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

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 so that reverse engineered diagrams 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 UML (Unified Modeling Language) notation, 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 [12]. 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 [7].

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
Technical Report SCE-03-03  February 2003

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 [3], 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.

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

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, 6, 10, 11, 13] 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 [10], 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 [10], 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 [6] either. In the simple example they use, interacting objects can easily be identified as they correspond to attributes and as there is no aliasing.

- 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 [11], 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.

- 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, 10, 11] 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).

- The model produced: Message Sequence Chart (MSC), Sequence Diagrams (SD), Collaboration Diagram (CD). Note that in [6], since the control flow information is not retrieved, the sequences of messages that appear in the generated collaboration diagram can be incorrect, or even unfeasible. Also, since the strategy only uses the source code, 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

---

\(^1\) In the case of [6], this criterion is not applicable as the strategy only uses the source code and no execution trace is produced (no execution is required).
approach, though producing UML sequence diagrams with information on the control flow, is also proposed by tools such as Together [5].

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Class/Object</td>
<td>Class</td>
<td>Class</td>
<td>Class</td>
<td>Object</td>
<td>Object (memory address)</td>
</tr>
<tr>
<td>level</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Information</td>
<td>Source code instrumentation</td>
<td>Virtual Machine</td>
<td>Customized Debugger</td>
<td>NA</td>
<td>Source code instrumentation</td>
</tr>
<tr>
<td>source</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Language</td>
<td>C++</td>
<td>Smalltalk</td>
<td>Java</td>
<td>Java</td>
<td>Smalltalk</td>
</tr>
<tr>
<td>Control flow</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Conditions</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Patterns</td>
<td>String matching (heuristic)</td>
<td>No</td>
<td>String matching</td>
<td>NA</td>
<td>Provided by the user</td>
</tr>
<tr>
<td>Models produced</td>
<td>MSC</td>
<td>Custom diagrams</td>
<td>SD (based on the UML notation)</td>
<td>UML CD</td>
<td>UML SD</td>
</tr>
</tbody>
</table>

Table 1 Related Works

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 cover 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 [6] 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, so as to ensure the completeness of our metamodels and allow their verification.

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 FROM RUN-TIME 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, which is transformed by our tool into an instance of the trace metamodel, using algorithms which are directly derived from consistency rules (or constraints) we define between the two metamodels. Those consistency rules are described in OCL 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 \([14]\) for the automatic instrumentation of the source code and Java for the transformation of traces into scenario 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. Note 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 \([9]\).

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 \([2]\). 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 \([8]\), 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:

\(^3\) This is mostly a simplification so as to simplify our mapping rules and makes the implementation more efficient.
a method call (class `MethodMessage`), a return message (class `ReturnMessage`), or the iteration of one or several messages (class `IterationMessage`). 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`).

![Figure 1 Scenario diagram metamodel (class diagram)](image)

### 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.
- 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.

![Figure 2 Trace metamodel (class diagram)]
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 `0x0012F700` 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 (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 implementing 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 Appendix A. 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.
methodCall.allInstances->forAll(m1,m2: MethodCall | m1.callee->includes(m2)) implies methodMessage.allInstances->exists(mm:MethodMessage | 
  mm.content = m2.statement  
  and if m1.context.oclType = InstanceTRACE then ( // checking the caller 
    mm.callerObject.addressID = m1.context.id and 
    mm.callerObject.theClass.name = m1.context.theClass.name 
  ) else { 
    mm.callerObject.name = m1.context.name 
  } 
  and if m2.context.oclType = InstanceTRACE then ( // checking the callee 
    mm.calleeObject.addressID = m2.context.id and 
    mm.calleeObject.theClass.name = m2.context.theClass.name 
  ) else { 
    mm.calleeObject.name = m2.context.name 
  } 
  and mm.returnType = m2.returnType and 
  mm.parameterSEQD->forAll(index:Integer | // checking the parameters 
    mm.parameterSEQD->at(index).name = m2.parameterTRACE->at(index).name and 
    mm.parameterSEQD->at(index).type = m2.parameterTRACE->at(index).type 
  ) and mm.parameterSEQD->size = m2.parameterTRACE->size 
  and // checking the conditions 
  if m2.precedentStatement.oclType = ConditionStatement then ( 
    mm.clause->forAll(index:Integer | 
      mm.clause->at(index).clauseStatement = 
      m2.precedentStatement.clauses->at(index).clauseStatement and 
      mm.clause->at(index).clauseKind = 
      m2.precedentStatement.clauses->at(index).clauseKind 
    ) and mm.clause->size = m2.precedentStatement.clauses->size 
  ) 
  and // checking repeated executions 
  if ( m2.precedentStatement.oclType = ConditionStatement and 
    m2.precedentStatement.isLoop = true and 
    m2.precedentStatement.followingMessage->size = 1 ) then 
    mm.timesOfRepeat = methodCall.allInstances->select(m3:MethodCall | 
      m3.statement = m2.statement and m3.returnType = m2.returnType and 
      m3.caller = m2.callee and 
      m3.parameterTRACE->forAll(index:Integer | 
        m3.parameterTRACE->at(index).name = m2.parameterTRACE->at(index).name and 
        m3.parameterTRACE->at(index).type = m2.parameterTRACE->at(index).type 
      ) and 
      m3.parameterTRACE->size = m2.parameterTRACE->size and 
      m3.precedentStatement.oclType = ConditionStatement and 
      m3.precedentStatement.clauses->forAll(index:Integer | 
        m3.precedentStatement.clauses->at(index).clauseStatement = 
        m2.precedentStatement.clauses->at(index).clauseStatement and 
        m3.precedentStatement.clauses->at(index).clauseKind = 
        m2.precedentStatement.clauses->at(index).clauseKind 
      ) and 
      m3.precedentStatement.clauses->size = m2.precedentStatement.clauses->size 
    )->size 
    and 
    mm.followingMessage->forAll(mm1:Message | mm1.callerObject = mm.calleeObject) 
  )

Figure 4 Consistency rule for MethodMessage instances

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

![Figure 5 Consistency rule – an example](image)

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 Appendix A for the consistency rules that were not presented here and in Appendix B 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 (www.math-cs.gordon.edu/local/courses/cs211/ATMExample/), 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 reversed engineer scenario diagrams allow us to validate our approach and algorithms.
The ATM class diagram contains 13 classes (Figure 6). We have selected two use cases, (Withdrawal and InvalidPIN) as an example in this section, as they illustrate complementary aspects of the reverse engineering process. Other use cases were investigated and are reported in Appendix C.

![Figure 6 ATM class diagram](image)

### 4.1 The Withdrawal use case

As shown in the design sequence diagram corresponding to use case Withdrawal (Figure 7), the customer is first asked to input an account and an amount. Note that class CustomerConsole is not part of the class diagram in Figure 6: it corresponds to classes Keyboard and Display. Then the system verifies whether there is enough cash in the ATM before sending the request to the bank (message numbered 1.4). If the bank approves the transaction, the cash is dispensed and the receipt corresponding to the transaction is issued.

One withdrawal transaction was executed on the instrumented system, and the generated trace file was processed by our prototype tool. The produced trace text file was then transformed into a UML sequence diagram (Figure 8). Instances of the two trace and scenario metamodels (i.e., object diagrams) generated by the tool can be found in Appendix B. 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 [2]. This is the case here as the InsertCard use case needs to be executed before the Withdrawal use case. Figure 8 does not show the whole scenario diagram generated but only the messages that correspond to the Withdrawal use case, as we want to compare it with the corresponding design sequence diagram (Figure 7). As a consequence, the first message is numbered 9.5.5.1 (instead of 1) as messages corresponding to the InsertCard use case are executed beforehand and messages take place after 9.5.5.3.7 (the last message in Figure 8) to model how the ATM system asks to the customer for another transaction when one transaction has been completed.

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 to 1.1.1.1 in Figure 7 correspond to messages 9.5.5.1.1 to 9.5.5.1.1.2 in Figure 8 (the customer chooses an account). Then the ATM asks for an amount and checks whether there is enough cash in the dispenser: messages 1.3 and 1.3.1 in Figure 7 and 9.5.5.1.3 to 9.5.5.1.3.2 in Figure 8. It is worth noting that the reverse engineered scenario diagram indicates returned values (e.g., message 9.5.5.1.3.2) and that conditions under which messages are sent are provided. For example, a withdrawal transaction is completed (message finishApprovedTransaction numbered 9.5.5.3) once the bank has validated the information that has been sent at message initiateWithdrawal numbered 9.5.5.2.4 (there is enough cash in the account) and thus the guard condition for message 9.5.5.3 is [code=Status::SUCCESS] (as it appears in the source code).

Figure 7 Withdrawal sequence diagram
4.2 The InvalidPIN use case

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 9 (sequence diagram coming from the design documentation), asks to 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.
Figure 9 InvalidPIN sequence diagram

Again, the system was executed so as to trigger the InvalidPIN use case, during a tentative withdrawal, and the text file produced by our prototype tool text file was transformed into a UML scenario diagram (Figure 10).

Comparing Figure 9 and Figure 10, we can see that, again, the reverse engineered sequence diagram is consistent with the one provided in design documents but contains a great deal of additional details. In particular, it 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 9) but InitiateWithdrawal (message number 9.5.5.3.2.4 in Figure 10), since the InvalidPIN use case was triggered during the executing of a withdrawal (the Transaction object is in fact a Withdrawal). Also, 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.
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 report 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. In a UML context, such an exercise can also be useful to verify the consistency between design and implementation by comparing reverse engineered sequence diagrams with the ones documented during the design stage. 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.

The strategy we report here 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, which is specific to C++ (though it can easily be adapted to other languages such as Java), and a scenario generation part that transforms a trace into a scenario diagram according to the formal mapping defined above as OCL constraints. Note that the tool produces textual representations of scenario diagrams, thus requiring their import into UML CASE tools, for example as XMI files, to visualize them.

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.

Four directions can be identified for future research. First, instrumentation should be improved so as to get additional information. This includes the use of reference names (e.g., attribute names, local variable names) instead of memory addresses to identify objects. This would be beneficial in two ways: (1) it would ease the understanding of the system under study as the mapping between the scenario diagrams and the source code would be more straightforward, (2) It would make our approach suitable to situations where object addresses change at run time (e.g., use of virtual memory). This is however difficult because of object aliasing and requires complex static analysis of the source code. A similar issue is the reporting of actual parameters instead of formal parameters when method calls are detected at run time.

When more than one condition triggers a call (e.g., in the source code, the call is located inside two nested loops), all the conditions are reported in the scenario diagram without any change: the message condition is a conjunction of the conditions that trigger the call. This conjunction 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 into one sequence diagram. For instance, this occurs when one executes several times the same use case in different ways, thus producing several scenario diagrams corresponding to different use case scenarios. It would then be useful to merge these scenario diagrams into a sequence diagram for the use case.

Last, a query system similar to what as already been proposed in the literature (e.g., [10]) 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. For
instance, the user may not want to report on classes that are part of the GUI or may limit
the scenario diagram scope to specific classes.

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.

6 REFERENCES


Executions,” *Proc. ACM International Conference on Software Engineering


Collaboration Diagrams,” *Proc. IEEE European Conference on Software

Current State of the Art in Tool-Supported UML-Based Static Reverse
Engineering,” *Proc. IEEE Working Conference on Reverse Engineering,


[9] OMG, XML Metadata Interchange (XMI),

of Collaborations and Roles,” *Proc. IEEE International Conference of Software
Maintenance (ICSM)*, Montreal, Quebec, pp. 34-43, 2002.

Engineering Java Software Systems,” *Software - Practice and Experience*, vol. 31


Isaak, “Visualizing Dynamic Software System Information through High-Level
Models,” *Proc. ACM SIGPLAN Conference on Object-Oriented Programming,*
Appendix A Additional Consistency Rules

A.1 Identifying ReturnMessage instances

Though not necessary in sequence diagrams, we create instances of class ReturnMessage that hold the expressions returned in the return statements. The structure of the rule shown in Figure 11 is similar to the one presented in Figure 4. The first three lines indicate that if there is a Return object \( r \) in the instance of the trace metamodel, there is a ReturnMessage \( rm \) instance in the instance of sequence diagram metamodel. Again, the characteristics of \( rm \) (callerObject and calleeObject objects) are described in the rest of the rule, according to the characteristics of \( r \).

```
1 return.allInstances->forAll(r :Return )
2   returnMessage.allInstances->exists(rm :ReturnMessage | 
3       // obtain information about the message to be returned
4       rm.content = r.statement
5       and
6       // obtain the caller object of the message
7       if r.getContext().oclType = InstanceTRACE then 
8         rm.callerObject.addressID = r.getContext().id
9       and
10      rm.callerObject.theClass.name =  r.getContext().theClass.name
11   } else {
12     rm.callerObject.name = r.getContext().name
13   }
14   and
15   // obtain the callee object of the message
16   if r.inMethod.context.oclType = InstanceTRACE then 
17     rm.calleeObject.addressID = r.inMethod.context.id
18   and
19     rm.calleeObject.theClass.name = r.inMethod.context.theClass.name
20   } else {
21     rm.calleeObject.name = r.inMethod.context.name
22 } 
```

Figure 11 Consistency rule for ReturnMessage instances

As an example, Figure 12 shows a code chunk (upper right part), the corresponding instance of the trace metamodel (upper left part), and the generated instance of the sequence diagram metamodel. It is assumed that instances of classes A and B are at memory addresses 0x0012FF00 and 0x0012A800, respectively.
A2 Identifying IterationMessage instances

Rule in Figure 13 shows how instances of class IterationMessage (sequence diagram metamodel) relate to instance of class ConditionStatement (trace metamodel).
conditionStatement.allInstances->forAll(c :ConditionStatement | (c.isLoop = true and c.followingStatement->size > 1 ) implies iterationMessage.allInstances->exists(im :IterationMessage | // obtain the loop condition of the group of messages im.content = c.statement and // obtain the number of iteration of the group of messages im.iteration = conditionStatement.allInstances->select(c1:ConditionStatement | c1.clauses->forAll(index:Integer | c1.clauses->at(index).clauseStatement = c.clauses->at(index).clauseStatement and c1.clauses->at(index).clauseKind = c.clauses->at(index).clauseKind ) and c1.clauses->size = c.clauses->size and c1.kindOfCondition = c.kindOfCondition and c1.precedentStatement = c.precedentStatement )->size and // obtain the condition clauses of the group of messages im.clauses->forAll(index:Integer | im.clauses->at(index).clauseStatement = c.clauses->at(index).clauseStatement and im.clauses->at(index).clauseKind = c.clauses->at(index).clauseKind ) and im.clauses->size = c.clauses->size )

Figure 13 Consistency rule for iterationMessage instances

As an example, Figure 14 shows a code chunk (upper right part), the corresponding instance of the trace metamodel (upper left part), and the generated instance of the sequence diagram metamodel. It is assumed that instances of classes A and B are at memory addresses 0x0012FF00 and 0x0012A800, respectively.
Appendix B Instances of the metamodels for the withdrawal use case

Figure 15 shows the body of method serviceCustomers() in class ATM, i.e., the method that loops indefinitely until the ATM is shut down (while loop in line 4), and accepts request from customers. In this loop, the ATM displays on the screen the message that asks the customer to enter a card. It then loops, waiting for the card to be inserted (do}
loop in line 6). When the card is inserted, a Session object is created (line 10), and the session starts (line 11). The choice of a particular transaction (in our case a withdrawal) is done during the execution of method Session::doSessionUseCase() (line 11).

```cpp
void ATM::serviceCustomers() {
    CardReader::ReaderStatus readerStatus;
    _state = RUNNING;
    while (_state == RUNNING) {
        _display.requestCard();
        do {
            readerStatus = _cardReader.checkForCardInserted();
        } while (_state == RUNNING && readerStatus == CardReader::NO_CARD);
        if (_state == RUNNING && readerStatus == CardReader::CARD_HAS_BEEN_READ) {
            Session *session = new Session (_cardReader.cardNumber(), *this, _bank);
            session -> doSessionUseCase();
            delete session;
            _state = STOPPED;
        }
    }
}
```

**Figure 15 Code chunk from class ATM**

Figure 16 is an excerpt of the instance of the trace metamodel corresponding to the execution of method ATM::serviceCustomer(). The three objects at the top left of the figure correspond to the instance of class ATM (at memory address 0x0012FF4C) that executes serviceCustomer() (method call m1). The aggregation between m1 and cond1 (instance of class ConditionStatement) correspond to the first loop in serviceCustomer() (line 4 in Figure 15). cond1 is then aggregated with: m2 of type MethodCall (on the right), corresponding to line 5 in Figure 15; cond2 of type ConditionStatement (on the bottom), corresponding to the second loop (line 6 in Figure 15); and cond3 of type ConditionStatement (bottom left), corresponding to the if statement in line 9 (Figure 15). Note that nothing indicates in Figure 16 that m2, cond2, and cond3 are triggered in sequence (lines 5, 6 and 9 in Figure 15). However, this information is part of the instance of the trace metamodel, as indicated by the ordered constraints in the trace metamodel class diagram (Figure 2) between classes CompoundStatement and ExecutionStatement.
Figure 16 Trace instance for the withdrawal use case

Figure 17 is an excerpt of the instance of the sequence diagram metamodel produced after the use of the OCL consistency rules to the instance of the trace meatmodel in Figure 16. First, a MethodMessage instance named mm1 is created (upper left part of Figure 17) and corresponds to object named m1 (of type MethodCall) in Figure 16. mm1 triggers the iteration of (link followedMessage from mm1 to object named im of type IterationMessage):

- MethodMessage mm2 (corresponding to MethodCall object m2);
- MethodMessage mm3 (corresponding to MethodCall object m3) whose execution is repeated (link between mm3 and ConditionClauseSD object con1);
- MethodMessage mm4 (corresponding to MethodCall object m4) whose execution is conditional (link between mm3 and ConditionClauseSD object cond3);
- MethodMessage mm5 (corresponding to MethodCall object m5) whose execution is conditional (link between mm3 and ConditionClauseSD object cond3);
- MethodMessage mm6 (corresponding to MethodCall object m6) whose execution is conditional (link between mm3 and ConditionClauseSD object cond3).

Note that these aggregations are ordered (order of execution), as indicated by the ordered constraints between classes IterationMessage and Message in the sequence diagram metamodel (Figure 1).
Appendix C Additional sequence diagrams (Case study)

In this appendix, we provide two sequence diagrams, the first one comes from the design documentation and the second one is reverse engineered, for each of the following use cases: Session (Section C.1), Deposit (Section C.2), Transfer (Section C.3), Inquiry (Section C.4).
C.1 Use case Session

Figure 18 Session sequence diagram
Figure 19 Reverse engineered Session scenario diagram
C.2 Use case Deposit

Figure 20 Deposit sequence diagram
9.5.5.1. getTransactionSpecificsFromCustomer()  
9.5.5.1.1. chooseAccountType(purpose, atm)  
9.5.5.1.1.1. getMoneyChoice(purpose, item[1])  
9.5.5.1.1.1.1. displayMenu(what, numbers, items)  
9.5.5.1.1.1.1.2. return choice  
9.5.5.1.1.2. requestMoneyEntry()  
9.5.5.1.1.2.1. readMoneyEntry()  
9.5.5.1.1.2.2. return Money(dollar, cent)  
9.5.5.1.1.2.3. return amount  
9.5.5.1.2. return AccountType(choice-1)  
9.5.5.2. sendToBank()  
9.5.5.2.1. number()  
9.5.5.2.1.1. return _number  
9.5.5.2.2. PIN()  
9.5.5.2.2.1. return PIN  
9.5.5.2.3. cardNumber()  
9.5.5.2.3.1. return cardNumber  
9.5.5.2.4. initiateDeposit(cardNumber, PIN, _serialNumber, _toAccount, _amount, _newBalance, _availableBalance)  
9.5.5.2.4.1. return Status::SUCCESS  
9.5.5.2.5. return _bank.initiateDeposit(_session.cardNumber(),_session.PIN(),_atm.number(),_serialNumber,_toAccount,_amount,_newBalance,_availableBalance)  
9.5.5.3. [code == Status::SUCCESS]  
9.5.5.3.1. finishApprovedTransaction()  
9.5.5.3.1.1. number()  
9.5.5.3.1.1.1. return _number  
9.5.5.3.2. finishDeposit(ATMnumber, serialNumber, succeeded)  
9.5.5.3.3. [envelopeAccepted] accountName(type)  
9.5.5.3.3.1. return accountName[type]  
9.5.5.3.4. [envelopeAccepted] cardNumber()  
9.5.5.3.4.1. return _cardNumber  
9.5.5.3.5. [envelopeAccepted] issueReceipt(cardNumber, serialNumber, description, amount, balance, availableBalance)  
9.5.5.3.6. [envelopeAccepted] printReceipt(theATMnumber, theATMlocation, cardNumber, serialNumber, description, amount, balance, availableBalance)  
9.5.5.3.7. return Status::SUCCESS  

Figure 21 Reverse engineered Deposit scenario diagram
C.3 Use case Transfer

Figure 22 Transfer sequence diagram
Figure 23 Reverse engineered Transfer scenario diagram
C.4 Use case Inquiry

Figure 24 Inquiry sequence diagram
9.5.5.1.1.1. displayMenu (whatChoose, numItems, item[]):
  return choice
  9.5.5.1.1.3. return choice

9.5.5.1.2. return AccountType(choice-1):
  return Status::SUCCESS
  9.5.5.2. sendToBank():
    number():
      return _number
    PIN():
      return PIN
    cardNumber():
      return _cardNumber
    doInquiry(cardNumber, PIN, ATMnumber, serialNumber, from, newBalance, availableBalance):
      return Status::SUCCESS
      return _bank. doInquiry (_session.cardNumber(), _session.PIN(), _atm.number(), _serialNumber, _fromAccount, _newBalance, _availableBalance)

9.5.5.3. [code == Status:: SUCCESS]:
  finishApprovedTransaction()
  accountName(type):
    return accountName[type]
  9.5.5.3.1. return accountName[type]
  9.5.5.3.2. cardNumber():
    return_cardNumber
  9.5.3.2.1. return_cardNumber
  issueReceipt (cardNumber, serialNumber, description, amount, balance, availableBalance):
    printReceipt(theATMnumber, theATMlocation, cardNumber, serialNumber, description, amount, balance, availableBalance)
    9.5.5.3.4. return Status::SUCCESS

Figure 25 Reverse engineered Inquiry scenario diagram