Towards the Reverse Engineering of UML Sequence Diagrams

L.C. Briand, Y. Labiche, Y. Miao
Software Quality Engineering Laboratory
Systems and Computer Engineering
Carleton University, Ottawa, Ontario, Canada
{briand, labiche}@sce.carleton.ca

Abstract

The objective of the work reported here is to define and assess a method to reverse engineer UML sequence diagrams from execution traces. We do so based on formal transformation rules and we reverse engineer diagrams that show all relevant technical information, including conditions, iterations of messages, and specific object identities and types being involved in the interactions. We present the fundamental principles of our methodology, illustrate it with examples, and validate it through a case study.

1. Introduction

To fully understand an existing object-oriented system (e.g., a legacy system), information regarding its structure and behavior is required. This is especially the case in a context where dynamic binding and polymorphism are used extensively. When no complete and consistent design model is available, one has to resort to reverse engineering to retrieve as much information as possible through static and dynamic analyses. For example, assuming one uses the Unified Modeling Language (UML) notation, the de-facto standard for describing the static and dynamic aspects of object-oriented software, the class, sequence, and statechart diagrams can be reversed-engineered.

Reverse engineering capabilities for the static structure (e.g., the class diagram) of an object-oriented system are already available in many UML CASE tool [16]. However, some challenges still remain to be addressed, such as how to distinguish between association, aggregation and composition relationships, and the reverse engineering of to-many associations. Distinguishing types of associations requires semantic analysis in addition to static analysis of the source code (e.g., composition implies live-time dependencies between the component and the composed class), and identifying to-many associations requires looking at usages of collection classes (e.g., Java Hashtable) that are implementations of to-many associations. Novel and recent tools are starting to address these issues [10].

Reverse engineering and understanding the behavior of an object-oriented system is more difficult. One of the main reasons is that, because of inheritance, polymorphism, and dynamic binding, it is difficult and sometimes even impossible to know beforehand (i.e., using only the source code) the dynamic type of an object reference, and thus which methods are going to be executed. It is then difficult to follow program execution and produce a UML sequence diagram. Similarly, identifying method call sequences from source code requires complex techniques, such as symbolic execution, in addition to source code analysis, and is not likely to be applicable in the case of large and complex systems [6], e.g., the problem of identifying infeasible paths in interprocedural control flow graphs. It then becomes clear that executing the system and monitoring its execution is required if one wants to retrieve meaningful information and reverse-engineer dynamic models, such as UML sequence diagrams from large, complex systems. However, a typical disadvantage of such dynamic approaches is that the (accuracy of the) results depend on how the system is executed. Though this issue is not addressed in this article, a practical solution consists in driving the dynamic analysis with systematic testing techniques as well as code coverage analysis.

In our context, a single execution of the system corresponds to a system level test case executing a single use case scenario and can be modeled using UML sequence diagrams denoted here as scenario diagrams, i.e., incomplete sequence diagrams only denoting what happens in one particular scenario instead of modeling all possible alternatives for a use case. Building a complete sequence diagram, for a given use case, would thus require triggering all possible scenarios through multiple executions of the system, and their analysis/grouping into one sequence diagram. We focus here on the first step,

Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)
1095-1350/03 $ 17.00 © 2003 IEEE

that is the generation of scenario diagrams, and leave out the latter step.

Any strategy to reverse engineer the behavior of a software system has to answer two main questions: (1) How to instrument the code in an efficient manner and analyze trace data to obtain high-level information, and (2) how to present and visualize large amounts of information? The latter issue, visualizing object interaction data (e.g., selecting interesting object interactions) is important but is out of the scope of this article (e.g., see [5] for possible approaches). The approach presented in the current article aims to instrument the source code and execute the system in order to build scenario diagrams. The main challenge comes from the fact that the scenario diagrams produced are not straightforward representations of the traces generated during the execution of the system. For example, the conditions under which calls are executed are reported in the scenario diagram and repetitions of message(s) are identified (if a message is executed several times, it appears only once with a repetition condition in the diagram). To address these issues, we define two metamodels: one for traces and another for scenario diagrams, and define mapping rules between them using the Object Constraint Language (OCL). These rules are then used as specifications to implement a tool to instrument code so as to generate traces, and transform the traces into scenario diagrams.

This article is structured as follows. Related work is first outlined in Section 2. Our approach is then detailed in Section 3 and used on a case study reported in Section 4. Conclusions and future research directions are drawn in Section 5.

2. Related Work

Many strategies aimed to reverse-engineer dynamic models, and in particular interaction diagrams (diagrams that show objects and the messages they exchange), are reported in the literature. Differences are summarized in Table 1. Though not exhaustive, this table does illustrate the differences relevant to our work.

The strategies reported in Table 1 [4, 7, 9, 11, 14, 15, 17] are compared according to seven criteria:

- Whether the granularity of the analysis is at the class or object level. In the former case, it is not possible to distinguish the (possibly different) behaviours of different objects of the same class, i.e., in the generated diagram(s), class X is the source of all the calls performed by all the instances of X.

  In [14], the memory addresses of objects are retrieved to uniquely identify them, though (symbolic) names are usually used in interaction diagrams. The reason is probably (this issue is not discussed by the authors) that retrieving memory addresses at runtime is simpler than using attribute names and/or formal parameter and local variable names to determine (symbolic) names that could be used as unique object identifiers: this requires more complex source code analysis (e.g., problems due to aliasing). Last, it seems that, in [14], methods that appear in an execution trace are not identified by their signature, but by their name (parameters are omitted), thus making it difficult to differentiate calls to overloaded methods.

  Source code analysis is not mentioned in [9] either. In the simple example they use, interacting objects can easily be identified as they correspond to attributes and as there is no aliasing. In [11] objects are identified by numbers, though nothing is said on how those numbers are determined.
- The strategy used to retrieve dynamic information (source code instrumentation, instrumentation of a virtual machine, or the use of a customized debugger[1]) and the target language.
- Whether or not the information used to build interaction diagrams contains data about the flow of control in methods, and whether the conditions corresponding to the flow of controls actually executed are reported. Note that in [15], as mentioned by the authors, it is not possible to retrieve the conditions corresponding to the flow of control since they use a debugger: the information provided is simply the line number of control statements. Though not mentioned in the article, this may also apply to [11].
- The technique used to identify patterns of execution, i.e., sequences of method calls that repeat in an execution trace[1]. The authors in [4, 7, 14, 15] aim to detect patterns of executions resulting from loops in the source code. However, it is not clear, due to lack of reported technical details and case studies, whether patterns of execution that are detected by these techniques can distinguish the execution of loops from incidental executions of identical sequences in different contexts. This is especially true when the granularity of the analysis is at the class level. For instance, it is unclear what patterns existing techniques can detect when two identical sequences of calls in a trace come from two different methods of the same class (no loop is involved).

[1] In the case of [9], this criterion is not applicable as the strategy only uses the source code and no execution trace is produced (no execution is required).

	[7]	[17]	[15]	[9]	[14]	[4]	[11]
Class/Object level	Class	Class	Class	Object	Object (memory address)	Object	Object
Information source	Source code instrumentation	Virtual Machine	Customized Debugger	NA	Source code instrumentation	Virtual Machine	Java Debug Interface
Language	C++	Smalltalk	Java	Java	Smalltalk	Java	Java
Control flow	No	No	Yes	No	No	No	No
Conditions	No	No	No	No	No	No	No
Patterns	String matching (heuristic)	No	String matching	NA	Provided by the user	Recurrences of calls	No
Models produced	MSC	Custom diagrams	UML SD-like	UML CD	UML SD	UML SD-like	UML SD

Table 1. Related works

- The model produced: Message Sequence Chart (MSC), Sequence Diagrams (SD), Collaboration Diagram (CD). Note that in [9], since the control flow information is not retrieved and the approach only uses the source code, the sequences of messages that appear in the generated collaboration diagram can be incorrect, or even unfeasible. Also, the actual (dynamic) type of objects on which calls are performed, which may be different from the static one (due to polymorphism and dynamic binding), is not known. Note that such a static approach, though producing UML interaction diagrams with information on the control flow, is also proposed by tools such as Together [8].

This suggests that a complete strategy for the reverse engineering of interaction diagrams (e.g., a UML sequence diagram) should provide information on: (1) The objects (and not only the classes) that interact, provided that it is possible to uniquely identify them; (2) The messages these objects exchange, the corresponding calls being identified by method signatures; (3) The control flow involved in the interactions (branches, loops), as well as the corresponding conditions. None of the approaches in Table 1 covers all three items and this is the goal of the research reported in this paper.

Another issue, which is more methodological in nature, is how to precisely express the mapping between traces and the target model. Many of the papers published to date do not precisely report on such mapping so that it can be easily verified and built upon. One exception is [9] but this approach is not based on execution traces, as discussed above. Our strategy in this paper has been to define this mapping in a formal and verifiable form as consistency rules between a metamodel of traces and a metamodel of scenario diagrams[2], so as to ensure the completeness of our metamodels and allow their verification.

3. From runtime information to scenario diagrams

Our high-level strategy for the reverse engineering of sequence diagrams consists in instrumenting the source code, executing the instrumented source code (thus producing traces), and analyzing the traces in order to identify repetitions of calls that correspond to loops. We first devise a metamodel of scenario diagrams that is an adaptation of the UML meta-model for sequence diagrams[3] (Section 3.1). This helps us define the requirements in terms of information we need to retrieve from the traces, i.e., what kind of instrumentation is needed. In turn, this results into a metamodel of traces (Section 3.2). Then, the execution of the instrumented system produces a trace (an instance of the trace metamodel), which is transformed by our tool into an instance of the sequence diagram metamodel, using algorithms which are directly derived from consistency rules (or constraints) we define between the two metamodels. Those consistency rules are described in the Object Constraint Language (OCL) [19] and are useful in several ways: (1) They provide a specification and guidance for our transformation algorithms that derive a scenario diagram from a trace (both being instances of their respective meta-model), (2) They help us ensure that our meta-models are correct and complete, as the OCL expression composing the rules must be based on the meta-models.

The implementation of our prototype tool uses Perl [18] for the automatic instrumentation of the source code and Java for the transformation of traces into scenario

[2] The term metamodel is used here to denote a class diagram whose instance represents a trace or scenario diagram, i.e., a model of the system behavior.

[3] This is mostly a simplification so as to simplify our mapping rules and makes the implementation more efficient.

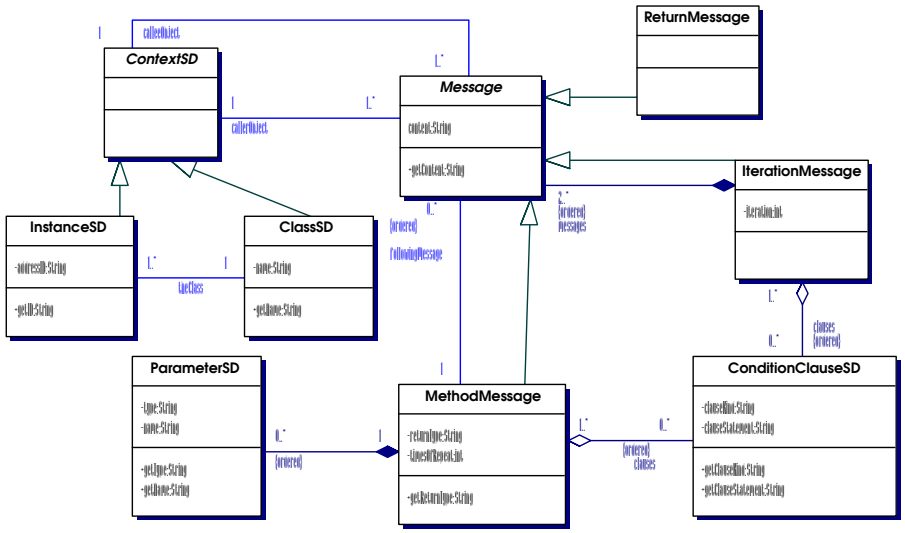

Figure 1. Scenario diagram metamodel (class diagram)

diagrams. The target language has been C++, but it can be easily extended to other similar languages such as Java, as the executed statements monitored by the instrumentation are not specific to C++ (e.g., method's entry and exit, control flow structures). We do not present the complete tool architecture here as the Perl part is simple in that respect, and the core of the Java part consists in the two metamodels presented in this section. Recall that the current work does not address the visualization of reverse-engineered sequence diagrams, as our intent is to interface with existing UML CASE tools and import the generated scenario diagrams using a data interchange format such as XMI [13].

3.1. Sequence diagram metamodel

Sequence diagrams [1] are one the main diagrams used during the analysis and design of object-oriented systems, since a sequence diagram is usually associated to each use case of the system [3]. In practice, though, different scenarios of a given use case can be specified across several sequence diagrams to improve their readability. A sequence diagram describes how objects interact with each other through message sending, and how those messages are sent, possibly under certain conditions, in sequence. We have adapted the UML metamodel [12], that is, the class diagram that describes the structure of sequence diagrams, to our needs, so as to ease the generation of sequence diagrams from traces. Our sequence diagram metamodel is shown in Figure 1.

Messages (abstract class `Message`) have a source and a target (`callerObject` and `calleeObject` respectively), both of type `ContextSD`, and can be of three different kinds: a method call (class `MethodMessage`), a return message (class `ReturnMessage`), or the iteration of one or several messages (class `Iteration Message`). The source and target objects of a message can be named objects (class `InstanceSD`) or anonymous objects (class `ClassSD`). Messages can have parameters (class `ParameterSD`) and can be triggered under certain conditions (class `ConditionClauseSD`): attributes `clauseKind` and `clauseStatement` indicate the type of the condition (e.g., "if", "while") and the exact condition, respectively. The ordered list of `ConditionClauseSD` objects for a `MethodMessage` object corresponds to a logical conjunction of conditions, corresponding to the overall condition under which the message is sent. The iteration of a single message is modeled by attribute `timesOfRepeat` in class `MethodMessage`, whereas the repetition of at least two messages is modeled by class `IterationMessage`. This is due to the different representation of these two situations in UML sequence diagrams. Last, a message can trigger other messages (association between classes `MethodMessage` and `Message`).

3.2 Traces and trace metamodel

We instrument the source code by processing the source code and adding specific statements to retrieve the required information at runtime. These statements are automatically added to the source code and produce one text line in the trace file, reporting on:

- Method entry and exit. The method signature, the class of the target object (i.e., the object executing the method), and the memory address of this object are retrieved.

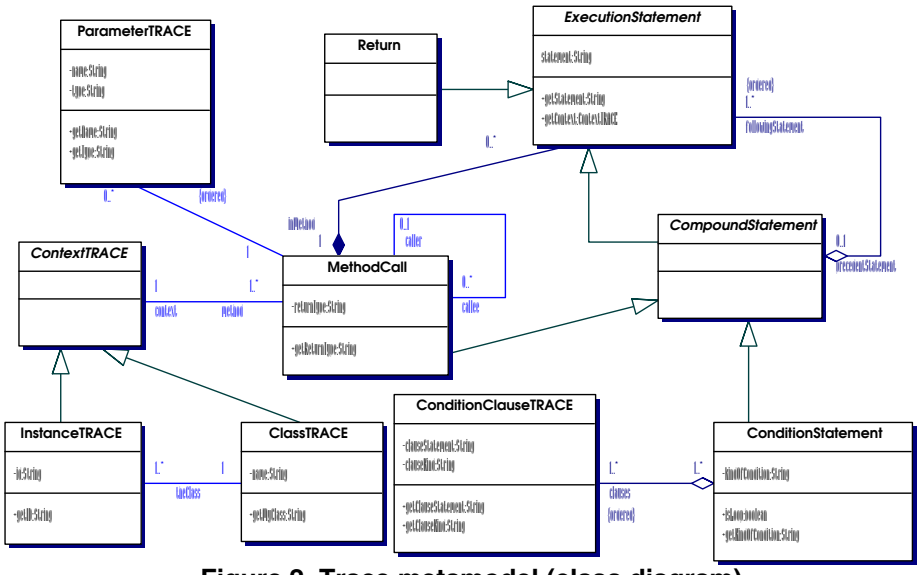

Figure 2. Trace metamodel (class diagram)

- Conditions. For each condition statement, the kind of the statement (e.g., "if") and the condition as it appears in the source code are retrieved.
- Loops. For each loop statement, the kind of the loop (e.g., "while"), the corresponding condition as it appears in the source code, and the end of the loop are retrieved.

These instrumentations are sufficient as it is then possible to retrieve: (1) The source of a call (the object and method) in addition to its target. The source of a call is the previous call in the trace file; (2) The complete condition under which a call is performed (e.g., due to nested if-then-else structures). The conjunction of all the conditions that appear before a call in the trace file form the condition of the call.

When reading trace files produced by these additional statements, it is possible to instantiate the class diagram in Figure 2, which is the metamodel for our traces. This class diagram is similar to our sequence diagram metamodel, though there are some important differences: for instance, a `MethodMessage` object has direct access to its source and target objects (instances of `ContextSD`) whereas a `MethodCall` has access to the object that executes it only (i.e., the target of the corresponding message) and has to query the method that called it to identify the source of the corresponding message. As a consequence, the mapping between the two is not straightforward and the identification of a return message for a call, the complete conditions that trigger calls, and calls that are repeated and located in a loop, are pieces of information that do not appear as is in the trace file but must be computed.

Consider for example the code chunk for method `methodA()`, class `A` in Figure 3. The method body contains a call to method `methodB()` on object `theB` of type `B` inside a while loop, which is itself executed when a condition is fulfilled (`if` statement). Figure 3 also shows the instance of the trace metamodel which is produced when reading the trace file and assuming that: (i) objects of type `A` and `B` are at memory addresses `0x0012FF00` and `0x0012A800` respectively; (ii) `condition1` is true; (iii) the while loop is executed only once. We can see from this simple example that identifying (1) that a call

```
A::methodA() {
  if (condition1)
    while (condition2) { theB.methodB(); }
}
```

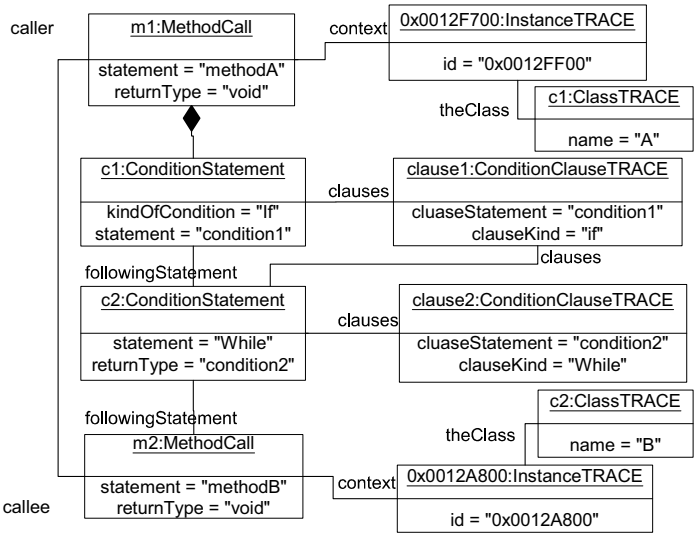

Figure 3. Instance of the trace metamodel – an example

(i.e., call to method `methodB()`) is performed under condition, and (2) what is this condition, may not be straightforward as it involves navigations through the object links of a trace metamodel instance. This is the purpose of the consistency rules we present in the next section, that precisely describe the mapping between metamodels.

3.3 Consistency rules

We have derived three consistency rules, expressed in the OCL, that relate an instance of the trace metamodel to an instance of the sequence diagram metamodel. Note that these OCL rules only express constraints between the two metamodels. They are not algorithms, though they provide a specification and insights into how to implement such algorithms.

These three rules identify instances of classes `MethodMessage`, `ReturnMessage` and `IterationMessage` (sequence diagram metamodel) from instances of classes `MethodCall`, `Return` and `ConditionStatement` (trace metamodel), respectively. We only present the first one here (from `MethodCall` to `MethodMessage` instances) in Figure 4. The other two, that refine what is done in this first rule regarding the return message and conditions, can be found in [2]. Note that these OCL expressions do not have contexts (which is expected in normal usages of OCL). However, we can see, from the expression in Figure 4, that navigations always start from the set of all instances of classes `MethodCall` (scenario diagram metamodel) and `MethodMessage` (trace metamodel), and there is therefore no need for a specific context.

The first three lines in Figure 4 indicate that if method `m1` calls method `m2` (instances of class `MethodCall` in the trace metamodel), then there exists a `MethodMessage` `mm` whose characteristics (attribute values and links to other objects) are described in the rest of the rule. The instance `mm` maps to the instance `m2` (line 4). Then lines 6 to 11 check the link between `mm` and its `callerObject` (instance of class `ContextSD`), i.e., whether `mm` is linked to the object that performed the call to `m2`. Lines 13 to 18 check the link between `mm` and its `calleeObject`, i.e., the object that executed `m2`. Lines 21 to 24 check that the parameters of `mm` (instances of class `ParameterSD`) are consistent with the parameters of `m2` (instances of `ParameterTrace`). Lines 26 to 32 check the conditions that may trigger `mm` and the order in which they are verified. Last, lines 35 to 53 determine how many times message `mm` has been sent. As an example, Figure 5 is the scenario diagram instance that satisfies the rule in Figure 4 based on the trace instance in Figure 3.

Due to space constraints and in order to clearly exemplify the basic principles, the above example is simple and does not fully show why complex consistency rules, as the ones we define, are necessary. More complex examples are provided in [2] for the consistency rules that were not presented here and for the `withdrawal` use case which is part of the case study presented in the next section.

4. Case study

We have selected an Automated Teller Machine (ATM) simulation (the hardware part is missing) system as a case study, since it has been developed independently from the current work[4], provides a complete set of functionalities (withdrawal, deposit, …) and the C++ source code is accompanied by a set of UML diagrams. In particular, sequence diagrams describe the behavior of each use case by specifying object interactions that take place during execution. These diagrams have been developed at the analysis and design stages and we find ourselves in a typical context where the reverse engineering of sequence diagrams can be useful: checking the consistency of the code with the design. In addition, consistency checking between design sequence diagrams and the reverse engineered scenario diagrams allows us to validate our approach and algorithms. The ATM class diagram contains 13 classes (Figure 6). We have selected one use case (`InvalidPIN`) as an example in this section, as it illustrates the most important aspects of the reverse engineering process. Other use cases were investigated and are reported in [2]. Though this case study may appear of limited size, recall that our focus is on retrieving more interesting and complete information on object interactions from the execution of the system (e.g., conditions under which messages are exchanged), rather than to address the visualization problem for large systems.

The `InvalidPIN` use case is triggered when a wrong PIN is provided by the user. Then, the `InvalidPIN` use case, as described in the corresponding sequence diagram in Figure 7 (sequence diagram coming from the design documentation), asks the user to re-enter the PIN: three tries are allowed. In case the three entered PIN's are incorrect, the card is retained, and the customer has to contact the bank.

[4] www.math-cs.gordon.edu/local/courses/cs211/ATMExample/

```
1  methodCall.allInstances->forAll(m1,m2: MethodCall | m1.callee->includes(m2)
2  implies
3    methodMessage.allInstances->exists(mm:MethodMessage |
4      mm.content = m2.statement
5      and
6      if m1.context.oclType = InstanceTRACE then (  // checking the caller
7        mm.callerObject.addressID =m1.context.id and
8        mm.callerObject.theClass.name = m1.context.theClass.name
9      ) else (
10       mm.callerObject.name = m1.context.name
11     )
12     and
13     if m2.context.oclType = InstanceTRACE then (  // checking the callee
14       mm.calleeObject.addressID = m2.context.id and
15       mm.calleeObject.theClass.name = m2.context.theClass.name
16     ) else (
17       mm.calleeObject.name = m2.context.name
18     )
19     and
20     mm.returnType = m2.returnType and
21     mm.parameterSEQD->forAll(index:Integer |      // checking the parameters
22       mm.parameterSEQD->at(index).name = m2.parameterTRACE->at(index).name and
23       mm.parameterSEQD->at(index).type = m2.parameterTRACE->at(index).type
24     ) and mm.parameterSEQD->size = m2.parameterTRACE->size
25     and     // checking the conditions
26     if m2.precedentStatement.oclType = ConditionStatement then (
27       mm.clause->forAll(index:Integer |
28         mm.clause->at(index).clauseStatement =
29           m2.precedentStatement.clauses->at(index).clauseStatement and
30         mm.clause->at(index).clauseKind =
31           m2.precedentStatement.clauses->at(index).clauseKind
32       ) and mm.clause->size = m2.precedentStatement.clauses->size
33     and
34     // checking repeated executions
35     if ( m2.precedentStatement.oclType = ConditionStatement and
36         m2.precedentStatement.isLoop = true and
37         m2.precedentStatement.followingMessage->size =1 ) then
38       mm.timesOfRepeat = methodCall.allInstances->select(m3:MethodCall|
39         m3.statement = m2.statement and m3.returnType = m2.returnType and
40         m3.caller = m2.caller and
41         m3.parameterTRACE->forAll(index:Integer|
42           m3.parameterTRACE->at(index).name = m2.parameterTRACE->at(index).name and
43           m3.parameterTRACE->at(index).type = m2.parameterTRACE->at(index).type
44         ) and
45         m3.parameterTRACE->size = m2.parameterTRACE->size and
46         m3.precedentStatement.oclType = ConditionStatement and
47         m3.precedentStatement.clauses->forAll(index:integer|
48           m3.precedentStatement.clauses->at(index).clauseStatement =
49             m2.precedentStatement.clauses->at(index).clauseStatemenr) and
50           m3.precedentStatement.clauses->at(index).clauseKind =
51              m2.precedentStatement.clauses->at(index).clauseKind)
52          and m3.precedentStatement.clauses->size = m2.precedentStatement.clauses->size
53       )->size
54     and
55     mm.followingMessage->forAll(mm1:Message| mm1.callerObject = mm.calleeObject)
56      )
57 )
```

Figure 4. Consistency rule for MethodMessage instances

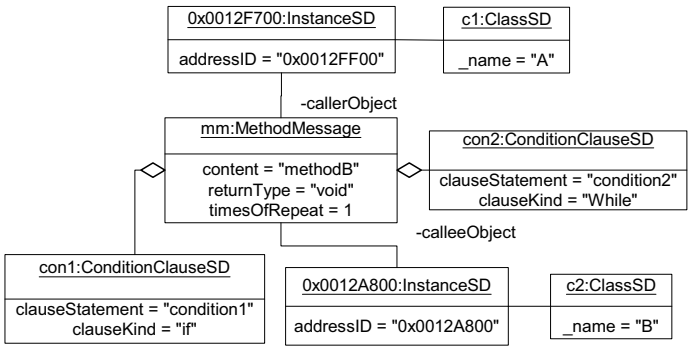

Figure 5. Consistency rule – an example

The instrumented system was executed so as to trigger the `InvalidPIN` use case (the other selected use cases were also executed and analyzed in a similar way [2]), and the text file produced by our prototype tool was transformed into a UML scenario diagram (Figure 8).

Before discussing the resulting scenario diagram, recall that use cases often have sequential dependencies, e.g., some use cases need to be executed before others in a valid system usage scenario [3]. This is the case here as the `InvalidPIN` use case was triggered during a tentative withdrawal (the `Withdrawal` use case is executed), which itself, requires that the `InsertCard` use case be executed (before the `Withdrawal` use case). Figure 8 does not show the whole scenario diagram generated during this execution of the system but only the messages that correspond to the `InvalidPIN` use case, as we want to compare it with the corresponding design sequence diagram (Figure 7).

As a consequence, the first message is numbered `9.5.5.3` (instead of `1`) as messages corresponding to the `Withdrawal` and `InsertCard` use cases are executed beforehand and messages take place after `9.5.5.3.4` (the last message in Figure 8) to model how the ATM system behaves once a card has been retained.

Comparing Figure 7 and Figure 8, we can see that, as expected, the reverse engineered scenario diagram has a consistent structure to the design sequence diagram. It, however, contains more details as additional messages, for example to GUI objects, are shown. Messages `1.1` and `1.1.1` in Figure 7 correspond to messages `9.5.5.3.1` to `9.5.5.3.1.3` in Figure 8 (the ATM asks for a PIN). Then the ATM sends the information to the Bank: message `1.2` in Figure 7 and `9.5.5.3.2` in Figure 8. It is worth noting that the reverse engineered scenario diagram indicates returned values (e.g., message `9.5.5.3.4`) and that conditions under which messages are sent are provided. For example, the `InvalidPIN` use case is triggered (message numbered `9.5.5.3: doInvalidPINExtension`) when an invalid PIN has been received as shown by the guard condition for message `9.5.5.3: [status=INVALID_PIN]` (as it appears in the source code).

Note also that Figure 8 shows that once a PIN has been re-entered, the message sent to the bank (as a result of message `sendToBank` to the transaction) is not `doTransaction` (message number `1.2.1` in Figure 7) but `InitiateWithdrawal` (message number `9.5.5.3.2.4` in Figure 8), since the `InvalidPIN` use case was triggered during the executing of a withdrawal (the `Transaction` object is in fact a `Withdrawal`). Last, the reverse engineered scenario diagram explicitly shows the set of messages that are repeated: a rectangle encloses the repeated messages, and a constraint at the bottom of the rectangle indicates the recurrence condition (again the constraint is written as it appears in the source code).

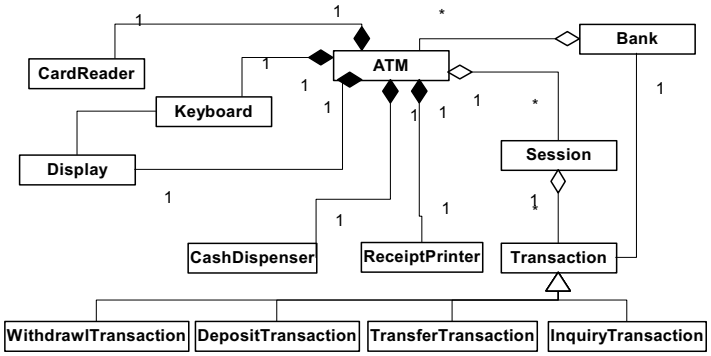

Figure 6. ATM class diagram

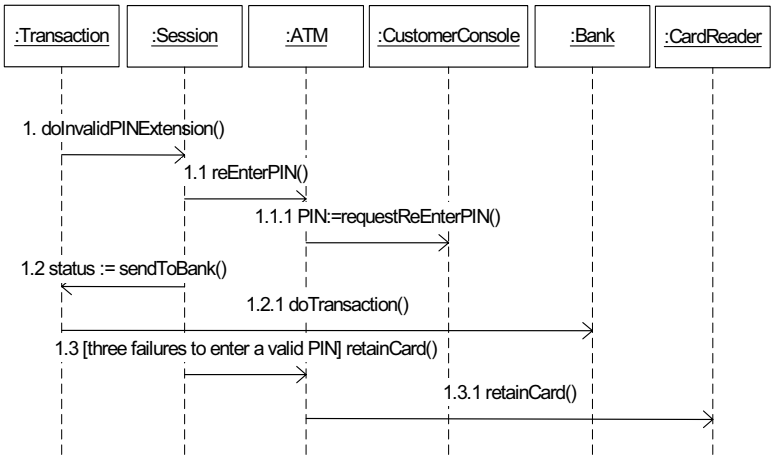

Figure 7. InvalidPIN sequence diagram

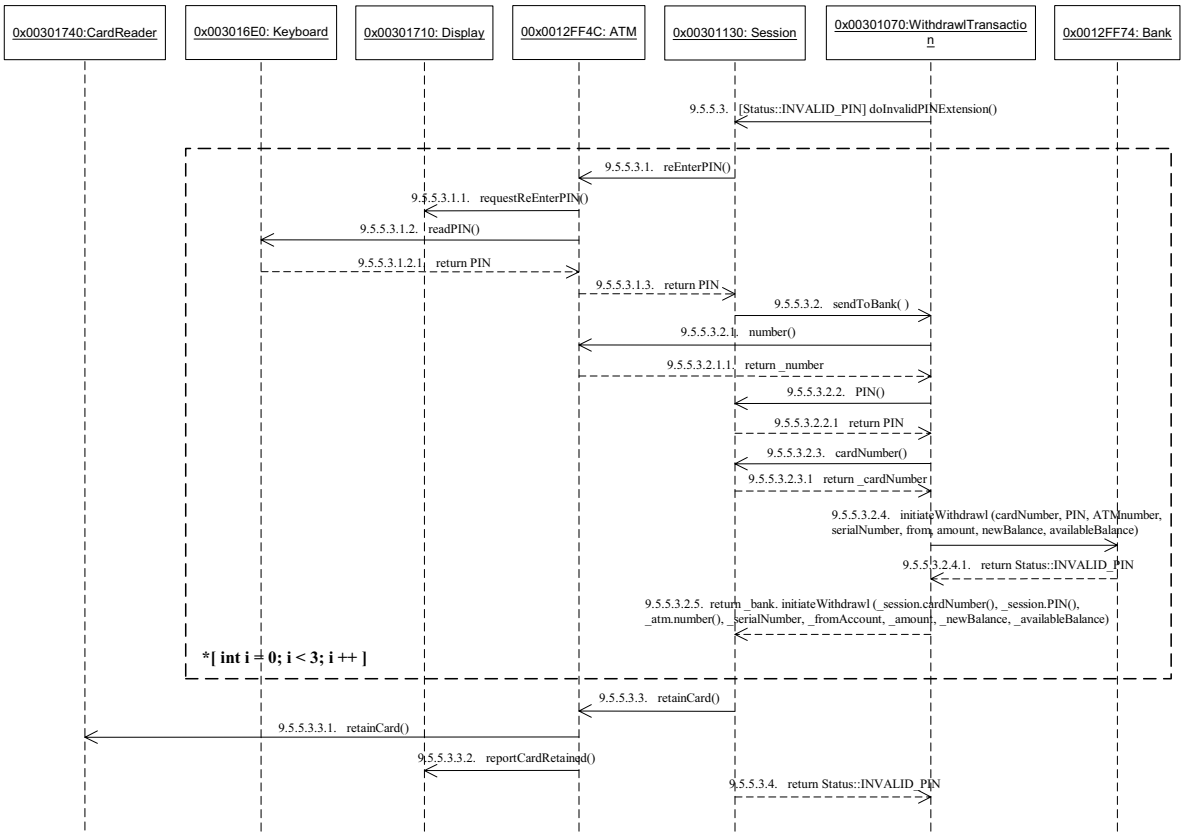

Figure 8. Reverse engineered InvalidPIN sequence diagram

5. Conclusion

Reverse engineering techniques are required to understand the structure and behavior of a software system whose documentation is missing or out of date. This article is a first step towards the full reverse engineering of the behavior of a software system (rather than its structure) under the form of UML (Unified Modeling Language) sequence diagrams. UML sequence diagrams can represent either several scenarios corresponding to the possible executions of a use case or one single scenario, in which case we use the term scenario diagram instead of sequence diagram.

Any strategy to reverse engineer the behavior of a software system has to face two main problems: How to retrieve interesting information and then how to visualize it for large systems. The strategy we report here addresses the first issue for the reverse engineering of UML interaction diagrams: It aims at building a scenario diagram for each execution of an instrumented system. To do so in a precise and rigorous manner, we define a mapping between two models. We first model the relevant aspects of a sequence diagram as a class diagram (metamodel) in order to identify the requirements for source code instrumentation. We then model what information execution traces must contain under the form of a trace metamodel. We then formally describe, using OCL (Object Constraints Language), the mapping that must exist between the two metamodels, i.e., how instances of the trace metamodel (i.e., a trace) relate to instances of the sequence diagram metamodel (i.e., a scenario diagram). The generated scenario diagrams report on objects that interact (we use their memory addresses in order to uniquely identify them), messages these objects exchange (i.e., the method calls that are performed, including parameters), conditions under which these messages are exchanged (i.e., the conditions appear in the sequence diagrams as they are written in the source code), and repetitions of one or several messages (again, the recurrence conditions are shown as they appear in the source code).

A platform and compiler independent prototype tool implementing the strategy has been built. It consists of an instrumentation part, and a scenario generation part that transforms a trace into a scenario diagram according to the formal mapping defined above as OCL constraints.

As a case study, we performed the reverse engineering of several scenario diagrams for an Automated Teller Machine (ATM) system that was designed (using UML)

and developed independently from our work. Results show that our tool produces scenario diagrams that are consistent with the design sequence diagrams but that contain additional details that were not described by the designers of the system.

Besides the use of our approach on other case studies, five directions can be identified for future research. First, instrumentation should be improved so as to get additional information and ease the understanding of the system (e.g., using reference names instead of memory addresses to identify objects). When more than one condition triggers a call (e.g., nested loops in the source code), all the conditions are reported in the scenario diagram without any change. This conjunction of conditions may have redundant terms and, when this is the case, it would be useful to simplify it when possible. Techniques have to be defined to group related scenario diagrams (e.g., from different executions) into one sequence diagram. A query system similar to what has already been proposed in the literature (e.g., [14]) can be developed to limit the range of investigation, i.e., select classes (or interaction) of interest, thus concentrating only on important interactions from the user viewpoint and help cope with the important amount of data. Last but not least, the method proposed here was used for non-concurrent, non-distributed objects. Other issues may arise when trying to reverse engineer scenario diagrams that report on asynchronous messages being sent to distributed, concurrent objects.

Acknowledgement

This work was partly supported by a Canada Research Chair grant. Lionel Briand and Yvan Labiche were further supported by NSERC operational grants. This work is part of a larger project (TOTEM) on testing object-oriented systems with the UML (TOTEM: www.sce.carleton.ca/ Squall/Totem).

References

[1] G. Booch, J. Rumbaugh and I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, 1999.

[2] L. C. Briand, Y. Labiche and Y. Miao, "Towards the Reverse Engineering of UML Sequence Diagrams," Carleton University, Technical Report SCE-03-03, www.sce.carleton.ca/Squall/Articles/TR_SCE-03-03.pdf, February, 2003.

[3] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering - Conquering Complex and Chalenging Systems*, Prentice Hall, 2000.

[4] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides and J. Yang, "Visualizing the Execution of Java Programs," in S. Diehl, Ed., *Software Visualization*, vol. 2269, *Lecture Notes in Computer Science*, Springer Verlag, 2002, pp. 151-162.

[5] S. Diehl, *Software Visualization*, Lecture Notes in Computer Science, vol. 2269, Springer Verlag, 2002.

[6] C. Ghezzi, M. Jazayeri and D. Mandrioli, *Fundamentals of Software Engineering*, Prentice Hall International Ed., 1991.

[7] D. F. Jerding, J. T. Stasko and T. Ball, "Visualizing Interactions in Program Executions," *Proc. ICSE*, pp. 360-370, 1997.

[8] J. Kern, "Sequence Diagram Generation - Effective Use of Options," TogetherSoft, White Paper, 2001.

[9] R. Kollmann and M. Gogolla, "Capturing Dynamic Program Behaviour with UML Collaboration Diagrams," *Proc. CSMR*, pp. 58-67, 2001.

[10] R. Kollmann, P. Selonen, E. Stroulia, T. Systa and A. Zundorf, "A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering," *Proc. WCRE*, pp. 22-32, 2002.

[11] R. Oechsle and T. Schmitt, "JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI)," in S. Diehl, Ed., *Software Visualization*, vol. 2269, *Lecture Notes in Computer Science*, 2002, pp. 176-190.

[12] OMG, "Unified Modeling Language," Object Management Group V1.4, www.omg.org/technology/uml/, 2001.

[13] OMG, XML Metadata Interchange, www.omg.org/technology/documents/formal/xmi.htm, 2001

[14] T. Richner and S. Ducasse, "Using Dynamic Information for the Iterative Recovery of Collaborations and Roles," *Proc. ICSM*, pp. 34-43, 2002.

[15] T. Systa, K. Koskimies and H. Muller, "Shimba - An Environment for Reverse Engineering Java Software Systems," *Software - Practice and Experience*, vol. 31 (4), pp. 371-394, 2001.

[16] TogetherSoft™, "Together", www.togethersoft.com.

[17] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson and J. Isaak, "Visualizing Dynamic Software System Information through High-Level Models," *Proc. OOPSLA*, pp. 271-283, 1998.

[18] L. Wall, *Programming Perl*, O'Reilly & Associates, 3rd Ed., 2000.

[19] J. Warmer and A. Kleppe, *The Object Constraint Language*, Addison-Wesley, 1999.