Reverse-engineering of UML 2.0 Sequence Diagrams from Execution Traces

Romain Delamare, Benoit Baudry
IRISA / INRIA Rennes
Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 Rennes Cedex – France
{romain.delamare,benoit.baudry}@irisa.fr

Yves Le Traon
France Télécom R&D
MAPS / EXA
2 avenue Pierre Marzin
22307 Lannion Cedex – France
yves.letraon@francetelecom.com

***Abstract*— To fully understand the behavior of a program, it is crucial to have efficient techniques to reverse dynamic views of the program. In this paper, we focus on the reverse engineering of UML 2.0 sequence diagrams showing loops and alternatives from execution traces. To build these complete sequence diagrams, we need to capture the systems state through dynamic analysis. We propose to build state vectors through trace analysis and we precisely discuss how the state of an object-oriented system can be captured. We also present an adaptable trace analysis tool that we have developed to experiment the ideas presented in this work.**

I. Introduction

SOFTWARE maintenance is a major concern in the industry as software are getting more and more complex and bigger. Due to high level languages and mechanism such as inheritance or polymorphism it is very difficult to understand the behavior of a program just by reading its source code. Moreover the constant evolution of softwares and technologies often makes documentations obsolete and difficult to maintain.

Reverse-engineering can help maintainers understanding a complex system by retrieving models and documentation from a program. This is the process defined as redocumentation in [1]. UML is a good target language for the reverse engineering of models since it is largely used and offers different diagrams.

The reverse-engineering of UML static diagrams – like class diagrams – has been studied and is now implemented in several tools. Static views of the system allow engineers to understand its structure but it does not show the behavior of the software. To fully understand its behavior, dynamic models are needed, such as sequence diagrams or statecharts.

But for now, little work has been done on the reverse-engineering of UML dynamic diagrams. Our work is inspired by [2]. We focus on trace analysis for the reverse engineering of sequence diagrams.

Simple sequence diagrams can be retrieved by a simple analysis of the messages exchanged between the objects. We concentrate on the generation of UML 2.0 sequence diagrams which model loops and alternatives. To retrieve those informations we need informations about the state of the system during the trace analysis.

The contribution of this paper consists in proposing an approach to capture the program state during the trace analysis in order to precisely reverse engineer sequence diagrams. We also present and adaptable tool for trace analysis.

The rest of the paper is organised as follows: Section II introduces our approach; Section III explains how to catch the program state; Section IV discusses the tracing methods that can be used to retrieve the needed informations; Finaly Sect. V concludes the discussion.

II. Reversing Sequence Diagrams

OUR WORK takes place in the process described in [2] and focuses on the two first steps of the process. The first step consists in tracing the program to produce basic sequence diagrams and the second step consists in combining the sequence diagrams obtained in the first step to obtain high level sequence diagrams.

High level sequence diagrams are opposed to basic (or simple) sequence diagrams. High level sequence diagrams describe a more complex behavior by using the UML 2.0 fragments. Those fragments show alternatives (`alt` fragment) or loops (`loop` fragement). Basic sequence diagrams do not show those kind of behavior.

To obtain high level sequence diagrams we annote basic sequence diagrams with informations on the state of the program before and after each message. We then use this state informations to combine the sequence diagrams and obtain a high level sequence diagram with the combination operators from UML 2.0. This method is our main contribution and it is detailed in the rest of the section.

Although generating basic sequence diagrams is easy, generating high level sequence diagrams is a complex task. A basic sequence diagram can be directly extracted from an execution trace. Figure 1 shows an example of Java code with two classes. If an instance of *A* executes the method *m1*, then it calls *m2* on an instance of *B*; so the sequence diagram corresponding to that execution must have a message *m2* from the instance of *A* to the instance of *B*, as shown in

```
public class A {
    B b;

    public void m1() {
        b.m2();
    }
}

public class B {
    public void m2() {
        ...
    }
}
```

Fig. 1. An example of Java code

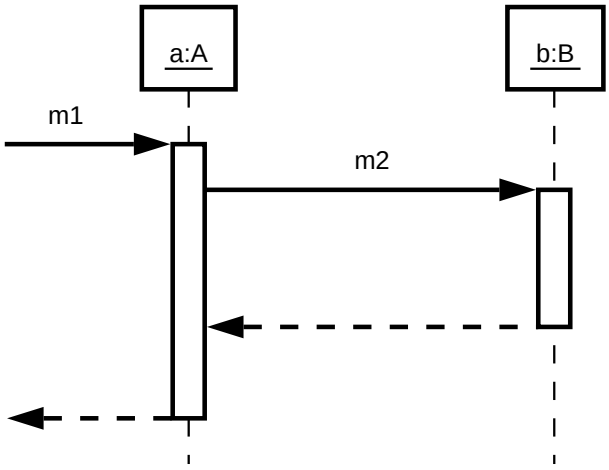

Fig. 2. A sequence diagram obtained by tracing the example from Fig. 1

Fig. 2. On the other hand, composing basic sequence diagrams to generate a higher level sequence diagram is a difficult task : the detection of iterations or optional messages is not obvious, even with many basic sequence diagrams.

We propose to use state vectors for the generation of high level sequence diagrams. A state vector is a vector of variables that characterizes the system state. State vectors are useful to detect iteration or conditional message.

State vectors can be extracted at runtime before and after each message. Then we can combine several basic sequence diagrams. With state vectors we can detect when two different sequence diagrams are in the same state, or we can detect when there is an iteration on a single sequence diagram. In Sect. III we detail how to catch the system state in a state vector.

Capturing the program state is not simple and we propose a precise analysis ot the state notion in Sect. III.

A sequence diagram can be annoted with state vectors before and after each message. Then this sequence diagram can be transformed to show loops or alternatives. For instance, Fig. 3 shows a sequence diagram annoted with state vectors. The sequence of the message *m1* and *m2* can be iterated as the vector state before the sending of *m1* and the state vector after the sending of *m2* are the same. The message *m1* can also be iterated as it does not change the state of the system.

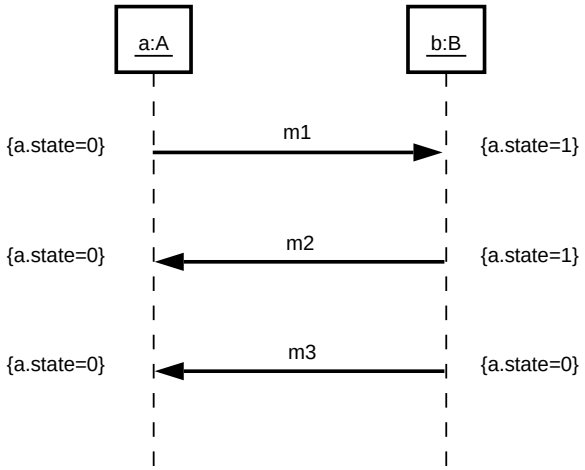

Fig. 3. A sequence diagram annoted with state vectors.

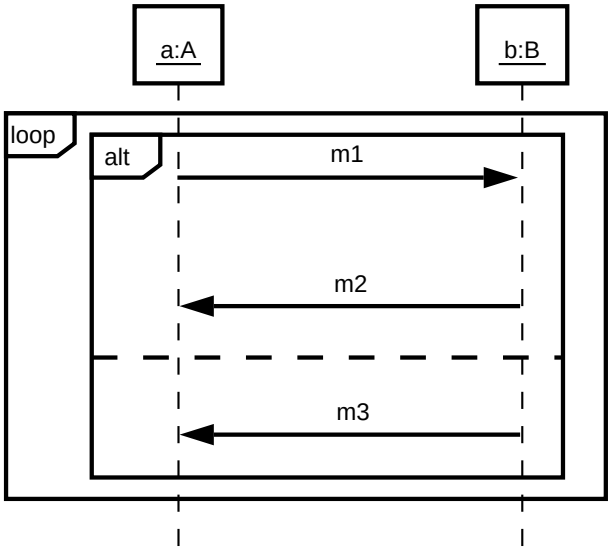

Fig. 4. A sequence diagram obtained by transforming the diagram of Fig. 3.

Figure 4 shows a transformation of the sequence diagram of Fig. 3 with the UML 2.0 `loop` and `alt` fragments.

Different sequence diagrams can be combined to obtain a more detailed sequence diagram. For instance, Fig. 5 is the result of a second trace analysis of the program traced in Fig. 3. We can combine those two annoted sequence diagrams to obtain the sequence diagram from Fig. 6. The combination of sequence diagrams, as well as the detection of loops and alternatives, can be automated.

Although our method has similarities with the work of [3], they are very different. Both methods rely on state vectors but in [3] the goal is to transform sequence diagrams into statecharts. Moreover in [3] the authors deduce the state vectors from preconditions and post-conditions on the message whereas we extract the state vectors from trace executions.

In the next section we define the state vectors we want to

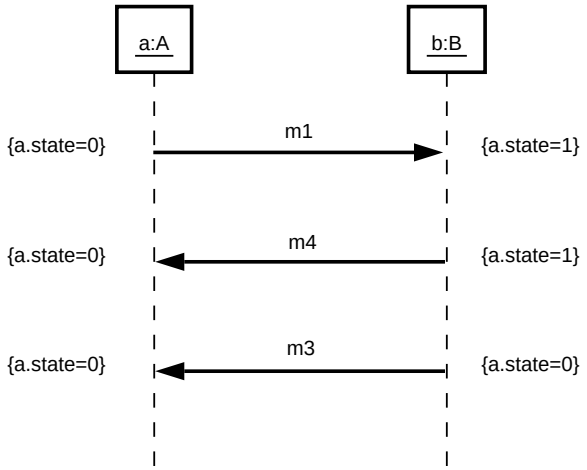

Fig. 5. An other annoted sequence diagram.

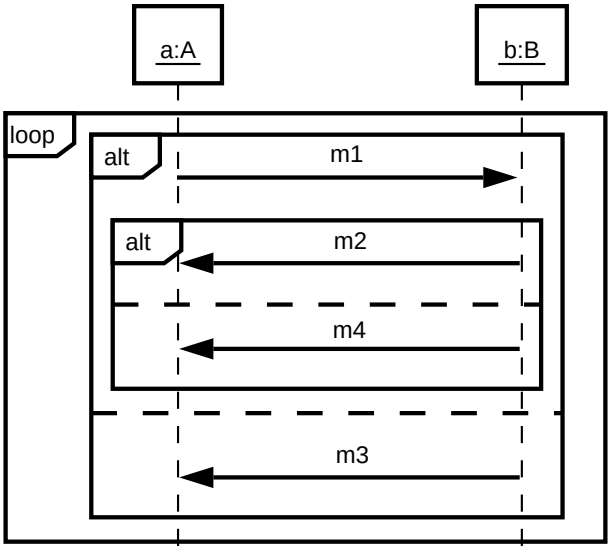

Fig. 6. A sequence diagram obtained by combining the diagrams from Fig. 3 and 5.

capture and how to capture them.

III. Capturing the Program State through Trace Analysis

CAPTURING the state of a program is an essential part of our approach as it is based on state vectors. Therefore we must have precise and consistent informations about the state of the program at several points of the program. We propose to catch the program state during the execution before and after each message. We also propose a methodology to extract the desired sequence diagrams.

State Vector: A state vector is a vector of values of attributes of the system that characterizes the state of the program. This implies that the state of the program can be characterized by a set of attributes. We can reasonably suppose that if a statechart can describe the behaviour of the system, then a set of attributes that characterizes the program can be found.

```
public class Boiler {
    private int temperature;

    public static void main(String[] args) {
        ...

        if (temperature > 180)
            ...
        if (temperature < 50)
            ...
        else
            ...

        ...
    }
}
```

Fig. 7. An example of a Java class with a state characterized by a large domain attribute.

The attributes of the state vector can be of several different types, and they must be considered differently. Mainly we can divide the different attribute types into three groups : the primitive values on a small domain, the primitive values on a large domain and the values that refer to an object.

A. Small domain attributes

The attributes with a primitive value on a small domain are essentially boolean, character or enumeration attributes . They all have a few different values. Boolean and enumeration attributes often characterize the state of the program, but character attributes or rarely used for the state of the program.

Small domain attributes are easy to capture. Comparing their values is enough to distinguish two states. So the trace must only store the value of those attributes at each considered event of the execution.

B. Large domain attributes

The attributes with a primitive value on a large domain, like integers or floats, are very difficult to catch. Mainly two cases can occur :

- the attribute takes only a few different values
- the attribute takes its values on a very large part of the domain

If they take only a few different values then they can be considered as primitive values on short domains : comparing their values is enough to distinguish two states.

But if they take a lot of different values the state is harder to catch. Instead of being characterized by the exact value of the attributes – like short domain attributes –, the state of the program can be characterized by the intervals containing the values of the attributes.

For instance, Fig. 7 shows a Java class representing a boiler. An instance of the class *Boiler* can be in three different states :

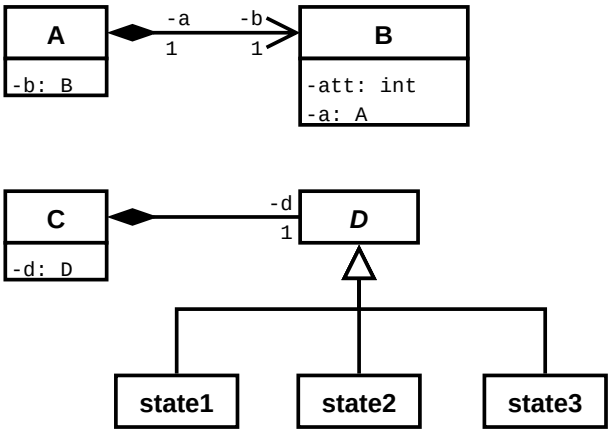

Fig. 8. A UML class diagram. Although the class *A* and *C* both have an attribute referencing an object, they must be catched differently.

when the temperature is greater than 180 degrees, when the temperature is between 50 and 180 degrees and when the temperature is less than 50. Distinguishing the state is difficult : the temperature can have two different values in the same state.

To solve this problem we must know the intervals of interest in regard of the state. Either those intervals must be specified by the user, either we must infer them.

C. Object reference attributes

Attributes referencing an object can describe the state vector in two different ways :

- the state of the program is characterized by the attributes of the referenced object
- the state of the program is characterized by the real type of the referenced object

If the state of the program is characterized by the attributes of the referenced object, we must capture the values of those attributes. For example in Fig. 8, the class *A* has an attribute *b* of type *B*. The state of the program is characterized by the attributes of *B*, so those attributes must be captured. To avoid cross-reference problems and to keep the state vector simple we suggest to capture only the primitive attributes of the object. Figure 8 shows an example of cross-reference : an instance of *A* and an instance of *B* can reference each other.

The state of the program can be characterized by the real type of the object. This is often the case when using specific design patterns like the state pattern. In this pattern the state of an object is characterized by the real type of an attribute declared with an abstract type. For instance in Fig. 8 the class *C* has an object reference attribute declared with type D. The class D is an abstract class extended by three different classes : the state of *C* is characterized by the real type of its attribute *d*.

To handle the case of state characterized by the real type of an object, we propose to extract the name of the real type as the relevant information for the attribute declared with an abstract type. The state pattern can be handled easily like this.

D. Methodology

In order to produce relevant sequence diagrams the state vector must be precise and easy to compare with other state vectors. A state vector with all the attributes of all objects of the program does not characterize the state of the program properly. So the state vector must only contain the object attributes that characterize the state of the program.

To extract relevant state vectors we propose an interactive approach and detail a methodology. This method is based on a sequence of exchanges between the user and the tracing tool.

The user specifies which attributes must be part of the state vector. This cannot be easily done with little knowledge of the system, but we propose a methodology to avoid this problem.

Our methodology consists in several interactions between the user and the tracing tool. At first the user will likely not know which attributes to choose. She can either try to guess the relevant attributes or select all the attributes. The first result has little chance to be meaningful. Then the user must try and remove some attributes from the state vector. If the result is worse then the removed attribute should be put back in the vector state, otherwise the user must continue and try to remove other attributes. This method will quickly converge to a minimal state vector where all the attributes characterizing the state are present, and only those attributes.

IV. Tracing Object Oriented Programs

THE DIFFICULT part of that method is to determine whether a result is better than an other or not. First a result showing a new alternative or iteration is better because it describes a more precise behavior. Also if after removing an attribute from the vector state the result has not changed, it is likely that the attribute does not characterize the state of the program.

In the worst case the result is a basic sequence diagram without the UML 2.0 fragments. If the user cannot find a correct state vector – either because of the difficulty of the task or the lack of correct state vector – the method neither improve nor worsen the sequence diagram. This means that if a result differs from the basic sequence diagram, then this result is better.

A second contribution of our work is to develop an adaptable tool for execution trace analysis. This tool is based on the use of a debugging tool called JTracor that is developped at IRISA. It traces Java program and was initially used for code coverage [4] and fault localization [5] purposes.

We have extended JTracor for reverse-engineering. Figure 9 shows the UML class diagram of JTracor. It relies on the Java Debug Interface (JDI), an API provided by Sun Microsystems. The class *TraceProvider* executes the program and at each event (e.g. method entry or exit, exception) it calls a method on its listeners. The listeners are instance of a class implementing the interface *Trace*. The trace produced depends on the implementation of *Trace*. For example *CodeCoverageTrace*

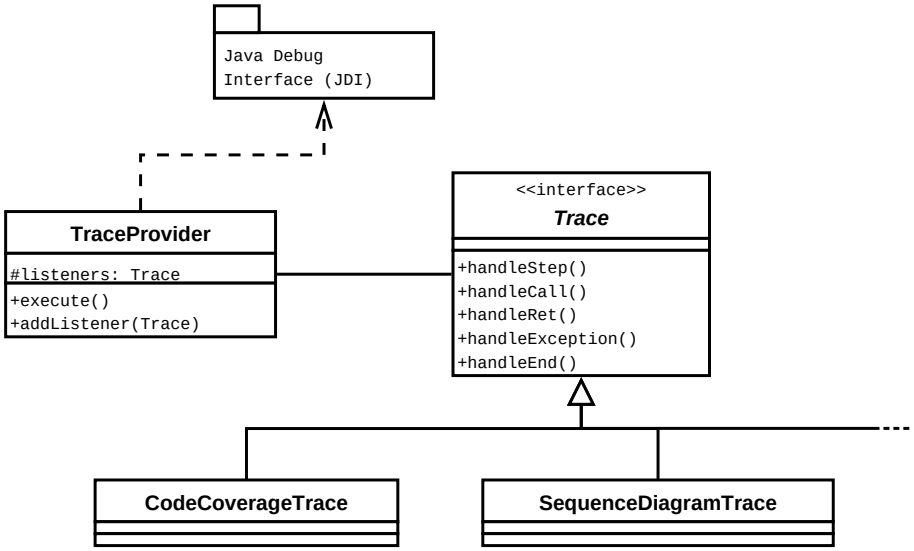

Fig. 9. UML class diagram of JTracor

produces code coverage statistics. This allows JTracor to be used in many different cases.

The *SequenceDiagramTrace* class generates sequence diagrams with informations on state vectors. The user must specify the classes and the method to appear in the sequence diagram. She must also specify for each class the attributes whose value must be catched in the state vectors. At each method entry, a message is created in the sequence diagram. The values of the watched attributes of the source object and the class object are read and put in the state vector. When the method returns, the attributes of the target are read again. Exceptions are handled too, because method exit may occur when an exception is thrown.

V. Conclusion

OUR WORK consists in generating UML 2.0 sequence diagrams from execution traces, as described in [2]. We propose a method based on state vectors that allows loops and alternatives detection within a single sequence diagrams and combination of several sequence diagrams. The state vectors are retrieved at runtime and we present a methodology to obtain meaningful sequence diagrams by finding a precise state vector.

For trace analysis we use a tool developed at IRISA, JTracor, and we are currently experimenting with several different situations. Future works include more experimentations and some case-studies to confirm our methodology and our hypothesis.

References

[1] E. J. Chikofsky and J. H. Cross II, "Reverse engineering and design recovery: a taxonomy," *IEEE Software*, 1990.

[2] Y.-G. Guéhéneuc and T. Ziadi, "Automated reverse-engineering of UML 2.0 dynamic models," in *proceedings of the 6^{th} ECOOP Workshop on Object-Oriented Reengineering*, 2005.

[3] J. Whittle and J. Schumann, "Generating statechart designs from scenarios," in *proceedings of the 22^{nd} International Conference on Software Engineering*, 2000.

[4] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jézéquel, "Requirements by contracts allow automated system testing," in *Proc. of the 14^{th}. IEEE International Symposium on Software Reliability Engineering (ISSRE'03)*, 2003.

[5] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *Proceedings of the 28^{th} International Conference on Software Engineering (ICSE 06)*. ACM, 2006.