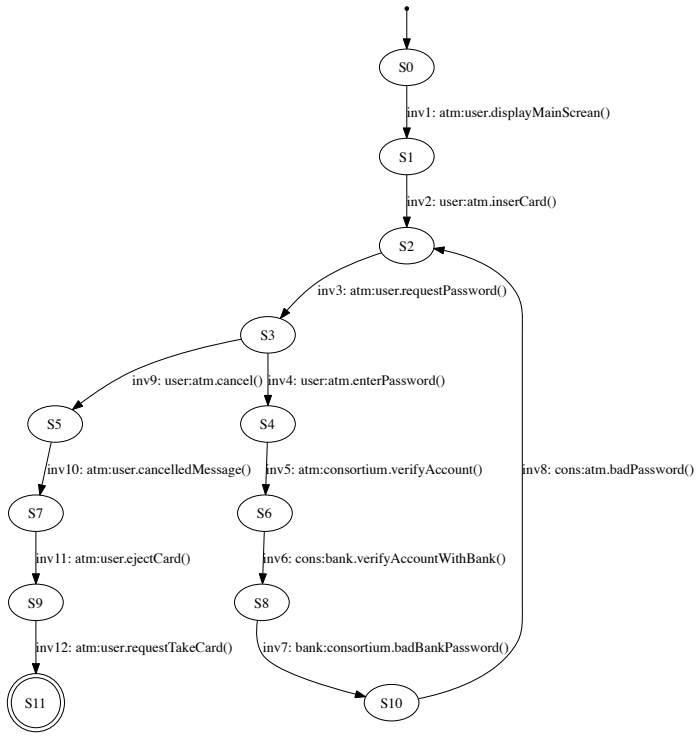

Figure 5. The extracted final LTS from the traces of Figure 1.

Figure 5 illustrates the LTS obtained from merging LTSes of Figure 4 using the k-tail with k = 2 and after minimization and determinization. Note that for the sake of clarity we labeled method invocations in transitions of the LTS of Figure 5 in the form `caller:callee.method`. For instance, `atm:user.displayMainScreen` is equivalent to the method invocation: `atm:ATM |displayMainScreen()| user:userIHM`.

The LTS obtained at the end of this step is thus an LTS that depicts the behavior specified in the input traces but allows other behaviors. The next step consists in extracting a SD from this LTS. Our approach for this extraction is presented in the next section.

IV. Sequence Diagrams Extraction

This section presents our approach to extract a SD from the LTS generated by the k-tail algorithm presented in the previous section. Our approach consists in reusing the

known solutions for the problem of *Converting Deterministic Finite Automata to Regular Expressions* to obtain a regular expression (RE) that is equivalent to the LTS [9], [10]. The obtained regular expression is then transformed into a SD by a simple mapping defined in Section II.

Neumann [10] presents a survey of existing techniques for converting finite automata to RE. We adopted the technique defined by Brzozowski in [15]. This technique is based on the creation of a system of equations of regular expressions with one variable for each state in the LTS. This system is then solved to the variable associated with initial state via straightforward substitution based on the Arden's theorem [16]. The solution for this system of equations is a regular expression. For instance, the regular expression obtained by this method over the LTS of Figure 5 is :

$RE_{LTS_{final}}$ = ((inv1·inv2·inv3·inv4·inv5·inv6·inv7·(inv8·inv3·inv4·inv5·inv6·inv7)*
inv8·inv3·inv9·inv10·inv11·inv12) +
(inv1·inv2·inv3·inv9·inv10·inv11·inv12))

where the **invi** are the labels of method invocations used in Figure 1.

The obtained RE can be mapped into a SD (see Definition 5).

The application of our mapping 5 on this regular expression results in the following sequence diagram whose expression is depicted as follows:

$SD_{LTS_{final}} = \left[RE_{LTS_{final}}\right]$ = **alt**(inv1 **seq** inv2 **seq** inv3 **seq** inv4 **seq** inv5 **seq** inv6 **seq** inv7 **seq loop** (inv8 **seq** inv3 **seq** inv4 **seq** inv5 **seq** inv6 **seq** inv7) **seq** inv8 **seq** inv3 **seq** inv9 **seq** inv10 **seq** inv11 **seq** inv12), (inv1 **seq** inv2 **seq** inv3 **seq** inv9 **seq** inv10 **seq** inv11 **seq** inv12))

The sequence diagram corresponding to the expression $SD_{LTS_{final}}$ is illustrated in Figure 6. Analyzing the obtained sequence diagram of Figure 6 shows that combining LTS merging and the mapping of LTS to regular expressions allows us to detect interaction operators. Indeed, in addition to the **seq** interaction operator, the sequence diagram of Figure 6 illustrates the detection of two other operators: **alt** and **loop**. The **loop** operator is extracted because the initial output traces contain a repetition of method invocations related to the situation where a bad password is entered by users.

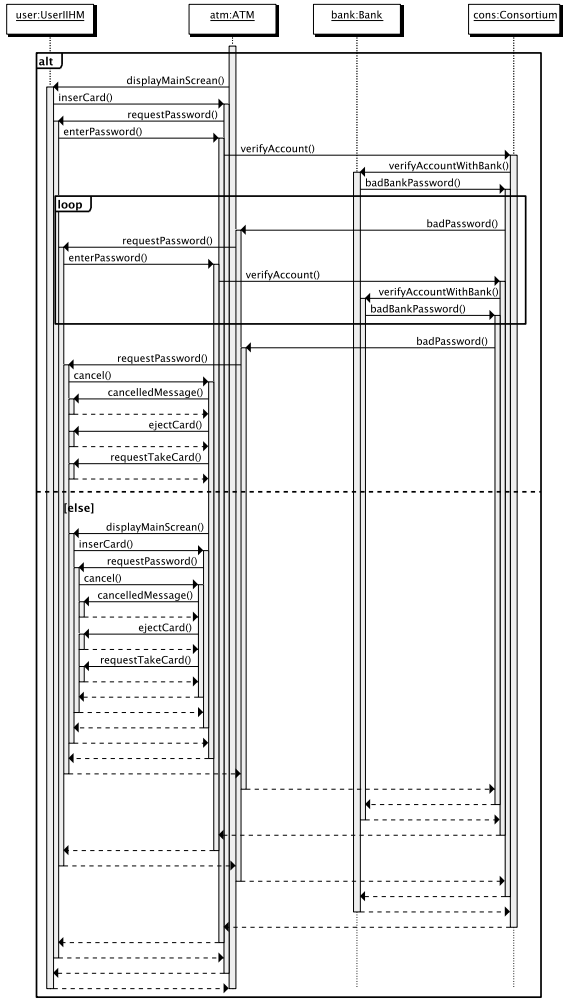

Figure 6. The extracted sequence diagram.

V. Preliminary Evaluation & Discussion

This section presents and discusses our preliminary evaluation of the approach presented in this paper. The two main hypotheses of this section are that (i) the presented approach is implementable as part of a state of the art programming environment; and (ii) it may be used to extract sequence diagrams from a real application.

In order to test the first hypothesis, a prototype implementation of the approach was developed as a chain of Model-to-Model (M2M) transformations based on the Eclipse Modeling Framework (EMF) environment. This prototype is able to observe a real system at runtime and to build a set of execution traces from it. From this set of execution traces it builds a state machine that abstracts the traces and from it a sequence diagram that depicts the interactions among the objects. The second hypothesis was tested by the means of a case study using a medium-sized scientific prototype of a distributed peer-to-peer modeling environment that has been developed independently from the present prototype. Our results show that we could successfully extract sequence diagrams that abstract the traces obtained by observation.

This section is organized as follows: Section V-A de-

scribes the prototype and details its architecture and Section V-B uses the prototype to build both behavioral models from the case study application.

A. Prototype Implementation

Our prototype implementation was developed in the Java programming language and contains 97 classes, with a total of 5.164 lines of code[4]. Approximately 50% of the code was automatically generated by Eclipse EMF.

The architecture of this prototype is organized into the 5 components described below.

- **Execution Trace Collector.** This component is responsible for observing the running system. To be able to perform this observation, this component uses a customized debugger to collect execution traces for Java systems. This component captures the execution traces and builds the related LTSes. The mapping from execution traces to LTSes was implemented as a M2M transformation.
- **LTS merger.** This component takes the LTSes generated by the previous component as input and merges them into a single LTS. This is done using the k-tail algorithm explained in Section III.
- **Determinizer & Minimizer.** We observed that the automaton generated by k-tail is non-deterministic (NFA). Therefore, in order to improve its readability, the present component uses the Graph library JFLAP 4.0 to determinize and minimize it. The output finite state machine (FSM) is the state machine generated by our approach.
- **Equation System Generator & Simplifier.** This component generates a system of equations of regular expressions from the FSM produced by the previous component. A M2M transformation generates an initial system of equations that is simplified until a solution is found. This implementation follows the method of Brzozowski [15].
- **Sequence Diagram Generator.** Finally, a transformation maps the regular expression that is the solution of the system generated by the last component into a sequence diagram. This mapping was described in Section IV. To visualise sequence diagrams, this component uses the Quick Sequence Diagram Editor (SDEDIT)[5].

B. Case Study

This section describes the case study that was run with the prototype implementation described in the previous section. A medium-sized research application that has been developed independently from the present prototype was chosen as the system to be observed. It features the Praxis Peer-To-Peer Collaboration Environment[6] which contains more than 500 classes and interfaces and approximately 25.000 lines of code.

This application was chosen because it contains a non-trivial set of components and interactions. It consists of a set of plug-ins to the Eclipse IDE that extend it with the possibility of peer-to-peer edition of EMF models. More specifically, this tool focuses on the management of the inconsistencies that may appear in such distributed environment. The inconsistency management is delegated to a SWI-Prolog inference engine. It is then responsible for keeping an up-to-date copy of the model currently being edited, and, under request from the Eclipse plug-ins, deliver an inconsistency report. This component, called SWIPrologBRidge, is therefore one of the core pieces of the Praxis Peer-To-Peer Collaboration Environment.

This component masks the complexity of the communication with the Prolog engine that runs in another virtual machine, and in another system process. That is why, even though the number of possible test cases being very reduced, the obtained sequence diagrams depict the interaction between many different objects.

We separated this case study into two parts. In the first one, we wanted to evaluate the performance of our approach. In order to do that, we defined 8 test cases and stress tested our approach with them. In the second part, we analyzed the obtained sequence diagrams to assess their utility in understanding the dynamic behavior of the application.

1) Performance Evaluation: 8 test cases were constructed in order to represent the most significative cases of utilization of this component. The corresponding execution traces were collected and the corresponding LTS and sequence diagram were constructed automatically by our prototype. The value of k for the k-tail was kept to 2 and the number of execution traces varied from 1 to 8 in each test case. The values of the following variables were recorded: time spent by each component, number of observed objects, number of messages in the sequence diagram and number of states and transitions in the LTS. We also recorded the number of the detected interaction operators. We particularly recorded the number of alternatives (alt) and loops in the obtained sequence diagrams. Figure 7 outlines the results of this experiment.

The time spent in each component was not divided equally. In average, 4% of the time was spent in merging the input LTSes; 81% was spent in the determinization and minimization step and 14% was spent in the equation simplification. The time spent in the transformation of the execution traces into the input LTSes and of the equations into the final sequence diagram were negligible. The time of generation of the behavioral models went from $0.65s$ in the

[4]This includes the JFLAP 4.0 Graph library that was used to minimize and determinize LTSes.

[5]http://sdedit.sourceforge.net/

[6]Available at *http://meta.lip6.fr/?page_id=17*.

Number of execution traces	1	2	3	4	5	6	7	8
Test case Running Time (ms)	202	265	558	457	508	578	672	669
Merging Time (ms)	15	62	105	401	459	528	461	501
Determinization & Minimization (ms)	631	1096	2213	4574	4905	8505	7499	16889
Equation System (ms)	3	512	483	781	725	1184	1911	1438
Objects	5	5	6	6	6	7	7	7
Messages in Sequence Diagram	22	71	103	1269	942	2561	6706	2437
Alternatives in Sequence Diagram	0	2	3	12	8	13	33	23
Loops in Sequence Diagram	1	6	9	156	116	335	836	276
States in LTS	17	18	18	21	21	25	24	30
Transitions in LTS	17	20	21	27	27	34	35	49

Figure 7. Results of our case study.

simplest case to $18.87s$ in the most complicated one. This shows that the approach is utilizable in the context of a real application.

2) Obtained Sequence Diagrams Evaluation: Figure 8 illustrates the Sequence Diagram obtained from one of the execution test cases. This test case requires the interaction between five objects:

1) The *PrologExecutionEngine* represents the interface of the Praxis environment with the SWIPrologBridge component.
2) The *Query* object encapsulates the code necessary to send queries to the Prolog process and get their result.
3) The *ProcessHolder* encapsulates the SWIProlog process that runs in another process in the operating system.
4) The *ReaderThread* encapsulates the raw communication with the Prolog inference engine thread, and makes sure that the data read through the Query object is valid.
5) The *WindowsValidator* encapsulates the validation of the strings returned from the Prolog inference engine.

This diagram represents a test case in which a PrologExecutionEngine is created and then a set of queries are sent to the SWIPrologBridge. Note that in the produced sequence digram, constructors are displayed as `CreateObject()` rather than the classical Java's `new()` invocation. This display is only imposed by the SDEDIT tool that we used to display SDs. Notice that the internal interaction between the objects cannot be deduced from the source code. This lies in the fact that the source code, in particular the Java source code, does not contain any information about the types of the objects used at runtime (like WindowsValidator, which would change in another operating system). The source code does not include a clear description of the interaction between different threads either.

When comparing the produced diagram with the source code of the original application[7], we notice that the obtained diagram correctly represents the set of traces obtained as input. However, there are some inconsistencies. For example, there is no call to $writePhrase()$ and $readPhrase()$ before the loop that continually sends queries to the Prolog engine.

This comes from the fact that our approach *generalizes* a set of extracted traces into a behavioral model. More specifically, this behavioral model is a sequence diagram that is an over-approximation of the input traces. Therefore, the completeness and correctness of the generated sequence diagram w.r.t. the system under analysis is hard to evaluate because it depends on the completeness of the input traces w.r.t. the same system. Even though these imperfections do not hinder the understandability of the diagram, an extension of the present work that is able to combine the present approach with a static analysis in order to rule out these imperfections is envisaged as future work.

VI. Related work

A complete survey of the reverse engineering of sequence diagrams is presented in [3]. In the same article, *Briand et al*. propose an approach to the reverse engineering of UML sequence diagrams for distributed Java applications. While our approach generates a sequence diagram by generalizing a collection of execution traces using the k-tail algorithm, their approach only considers one execution trace. Additionally, they discover interaction operators using source code analysis and manual instrumentation, which is a drawback when the source code is not available.

Different techniques for the reverse engineering of sequence diagrams have been published [5]–[7]. Unlike our work, however, all consider one execution trace without trying to generalize from it and do not support the identification of UML interaction operators (alt, loop,...).

Many approaches for protocol state machine extraction reuse the k-tail algorithm [11], [17]–[20]. However, the main difference between their use of this algorithm and ours is in the nature of the input traces and consequently in the obtained LTSes. Indeed, while our LTSes used as input of the k-tail show the interaction among the system's objects, the used models in [17]–[19] only show the behavior

[7]The original source code of the application is not presented here for the sake of brevity.

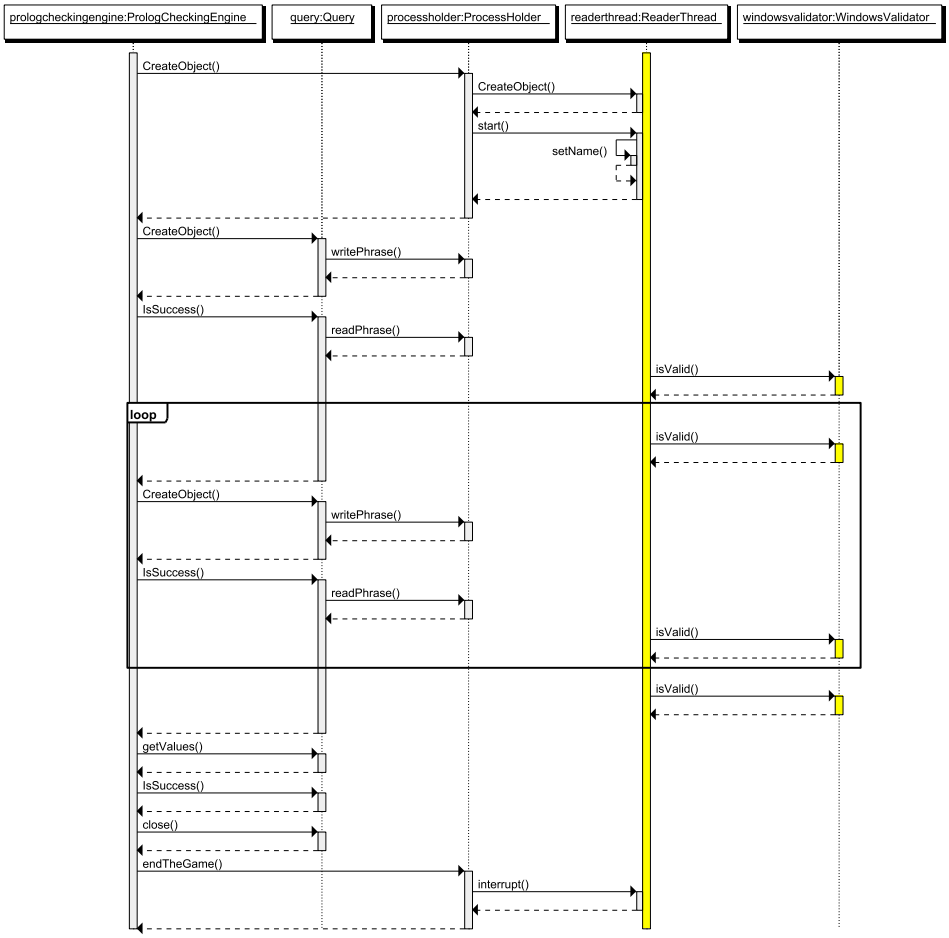

Figure 8. The extracted Sequence Diagram for the Praxis Peer-To-Peer Collaboration Environment.

of a single component through its interaction with a user via its graphical interface. We consider that our present results constitute a further validation of the k-tail family of algorithms in the context of reverse engineering of sequence diagrams.

As underlined by Lo & Khoo in [21], LTS merging algorithms such as k-tail can generate imprecise and over-generalized models. This produces many false negative or false positive behaviors. In the future, we plan to investigate the use of the new variants of the k-tail algorithm that avoid such problems of imprecision. We plan in particular to investigate the use of the algorithms of [11], [20]. Notice that the integration of these new algorithms will only modify the *Traces Merging* step in our approach.

VII. Conclusion and Future Work

This paper proposes an approach to the reverse engineering of sequence diagrams from execution traces for object-oriented software. Our approach is completely based on a dynamic analysis. That is a very important point since in some contexts, such as in legacy or in highly secured systems, source code may not be available and some analyses should be carried out anyway without having the source code.

Our approach reuses the well-known k-tail algorithm to extract a generalized labeled transition system from a set of observed execution traces. This LTS is then translated into a sequence diagram by an intermediary mapping into regular expressions. Notice that even though the k-tail algorithm is well known and used for LTS inference, to the best of our knowledge, no other work uses it to extract sequence diagrams.

Beyond offering a systematic and semantically well founded method, our approach 1) detects interaction operators in sequence diagrams, and 2) merges multiple execution traces to obtain a single sequence diagram.

A prototype implementation of our approach was developed as a chain of Model-to-Model (M2M) transformations using the Eclipse EMF Environment. Notice that we used a customized debugger to easily collect execution traces, but that we do not depend on it directly. The same traces could be obtained by other strategies such as instrumenting the

virtual machine, for example. This prototype was validated on a case study using a real application that has been developed independently from the present work. Our results show that the approach performs reasonably when it comes to analyzing real code and that the generated diagrams (although not perfect, because they are obtained from input traces without seeing the code) are helpful in the allowing an engineer understand the behavior of the system.

Our future work spans three main directions: 1) improving our generalization power by integrating other algorithms such as [11], [20], [22]. Notice that this only affects the second step in the approach; 2) considering other UML interaction operators such as the *par* operator [12] which is mandatory to support multi-threaded systems; 3) evaluating our approach on bigger object-oriented systems to assess its scalability and usability in more complicated cases. In addition, we aim at evaluating the definition of equivalence of objects that were used to implement our merging algorithm. For instance, we considered that two objects are equivalent if they are instances of the same class and were created with the same values, but this definition should be evaluated on larger case studies.

ACKNOWLEDGMENT

This work was partly funded by the ANR Project Proteus.

REFERENCES

[1] 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. Evol.*, vol. 20, no. 4, pp. 291–315, 2008.

[2] S. Pickin, C. Jard, T. Jéron, J.-M. Jézéquel, and Y. Le Traon, "Test synthesis from UML models of distributed software," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 252–268, Apr. 2007. [Online]. Available: http://www.irisa.fr/triskell/publis/2007/Pickin07a.pdf

[3] L. C. Briand, Y. Labiche, and J. Leduc, "Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software," *IEEE Tran. on Sof. Eng.*, vol. 32, no. 9, pp. 642–663, 2006.

[4] T. Ziadi, L. Helouet, and J.-M. Jezequel, "Revisiting statechart synthesis with an algebraic approach," in *ICSE 04*, ser. ACM, Edinburgh, UK, May 2004, pp. 242–251.

[5] O. Rainer and S. Thomas, "Javavis: Automatic program visualization with object and sequence diagrams using the java debug interface (JDI)," in *Revised Lectures on Software Visualization*. London, UK: Springer-Verlag, 2002, pp. 176–190.

[6] D. Lo, S. Maoz, and S.-C. Khoo, "Mining modal scenario-based specifications from execution traces of reactive systems," in *ASE 02*, 2007, pp. 465–468.

[7] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue, "Extracting Sequence Diagram from execution trace of Java Program," in 8^{th} *International Workshop on Principles of Software Evolution (IWPSE'05)*. IEEE Computer Society, 2005.

[8] 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.

[9] S. Kleene, "Representation of events in nerve nets and finite automata," *Ann. of Math. Studies*, vol. 34, pp. 3–40, 1956.

[10] C. Neumann, "Converting deterministic finite automata to regular expressions," 2005.

[11] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic Generation of Software Behavioral Models," in *Int. Conf. on Sof. Eng., ICSE'08, Leipzig, Germany*, 2008.

[12] OMG, *Unified Modeling Language: Superstructure - Version 2.3 formal/2010-05-05*, OMG, 2010. [Online]. Available: http://www.uml.org/

[13] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Trans. Software Eng.*, vol. 35, no. 5, pp. 684–702, 2009.

[14] J. E. Cook and A. L. Wolf, "Discovering models of software processes from event-based data," *ACM Transactions on Software Engineering and Methodology*, vol. 7, pp. 215–249, 1998.

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

[16] D. N. Arden, "Delayed-logic and finite-state machines," in *SWCT 1961)*. Washington, DC, USA: IEEE CS, 1961, pp. 133–151.

[17] S. P. Reiss and M. Renieris, "Encoding program executions," in *ICSE 2001*, 2001, pp. 221–230.

[18] J. Whaley, M. C. Martin, and M. S. Lam, "Automatic extraction of object-oriented component interfaces," in *ISSTA 02*, 2002, pp. 218–228.

[19] T. Xie, "Software component protocol inference," Univ. of Washington Dep.of Comp.Sc.and Eng., Seattle, WA, General Examination Report, June 2003. [Online]. Available: http://www.csc.ncsu.edu/faculty/xie/publications/generals-tao.pdf

[20] D. Lo, L. Mariani, and M.Pezzè, "Automatic steering of behavioral model inference," in *ESEC/SIGSOFT FSE*, 2009, pp. 345–354.

[21] D. Lo and S.-C. Khoo, "Quark: Empirical assessment of automaton-based specification miners," in *WCRE '06*. Washington, DC, USA: IEEE CS, 2006, pp. 51–60.

[22] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S., "Reverse engineering state machines by interactive grammar inference," in *WCRE 2007*. IEEE Computer Society, 2007, pp. 209–218.