Reverse Engineering of UML Sequence Diagrams Using Dynamic Information

By

Yucong Miao, B. Eng.

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements of the degree of
Master of Science in ISS

Information and Systems Science
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, K1S 5B6
Canada

February 2003

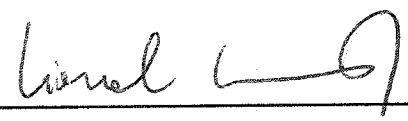
Your file Votre référence
ISBN: 0-612-83504-9
Our file Notre référence
ISBN: 0-612-83504-9

The undersigned recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis

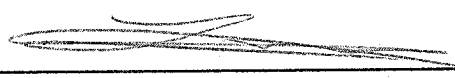
Reverse Engineering of UML Sequence Diagrams Using Dynamic Information

Submitted by
Yucong Miao, B. Eng.
in partial fulfillment of the requirements for
the degree of Master of Science

Chair, Professor Rafik A. Goubran



Thesis Co-Supervisor, Professor L. Briand



Thesis Co-Supervisor, Professor Y. Labiche

Carleton University
February 2003

ABSTRACT

Recovering dynamic models from run time information is an important aspect of software system maintenance and reverse engineering. One common way to represent the run time behaviors of objects in a system is to use UML sequence diagrams. Each execution of a use case scenario results into a scenario diagram, that is, an incomplete sequence diagram only modeling object interactions for one single scenario.

This thesis presents an approach to detect and decipher dynamic information of C++ programs and recover the corresponding scenario diagrams. The approach is composed of two parts: code instrumentation and model recovery. The methodology of code instrumentation, which is platform and compiler independent, is provided to collect dynamic information for sequential, non-distributed C++ program under the form of a trace file. A generic methodology of model recovery is introduced to analyze a trace file. The transformation of a trace file into a scenario diagram is formally defined using OCL expressions and UML class diagrams (metamodels to structure the information in trace files and scenario diagrams).

Prototype tools, for both code instrumentation and model recovery, are developed and used to illustrate the approach on a realistic case study.

Acknowledgments

I would like to thank my supervisors, Lionel Briand and Yvan Labiche, for their understanding, guidance, support, and feedback.

I would like also to thank my families for their continued support.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Software maintenance is becoming more and more important in software development cycle due to the rapidly increasing requirements in software engineering. But software maintenance, re-engineering and reuse are usually complex, costly and risky, especially for large software systems. However, these large systems, so-called legacy systems, perform a critical task for the enterprise and usually can't be discarded completely. Thus, the only really viable choice is a gradual technical improvement of the existing software system. However, before improvements to the software system are undertaken, system understanding should be addressed. The problem is that for most of legacy systems, understanding and reasoning about changes in the source code becomes very difficult due to incomplete or out-of-date documentation, or a lack of architectural view to assist in understanding the behavior of the system. Therefore, reverse engineering and design recovery techniques are needed in order to understand the current structure and dynamic behavior of a system.

In general, there are two types of reverse engineering techniques. One is static reverse engineering, which composes static models of applications from static information; the other is dynamic reverse engineering, which composes dynamic models of applications from dynamic information. Static models describe the static structure of a system the way it is written in the source code. It can't determine behavioral properties and there is no satisfactory way to trace the events during the execution of the system. Dynamic models, which present the dynamic aspect of systems, can be used to understand the run time behavior of the system.

The process of reverse engineering can be divided into two phases: the identification of low-level source code artifacts (code extraction/code instrumentation) to obtain the necessary information; and the analysis of the extracted information to compose high-level models, which includes retrieving and visualizing models [Mendonca+ 96].

On one hand, there is an increasing demand for an effective software analysis approach to help the user understand software dynamic behavior in order to adapt to rapidly changing and fast-growing software applications. On the other hand, there is no commercial software analysis tool that can use dynamic information to generate software interaction diagrams automatically to present software run time behavior. This thesis is fundamentally motivated by the challenge of analyzing large and complicated object-oriented software run time behaviors.

Compared with static reverse engineering, the dynamic reverse engineering of an object oriented software system holds more challenges. Dynamic information, which is obtained during the execution of a software program, is used to build dynamic models. Sometimes it is difficult to detect and decipher the dynamic information due to the characteristics of object-oriented software. Dynamic binding makes it hard to distinguish the behavior of an object, which is defined not only in its own class but also in its super classes. Polymorphism makes it difficult to determine which operation is actually executed at run time. Therefore, it is very difficult, and in most cases impossible, to understand the run time behavior of a software system just examining source code.

Different kinds of dynamic information can be applied to build different kinds of dynamic models. Dynamic models can be various according to the diverse requirements

of software behavior demonstration. For example, some dynamic models focus on presenting interactions between objects of the target system, while some are intended to summarize the relevant computation of the objects of the target for further investigation.

The Unified Modeling Language (UML) [UML] is one of the most common notations used to specify, visualize, understand and document OO software systems. It has been accepted as an industrial standard. UML provides several diagrams that can be used to view and model the software system. A sequence diagram is a UML behavioral model used to describe the object interactions of the target system, which are arranged in sequence. Typically, a sequence diagram is used to describe the event flow of a use case and identify the objects that participate in the use case. A use case can have several possible scenarios involving the described functionality [Bruegge+ 00]. We use the term scenario diagram to denote a sequence diagram that captures object interactions taking place in a single scenario of a use case.

This thesis provides a systematic solution to investigate which dynamic information is required to model a scenario diagram; extract necessary information from a batch of run time information of the target system and recover the corresponding scenario diagram. A scenario diagram depicts only one possible scenario of a use case of system, while a sequence diagram can depict all the possible scenarios of a use case. All the scenario diagrams of a use case can be grouped to form the corresponding sequence diagram. However, how scenario diagrams are automatically grouped to form sequence diagrams is out of the scope of this thesis. A related issue, which is also out of the scope of this thesis, is the use of functional testing techniques (Black box testing technique, such as category partition technique) to exercise entire functionalities, i.e., the selection

of test cases that can produce sets of scenario diagrams that, when grouped, produce relevant, hopefully complete sequence diagrams. Eventually, an automation tool is developed and implemented to realize our approach.

The dynamic reverse engineering is usually composed of two parts: modeling and graphic visualization. As modeling is the fundamental process in the reverse engineering procedure, our approach focuses on modeling sequence diagram other than visualizing sequence diagram in graphic appearance. Graphic visualization is the next phase and can be continued in our future work.

This thesis proposes an approach to presenting a systematic solution to detect and decipher the required dynamic information, and to retrieve the related sequence diagram of the target system. The contributions of this thesis are:

First, we analyze the characteristics of a scenario diagram, which can describe the dynamic behavior of the target system. The dynamic information required to generate a sequence diagram is identified. The challenge in detecting and deciphering the dynamic information is discussed.

Second, a code instrumentation methodology is provided to augment C++ source codes with trace-generating statements: Run time information concerning operations, return statements and control flow structures will be detected during the execution of the instrumented programs. The methodology is defined for C++ programs, but nothing is really specific to C++. Therefore, it can be easily applied to Java programs.

Third, a generic and systematic modeling methodology is offered to interpret dynamic information and reverse-engineer a scenario diagram. Interactions between objects are recognized and deciphered from a batch of run time information. The

information about conditions, which must be fulfilled to send/receive messages, is also detected. The design and implementation of the methodology is separated from the code instrumentation methodology. Although it is designed to reverse-engineer sequence diagrams from C++ programs, it can be applied to retrieve a scenario diagram model for a software system implemented using any object-oriented programming language as long as the dynamic information has been collected and recorded with a pre-defined format.

Fourth, an automation tool is developed and implemented to prove the feasibility of our methodologies. Java [J2SE™ 1.4] and Perl [Christiansen 98, Wall 00] interpreters, which are freely available, are required. Although the retrieved sequence diagram is presented in a human readable text format rather than a graphic visualization format, it displays an entire trace and provides a depiction of the behavior of the target system. In the future, it also can be transformed into a XML-based Metadata Interchange (XMI) file, which is able to interchange metadata between UML modeling tools easily [XMI], and then imported in any UML case tool to visualize the sequence diagrams with graphic format.

Fifth, the tool has been used to retrieve scenario diagrams (corresponding to four different use cases) of an Automated Teller Machine (ATM) system that has not been developed by the author. The ATM system was provided with UML sequence diagrams, thus allowing us to verify (oracle) the methodology and the tool.

The rest of the thesis is organized as follows: related work is reviewed in Chapter 2. The methodology of our approach is presented in Chapter 3. Chapter 4 introduces the main feature of an automation tool as well as detailed design and implementation aspects.

A case study is examined in Chapter 5. In Chapter 6, the thesis ends with a conclusion and a discussion of future work.

Chapter 2 Related Works

A great deal of effort is made in the realm of program understanding and reverse engineering. In this section, we review and evaluate approaches that support the dynamic reverse engineering of object-oriented software systems by constructing dynamic views of the target system using dynamic information.

The process of dynamic reverse engineering can be divided into two parts: extracting dynamic information and retrieving dynamic model. Although emphasis of approaches varies, dynamic information is used to analyze and present the behavior of the target system. No matter what kind of dynamic model is to be retrieved, there are basically three ways of collecting dynamic information at run time [Lange+ 95a]: (1) *Compiler-extended technique* (Section 2.1): extend or modify the compiler so that trace-generating statements can be added into source codes; (2) *Debugger-based technique* (Section 2.2): use debugging techniques, set breakpoint at corresponding locations in a program; (3) *Program-instrumentation-based technique* (Section 2.3): augment source code with trace-generating statements. Some typical approaches are briefly examined and assessed in the following sections according to the above dynamic information collection strategies.

We then summarize, and compare the approaches according to different criteria (Section 2.4), and briefly mention what functionalities existing case tools provide with respect to the problem addressed in this thesis (Section 2.5).

2.1 Compiler-extended techniques

Walker's approach [Walker+ 98] analyzes collected dynamic information in order to determine how many objects of a class might exist at run-time and how many operations might occur between particular objects. The collected dynamic information includes (1) the class of the object executing an operation or having an object created; (2) the class of the object and operation being executed, or the class of object being created or deleted. The approach is implemented using Smalltalk programming language. The Smalltalk virtual machine has been instrumented to record the desired dynamic information. Therefore, only dynamic information of Smalltalk programs can be gathered and analyzed.

In this approach, the total number of objects of a class, which has been allocated and destroyed at a particular point in the system's execution, is presented. The total execution times of an operation between two classes at the corresponding execution point is calculated and displayed. The approach visualizes the dynamic information using a set of figures. Each figure displays dynamic information representing both a particular point in the system's execution and the history of the execution to that point. Within a figure, each class is represented by a box. Each box is annotated with numbers and bar-chart style of histogram, which indicate the total number of objects that map to the box that have been allocated and de-allocated until the current point in system's execution. A directed arc between two boxes indicates that an operation on an object in the destination box has been invoked from an operation on an object in the source box. Multiple instances of interaction between two boxes are shown as a number annotated the directed arc.

Although it analyzes the dynamic information about objects and the interactions between objects, the approach does not describe interactions and the sequence of those interactions in detail. The approach focuses on performance evaluation and enhancement of the target system rather than system understanding.

Tamar Richner and Stephance Ducasse [Richner+ 02] present an approach, which uses dynamic information to support the recovery and understanding of collaborations and roles. A tool named Collaboration Browser was developed to validate the approach. Terms *collaboration instance* and *collaboration pattern* are brought in. A collaboration instance is the sequence of messages sent between objects, ordered as a call tree, which results from a method invocation. A collaboration pattern is an equivalence class of several collaboration instances. A collaboration pattern is an approximation to the high-level design concept of collaboration. The corresponding approximation to the high level notion of roles is the set of public methods of a class which are presented in the context of a collaboration pattern.

The sender class, sender unique identity (the object's memory address), receiver class, receiver unique identity and name of the invoked method (not the whole signature) are detected for every method invocation. Pattern matching is used to find similar collaboration instances and group them into a collaboration pattern. Patten matching criteria, which specify what it means for two collaboration instances to be considered equivalent, are defined by user in advance. Applying the pattern matching criteria, the execution trace, which can be seen as an ordered call tree, is traversed in a bottom-up fashion to abstract the collaboration patterns. Once pattern matching has been performed, the dynamic information is presented in terms of classes, methods and collaboration

patterns. The user can query about collaboration patterns from the information of participate senders, receivers and invoked methods, and vice versa. The role (a set of methods) of a receiver class in a collaboration pattern can be queried as well.

Collaboration Browser is implemented in Smalltalk and currently handles single-thread Smalltalk applications. Smalltalk virtual machine has been extended to instrument the applications. Therefore, the approach is not suitable for a programming language without virtual machine. Collaboration Browser focuses on understanding a small chunk of interactions and the roles what the classes play in the interactions. An overall behavior view of an application is not provided. The dynamic information of control flow structures in source code is not collected in this approach. Therefore, conditions of interaction executions of an application, which are important to describe the behavior of the application properly, are no way to be presented in this approach.

2.2 Debugger-based techniques

SCED (SCenario EDitor) is a dynamic reverse engineering technique for modeling the run time behavior of Java software systems [Systä 99a, Systä 99b, Systä 00]. The dynamic event trace information is generated as a result of running the target system under a customized java debugger. The debugger has been implemented using public classes belonging to sun.tools.java and sun.tools.debug packages, which are included in jdk1.2. To capture the dynamic information, breakpoints are set in the debugger. Breakpoints are set at the first line of all methods and control flow structures in order to detect the dynamic information. However, when the dynamic information is collected, the system spends a great deal of time in context switching between the process

of the target program and the process of the debugger. The mechanism of this instrumentation tool means that it can only be applied to Java software systems.

Message sequence charts with extended notations are used to describe the behavior of a Java system in SCED. When the trace information is loaded into SCED, the senders, receiver, operations, control flow structures and repetition of message structures are detected and presented in the sequence diagrams. Instead of being labeled by the actual condition, a condition structure is labeled by the line number in the source code.

2.3 Program-instrumentation-based techniques

In this section, two approaches Program Explorer and ISVis, which obtain dynamic information using Program-instrumentation-based technique, are reviewed.

Program Explorer [Lange+ 95a, Lange+ 95b] is developed by Danny B. Lange and Yuichi Nakamura as an understanding tool for C++ programs. It has an instrumentation utility that augments C++ programs with code that produces trace information, and a trace recorder that captures traces. Note that the instrumentation does not report on conditions that trigger operations. A visualization component has been developed to present interactions of objects.

Program Explorer tracks dynamic information concerning function invocation and object instantiation. Member functions, which are called during the program execution, are observed and analyzed. The sequence, in which member functions are called, is also notified. Program Explorer is designed to retrieve interactions in which objects of a given class (provided by the user) and its subclasses are involved. This helps to find interaction patterns, that is, interactions that happen several times during an execution of the system. However, the user must know the interaction pattern which he/she is looking for

beforehand, and have an idea where that pattern occurs (i.e., which classes are involved) in order to exploit the visualization. In addition, since only one class can be selected at a time, 'complete' interaction sequences (e.g., corresponding to a use case scenario) cannot be retrieved.

The trace-generating statement fragments, which are added by the instrumentation utility of Program Explorer, are applicable for tracing C++ programs. The instrumentation phase is platform and compiler independent. However, Program Database, which is part of Program Explorer and provides the query mechanisms mentioned above, relies on files generated by the IBM's xlC compiler, making the whole approach compiler dependent.

ISVis (Interaction Scenario Visualizer) [Jerding+ 97a, Jerding+ 97b] is a visualization tool presented by Jerding in 1997. ISVis has an instrumentation tool which takes the source code, the static information file, and information supplied by the user about classes. Like Program Explorer, the instrumentation tool also augments C++ or C programs with trace-generating statements to obtain run time information. When the instrumented system is executed, traces are generated. A trace analyzer reads the trace information and converts it into scenarios. Message sequence charts have been used to visualize the scenarios.

ISVis uses Perl script language to implement the code instrumentation tool. C++ or C programs can be instrumented even though they may have been developed under different platforms. Although it can provide message sequence charts to show the behaviors of a system, ISVis does not look after the effect of control flow structures in program execution, which is useful for understanding the behaviors of the target system.

For example, a message is sent only when a condition is fulfilled. The repeated messages without condition description can be shown in message sequence chart view sequentially. However, the abstraction of repeated messages is not applied in ISVis. Also, without condition description, sometimes it is hard to tell whether a group of repeated messages are executed under exact same condition.

2.4 Summary

We can compare the above strategies based on the following aspects:

1. Conditions of interactions: are the conditions of interaction executions handled?
2. Iteration of messages: is iteration of a single message distinguished from a group of messages?
3. Instrumentation technique: what kind of instrumentation technique is applied, according to the categories described above?
4. Recovered dynamic model: what kind of dynamic model does the approach use as an output?
5. Applied programming language: what O-O programming language can the approach be applied to?
6. Extensibility to other languages: is the approach extensible to apply to other programming languages?
7. Abstraction of sequences from scenarios: can the approach abstract sequence diagrams from scenario diagrams?

Table 1 below summarizes the related works according to the above aspects.

	Walker's approach	Collaboration Browser	SCED	Program Explorer	ISVis
Conditions of interactions	No	No	Yes	No	No
Iteration of messages	No	No	Yes (but not in UML notation)	No	No
Instrumentation technique	Compiler-extended technique	Compiler-extended technique	Debugger-based technique	Program-instrumentation-based technique	Program-instrumentation-based technique
Recovered dynamic model	Histograms view of object interaction	Approximation of Collaboration	Message sequence chart	Design pattern view	Message sequence chart
Applied programming language	Smalltalk	Smalltalk	Java	C++	C++
Extensibility to other languages	No	No	No	Yes	Yes
Abstraction of sequences from scenarios	No	Yes	No	No	No

Table 1 Summary of Related Work

Our approach is to recover sequence diagrams using dynamic information. Our instrumentation tool, named InstrTC++, augments C++ source codes with trace-generating statements for every method, control flow structure and return statement. Since it is independent of C++ programs, our instrumentation tool can instrument any C++ software system, which may be implemented under various platforms. With a little modification, InstrTC++ can instrument Java programs as well. When the instrumented program is executed, traces are captured automatically.

Through an analysis of the traces, a sequence of system behaviors can be described by a set of messages. The conditions of the messages are also detected during

the analysis. The detailed conditions are described with messages in order to make the messages more descriptive. A condition of a message can be composed by more than one condition clauses due to the nested control flow structures. In our approach, the way of presenting repetition differs between a single message and a group of messages.

Since the code instrumentation and dynamic information recovery are designed and implemented separately, the methodology of the dynamic information recovery can be applied to recover the sequence diagram of sequential, non-distributed software system from dynamic information with a pre-defined trace format.

2.5 Reverse-engineering dynamic models with existing case tools

To the best of our knowledge, no commercial case tool allows the reverse-engineering of dynamic models (and in particular, sequence diagrams) from dynamic information (traces). Some of them do not support at all the reverse engineering of dynamic models (e.g., [Rational Rose]). Others, such as Together [TogetherSoft], attempt to generate sequence diagrams based on the reverse engineering of static information.

Together allows the user to generate a sequence diagram based on the source code. Indeed, reverse-engineering the source code can provide information on the operations between objects, and the conditions under which those calls would be performed if the system were executed.

However, the lack of dynamic information prevents Together from identifying the actual operations that would be executed, which are the operations whose targets are decided at run time due to polymorphism and dynamic binding. The only calls that can be detected by Together are those performed on parent classes, as specified in the source code, and not those actually performed on child classes.

Chapter 3 Modeling Scenario Diagram from Run Time Information

Our objective is to recover the scenario diagram of a target system from a batch of trace information. In this chapter, UML class diagrams [Bruegge+ 00], which represent the structure of the system in terms of objects, classes, attributes, operations and associations, are used to model traces (section 3.2.3) and scenario diagrams (section 3.1.2). The transformation rules from a trace file to a scenario diagram (i.e., from an instance of the trace metamodel to an instance of the scenario diagram metamodel) are expressed in Object Constraints Language (OCL) [Warmer+ 99], as well as regular expressions. OCL is a logical choice as it comes with UML. Using OCL provides a way to navigate through a model in order to unambiguously identify the relationships that can exist between model elements. OCL also provides collection types and operations to retain objects and describe their behaviors.

Instead of providing a GUI with a graphic visualization format, we have presented the result of the analysis in an understandable and readable text format. The text format can be transformed further into a XMI file [XMI] and imported to any UML case tool.

3.1 Scenario Diagram

In this section, we study the UML sequence diagram and present our adaptation of a UML scenario diagram meta-model.

3.1.1 UML Sequence Diagram and Scenario Diagram

A sequence diagram [Bruegge+ 00] is an important UML behavioral model, which shows how objects interact with each other and provides an efficient way to formalize the behavior of the system and to visualize the communication between objects. It illustrates how messages are sent and received between objects. An object interacts with other objects by sending messages. The reception of a message by an object triggers the execution of an operation, which in turn may send messages to other objects. Each object is typically a named or anonymous instance of a class and is depicted as a column in the sequence diagram. Each message is depicted as an arrow between two columns. Conditions may be applied to the messages. A condition must be true for the message to be sent and received. Iterations can be used to help describe the messages, which are executed repeatedly due to a loop condition structure. Arguments may be passed along with a message and are bound to the parameters of the executing operation in the receiving object. Two operations with different signatures are different even if they have the same message name. Messages are illustrated sequentially according to the execution orders [Erikssion+ 98]. Typically, we use a sequence diagram to describe the event flow of a use case and identify the objects that participate in the use case [Bruegge+ 00].

A sequence diagram can be used to describe either an *abstract sequence* (i.e., all possible interactions) or *concrete sequences* (i.e., one possible interaction or scenario) [Bruegge+ 00]. We call a concrete sequence diagram a scenario diagram. Different scenarios are differentiated by conditions. Although a scenario diagram normally doesn't show any condition in forward engineering, it is argued that describing conditions with

scenario diagrams is helpful for understanding the behavior of the target system in reverse engineering. Our approach focuses on reverse engineering scenario diagrams using run time traces.

3.1.2 Adaptation of Scenario Diagram Metamodel

Our adapted scenario diagram (referred to hereafter as scenario diagram for simplicity) is composed of a set of messages and objects, which participate in the interactions. Messages consist of method messages and return messages. The participating objects represent the senders and receivers of messages. A method message can be sent along with parameters. Other messages can be triggered by a method message. The conditions and iterations of messages are described as well. However, our scenario diagram doesn't include all the concepts and notations of UML sequence diagrams. For instance, we are not interested in active and passive objects. We have developed a scenario diagram (SD) metamodel, which adapts to the UML sequence diagram metamodel in our approach.

Figure 1 shows the class diagram of our SD metamodel. The SD metamodel has an abstract class Message, which uses the attribute content to present the detailed statement of an interaction message, such as method name. The Message class has three derived classes: the MethodMessage class, ReturnMessage class and IterationMessage class. The MethodMessage class represents messages which are triggered by the execution of operations. The attribute timesOfRepeat describes number of times of the operation execution, if iteration exists. A method message is performed once without any iteration if the value of timesOfRepeat is zero as default. If the value of timesOfRepeat of a MethodMessage object is greater than zero, the iteration of the operation must be

observed. The ParameterSD class represents the formal parameters of method messages, which are passed along with operations.

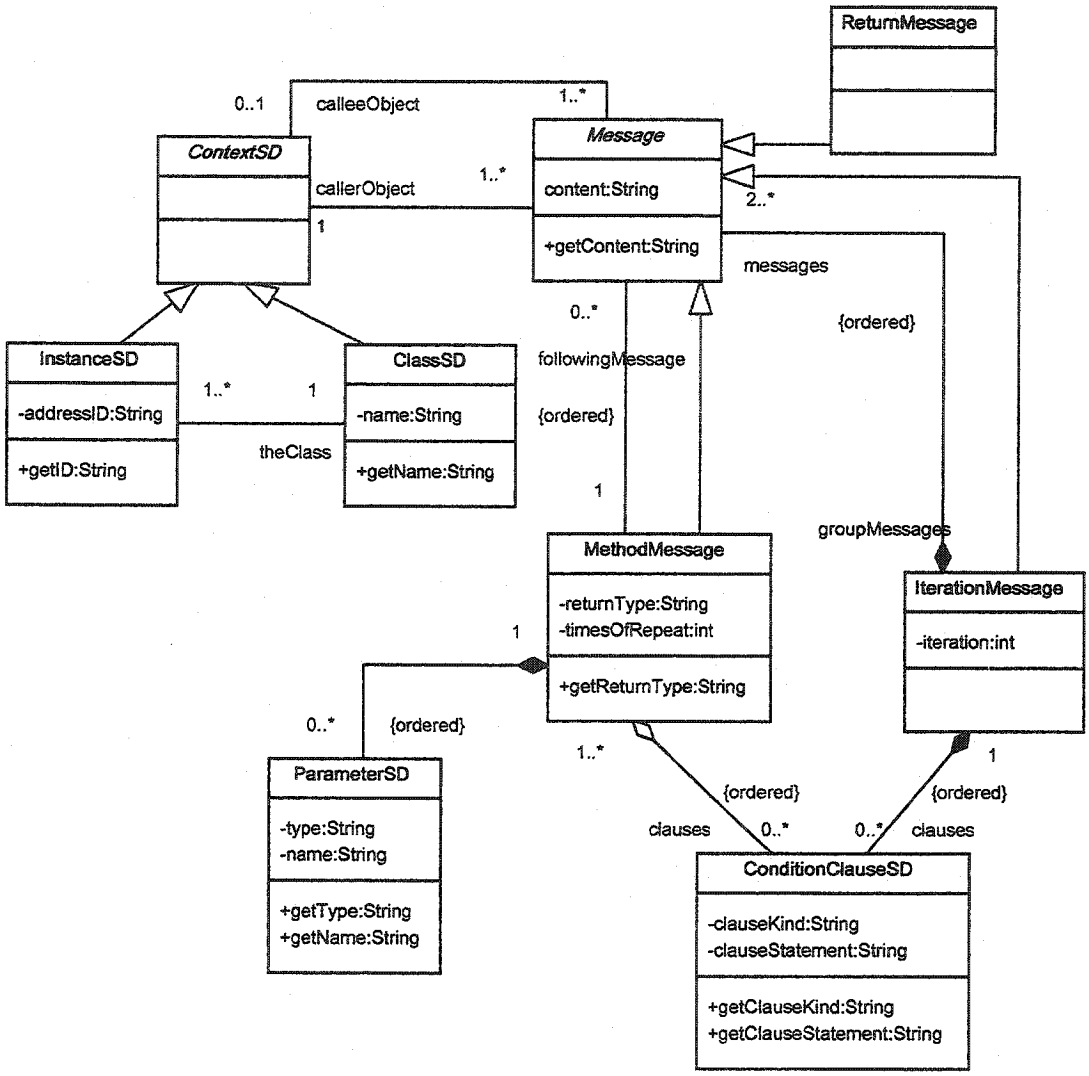


Figure 1 Class Diagram of SD Metamodel

A method message or a group of iterative messages may be sent under a particular condition. The messages can be sent only when the condition is satisfied. A condition can be composed of one or more condition clauses, which are represented by the `ConditionClauseSD` class. The attributes `clauseStatement` and `clauseKind` describe the statement and type of a condition clause.

The navigation followingMessage indicates that other messages can be invoked by the method message. The messages is the role played by Message to represent the messages which are sent iteratively in a group. The ReturnMessage class declares the return messages.

In UML, the way to illustrate the iteration of a single message is different from the way to illustrate the iteration of a group of messages. In order to present a group of iterative messages clearly, the IterationMessage class is introduced. The IterationMessage class implies that a group of iterative messages exists. An IterationMessage object is composed of two or more messages, which could be MethodMessage or IterationMessage. The attribute iteration describes the execution times of the group of messages. The groupMessags is the role played by IterationMessage to represent a group of iterative messages.

The ContextSD class describes the callers and callees of messages. The ContextSD class represents the context of objects in which messages are sent. The concrete classes InstanceSD and ClassSD are sub-classes of the abstract class ContextSD. The attribute addressID of the InstanceSD class gives a unique ID of an object. A ClassSD object describes the class of an InstanceSD object. The callerObject and calleeObject are the roles played by Context to represent the caller object and callee object of a message respectively. The theClass is the role played by ClassSD to represent the class of the object which participates in an interaction in the target system.

3.2 Source Code Instrumentation

Source code instrumentation is the first step in our reverse engineering process. As described in section 3.1, a scenario diagram is composed of a set of messages and participating objects. Events such as operation invocations and return statement executions are presented by message interactions depicted in the scenario diagram. Since they may affect the execution of operations, the executions of control flow structures in the source code need to be detected as well. Our methodology of code instrumentation is presented in the following section.

3.2.1 Instrumentation Strategy

The information about the message, the sender object and receiver object of the message and the condition of the message sending is required to describe an interaction. A series of trace-generating statements are inserted into the source code to capture the run time information.

The execution of an operation is expressed as a message in a scenario diagram. During the execution of a program, every operation has a chance to be invoked. Therefore, each method needs to be instrumented in order to capture every operation invocation. Trace-generating statements are added to the definition body of a method to identify the operation executions and ascertain the duration of the operation execution. Once the operation has been invoked, logs are generated.

Static methods are independent of any object of a class. Since we are interested in interactions between objects only, the execution of static methods (except constructors) won't be investigated. Therefore, we don't instrument static methods except constructors.

Parameters, which are bounded to the parameters of an executed method, can be passed along with a message. There are two kinds of parameters: formal parameters and actual parameters. Formal parameters of a method, which are specified at compile time, are declared in the method signature. The information about formal parameters can be obtained by analyzing the method signature. In our approach, we illustrate the formal parameters of a method message in a scenario diagram. Actual parameters of a method are real values passed to the method. Although using an actual parameter to describe a message is more precise, complex static information analysis is required to extract the information of the actual parameters. For example, we need to detect operations, and the type of the object on which these calls are performed, which is known to be a hard problem in object-oriented systems.

A return message can also be depicted in a scenario diagram. The trace-generating statements are added to track the executions of the return statements. After the execution of a return statement, the operation which contains the return statement is executed completely.

Messages are sent between objects. Therefore, identifying caller and callee objects is essential in describing a message. To uniquely identify an object in a scenario diagram, a unique identifier, such as a literal name, should be applied to the object. In order to distinguishing objects, a developer can assign different names to them on purpose. However, things are different in the reverse engineering process. A literal name may not be always unique to represent an object.

For example, we have class A, B, C in a program. The class diagram is shown in Figure 2. Both class B and C has a data member *a1* which is an instance of class A. The

constructor code fragments of class B and C are also described in Figure 2. When an object (e.g., *b*) of class B is instantiated, an object *a1* (to which the value of "theA1" attribute is assigned) of class A is provided. Similarly, an object *a1* (to which the value of "theA2" attribute is assigned) of class A is provided when an object *c* of class C is instantiated. The two *a1* objects are the same object only if "theA1" and "theA2" are identical. However, if "theA1" and "theA2" objects are referred to two different objects, the two *a1* are different as well. Therefore, whether two objects of a class (with the same literal name) are identical depends on the detailed source code. Objects may not always be distinguished only by literal names (such as *a1*) in reverse engineering.

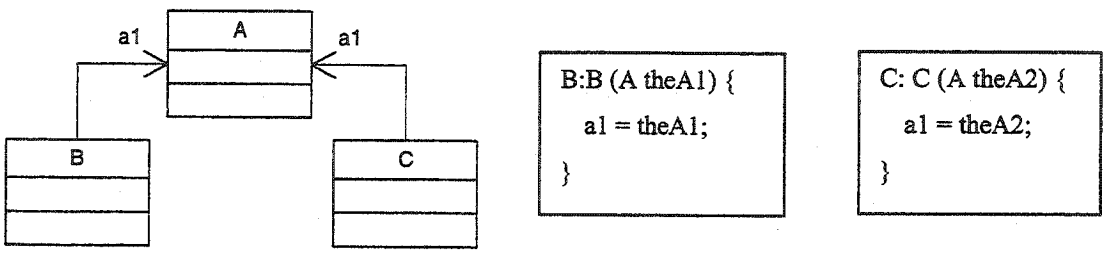


Figure 2 An example illustrating the insufficiency of using an object literal name in the reverse engineering process

In our approach, objects are identified by a unique physical memory address. Since each object is assigned to a unique memory address during the execution of the program, objects can be distinguished in this way.

If a message has a condition, the message can be sent only when the condition has been fulfilled. Therefore, it is necessary to illustrate the conditions in the scenario diagram as well. A condition is produced by the execution of the control flow structures in a program. Therefore, the control flow structures in a program need to be detected as well.

The iteration of messages is depicted in the scenario diagram as well. The method of presenting the iteration of a single message is different from that of the iteration of a group of messages in UML. Distinguishing and processing the two kinds of message iteration is not dealt with by our source code instrumentation methodology. This problem will be solved in our scenario diagram retrieval process. Repeated traces are recognized and presented by the corresponding message (or a group of messages) with iteration. We will discuss this in detail in section 3.3.

3.2.2 Trace File

A trace file is generated during the execution of an instrumented program. Dynamic information, which presents the behavior of objects, is recorded in this trace file. It indicates a set of ordered object interactions.

A trace file is composed of many log lines. Each log line indicates that a particular statement in the source code has been executed, and provides information about that statement. Each log line contains information about the object that participates in this statement execution as callee, the type of the execution statement, details on the statement that is executed, and the class where the statement is defined as well. According to the instrumentation strategy demonstrated in section 3.2.1, the following four types of information are detected and collected in the log lines:

* **Object ID**: a string (the memory address) uniquely representing the object on which a statement is executed.
* **Class name**: name of the class where an execution statement is defined.
* **Kind of statement**: a string indicating the type of the statement structure which is detected. It can be a "Method Entry", to indicate that an operation

execution is beginning; "Method Exit" to indicate that an operation execution is completing; "Return" to indicate that a return message is executing; a word to indicate the execution of a control flow structures, such as "If" or "EndControl" to indicate that the execution of a control flow structure is completing.

* **Execution statement**: the statement to be traced. It can consist of the signature of a method, the statement of a return message, the condition expression of a control flow structure, and a string to indicate the end of the control flow structure, such as "end of control flow", "breakLog" and "continueLog".

The information about an interaction and its callee object can be obtained from the content of a log line directly, whereas the information about the caller object of an interaction must be acquired from the related previous log line. For example, two log lines L1 and L2 in a trace file described in Figure 3:

L1: "0x123450","classA","Method Entry","mA"

L2: "0x543210","classB","Method Entry","mB"

Figure 3 Examples of Log Lines

By analyzing log line L2, we know that method `mB`, which is defined in the definition of `classB`, has been executed on callee object `"0x543210"`. L1 and L2 indicate that method `mB` is invoked inside the definition body of method `mA`. Therefore, we can tell that the caller object of `mB` interaction is object `"0x123450"`, which is also the callee object of the `mA` interaction.

A loop control flow structure, such as a "for", "while" and "do/while" structure, will be executed more than once as long as the corresponding condition is fulfilled. Therefore, methods invoked in the loop control flow structure may be executed more than once. The corresponding repeated logs are recorded in an orderly fashion. Those logs indicate the iteration of messages. These rules will be discussed in detail in section 3.3.

A control flow structure may be nested within another control flow structure. Both can affect the execution of an operation which is nested in their definition bodies. Therefore, a condition is composed of one or more condition clauses, which are condition statements of control flow structures. For example, we have a class named CC. Figure 4 shows the fragment of source code (defined in class CC) and the corresponding traces.

```
if (a==2) {

    if (b+1>4) {
      m1();
    }
}
```

```
"0x123450","CC","If","a==2"
"0x123450","CC","If","b+1>4"
"0x123450","CC","Method Entry","m1()"
"0x123450","CC","Method Exit","m1()"
"0x123450","CC","EndControl","end of control flow"
"0x123450","CC","EndControl","end of control flow"
```

Figure 4 Source code and traces of a nested control flow structure of a small example

Since the execution of m1 is affected by condition statements "a==4" and "b+1>4", the condition of m1 is "a==4 **and** b+1>4", which is the combination of two condition clauses.

3.2.3 Trace Metamodel

As described above, dynamic information is contained in a trace file with a text format. Each log line represents the information about a detected execution. A metamodel named TRACE is designed to interpret dynamic information into an object-oriented paradigm and store the dynamic information.

Figure 5 shows the class diagram of the TRACE metamodel, which is designed to store the dynamic information obtained from the log lines in a trace file.

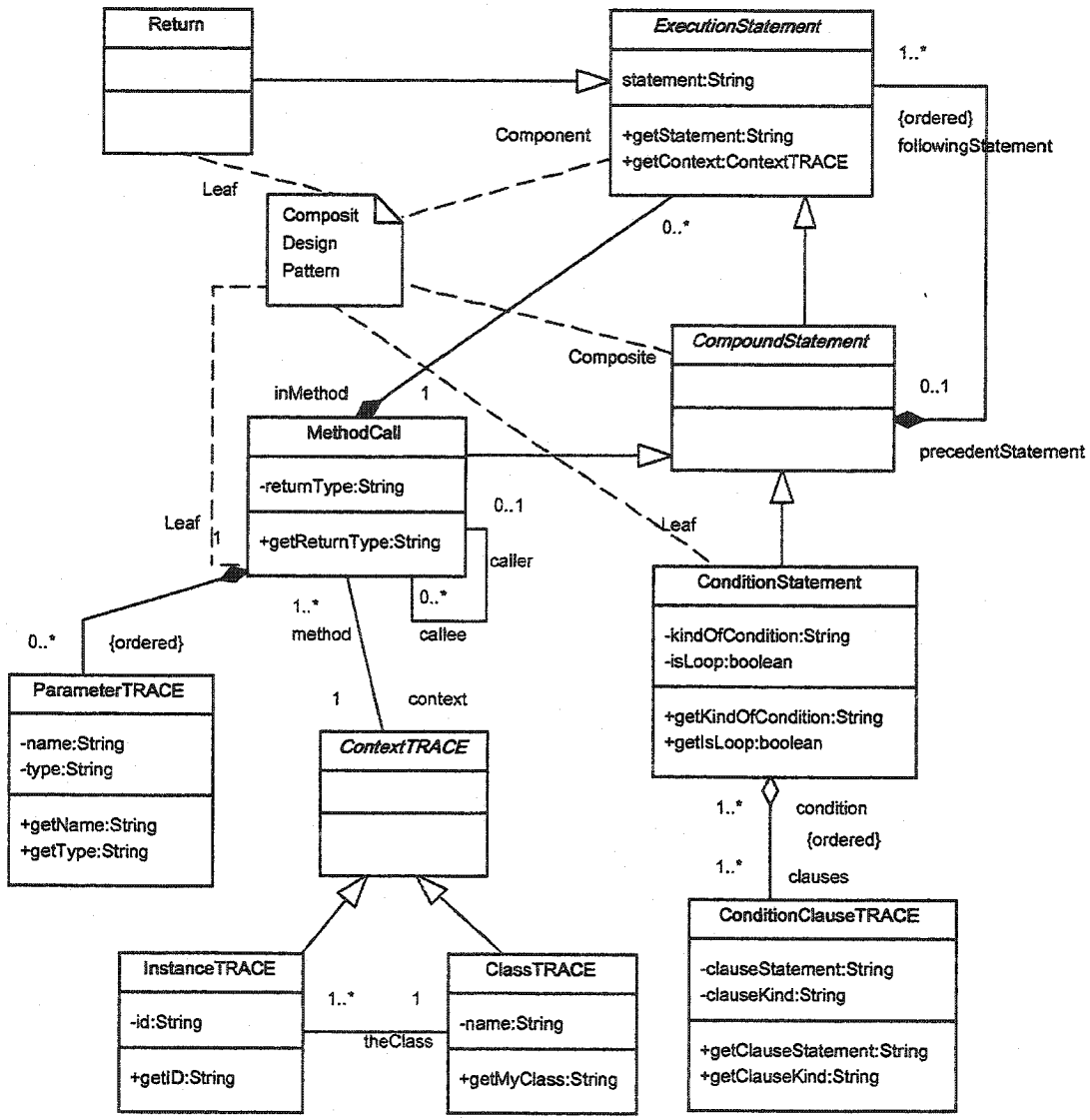


Figure 5 Class Diagram of TRACE Metamodel

The `ExecutionStatement` and `CompoundStatement` class are abstract classes. Every executed statement is an `ExecutionStatement`, which has an attribute `statement` to represent the content of the executed statement, such as the method name of a method, the return value of a return statement and the control structure.

An ExecutionStatement can be further specified as a return statement or a compound statement. A return statement, which is an object of the Return class, is an executed statement that is not followed by any execution statement. Unlike a Return object, CompoundStatement, an abstract subclass of ExecutionStatement, represents an executed statement, which may be followed by other executed statements. Each ExecutionStatement is associated with a CompoundStatement object. The attribute executionStatementSeq is a collection designed to collect all the followed executed statements of a CompoundStatement.

The MethodCall class and ConditionStatement classes are concrete subclasses of CompoundStatement. The MethodCall class represents operation executions and the ConditionStatement class represents the execution of the control flow structures. Each operation is made within a context, which is either an object or a class. The ContextTRACE class is designed to represent the context of an operation. It is an abstract class, which is inherited by the concrete classes InstanceTRACE and ClassTRACE. The attribute id of the InstanceTRACE class gives a unique ID to an object, which is the physical memory address of the object inside a computer. It can be used to identify an object during program execution. The class of an InstanceTRACE object is represented by the ClassTRACE object.

An interaction may be performed under a particular condition. The ConditionStatement class represents a condition which affects other execution statements contained in its executionStatementSeq collection. A condition is composed of condition clauses. The condition is true only if all the condition clauses are true. Each condition clause is created to present an execution of a control flow structure.

The ConditionClauseTRACE class represents the clauses of conditions. The attribute clauseStatement and clauseKind represent the control statement and type of condition clause. The attribute kindOfCondition describes the type of the last condition clause of a ConditionStatement object.

If the execution statement is an operation, some arguments may be passed along. In a scenario diagram, a method message can be shown with its actual parameters. Because of the mechanism of the code instrumentation used to data mine the dynamic information, we are not able to obtain the actual parameters of a method. Instead of leaving the parameter field of a method vacant, we use formal parameters to help describe the method. The ParameterTRACE class describes the information concerning a formal parameter of a method, such as parameter name and type.

The operation getContext returns the context in which an executed statement is made. The getConditionClauses returns a collection of the condition clauses of an executed statement. The navigation caller and callee on the MethodCall class represent the caller and callee methods of an interaction.

The followingStatement is the role played by ExecutionStatement to represent an execution statement which follows from another execution statement. The precedentStatement is the role played by CompoundStatement to represent a compoundStatement which is followed by other execution statement. The condition is the role played by ConditionStatement to represent a condition statement which can have more than one condition clause. The clauses is the role played by ConditionClauseTRACE to represent condition clauses which can be part of a condition. The method is the role played by MethodCall to represent an operation

which is executed on an object. The inMethod is the role played by MethodCall to represent an operation in which other statements may be executed. The `context` is the role played by `ContextTRACE` to represent the context in which an operation is made. The theClass is the role played by ClassTRACE to represent the class of the object on which an operation is executed.

A composite design pattern, which represents part-whole hierarchies and describes how to use recursive composition [Gamma+ 98], is applied to the `TRACE` metamodel. The `ExecutionStatement` class is the component class of the composite design pattern, which declares an interface for objects in the composition. The `CompoundStatement` class is a composite class of the composite design pattern. It defines an aggregate of `ExecutionStatement` objects and stores leaf components. The `MethodCall`, `ConditionStatement` and `Return` classes are leaf classes of the composite design pattern. They define behaviors for primitive objects in the composition.

3.3 Transformation Rules between TRACE and SD

All the executed operations, return statements and condition statements are stored in the `TRACE` metamodel. Messages, composed of method messages, return messages and groups of iterated messages, are stored in the `SD` metamodel. In this section, we describe the transformation rules between the `TRACE` and the `SD` metamodels using OCL, and provide an abstract example for each rule. Using OCL to express the transformation rules is useful to check the completeness and correctness of metamodels. The transformation rules expressed in OCL can also be considered as a specification for designign effective algorithms.

3.3.1 Rule 1 - Identifying a method message

In order to describe a `MethodMessage`, we need information such as method name, caller and callee objects, condition and parameters of the method. The information is stored in a `MethodCall` object, which is created after the corresponding log lines have been analyzed. In this section, we describe the method message transformation rule for defining the method of transforming the information from a `MethodCall` object of the TRACE metamodel to a `MethodMessage` object of the SD metamodel. Figure 6 below shows the rule for transforming method messages between the TRACE metamodel and the SD metamodel.

```
1   methodCall.allInstances-> forAll(m1,m2: MethodCall | m1.callee->includes(m2)
2       implies
3       methodMessage.allInstances->exists (mm:MethodMessage |
4           // obtain the method name
5           mm.content = m2.statement
6           and
7           // obtain the caller object of message mm
8           if m1.context.oclType = InstanceTRACE
9           then (
10              mm.callerObject.addressID =m1.context.id
11              and
12              mm.callerObject.theClass.name = m1.context.theClass.name
13          )
14          else (
15              mm.callerObject.name = m1.context.name
16          )
17          and
18          // obtain the caller object of message mm
19          if m2.context.oclType = InstanceTRACE
20          then (
21              mm.calleeObject.addressID = m2.context.id
22              and
23              mm.calleeObject.theClass.name = m2.context.theClass.name
24          )
25          else (
26              mm.calleeObject.name = m2.context.name
27          )
28          and
29          // obtain return type of the operation m2
30          mm.returnType = m2.returnType
31          and
32          // obtain the parameters of the method
33          mm.parameterSD->forAll(index:Integer |
34              mm.parameterSD->at(index).name = m2.parameterTRACE->at(index).name
35              and
36              mm.parameterSD->at(index).type = m2.parameterTRACE->at(index).type
37          )
38          and
39          mm.parameterSD->size = m2.parameterTRACE->size
40          and
41          // obtain the condition clauses of the operaiton
42          if m2.precedentStatement.oclType = ConditionStatement
43          then (
```

```
44                  mm.clauses->forAll(index:Integer |
45                      mm.clauses->at(index).clauseStatement =
46                        m2.precedentStatement.clauses->at(index).clauseStatement
47                      and
48                      mm.clauses->at(index).clauseKind =
49                        m2.precedentStatement.clauses->at(index).clauseKind
50                  )
51              )
52              and
53              mm.clauses->size = m2.precedentStatement.clause->size
54              and
55              // obtain the number of executon times of the operation
56              if (m2.precedentStatement.oclType = ConditionStatement
57                  and
58                  m2.precedentStatement.isLoop = true
59                  and
60                  m2.precedentStatement.followingStatement->size =1)
61              then
62                  mm.timesOfRepeat =
63                      methodCall.allInstances->select(m3:MethodCall|
64                          m3.statement = m2.statement
65                          and
66                          m3.returnType = m2.returnType
67                          and
68                          m3.caller = m2.caller
69                          and
70                          m3.parameterTRACE->forAll(index:Integer |
71                              m3.parameterTRACE->at(index).name =
72                                  m2.parameterTRACE->at(index).name
73                              and
74                              m3.parameterTRACE->at(index).type =
75                                  m2.parameterTRACE->at(index).type
76                          )
77                          and
78                          m3.parameterTRACE->size= m2.parameterTRACE->size
79                          and
80                          m3.precedentStatement.oclType = ConditionStatement)
81                          and
82                          m3.precedentStatement.clause->forAll(index:Integer |
83                            m3.precedentStatement.clause->at(index).clauseStatement
84                              =m2.precedentStatement.clause->at(index).clauseStatement
85                            and
86                            m3.precedentStatement.clause->at(index).clauseKind =
87                                m2.precedentStatement.clause->at(index).clauseKind
88                          )
89                          and
90                          m3.precedentStatement.clauses->size =
91                                          m2.precedentStatement.clauses->size
92                      )->size
93              and
94              // obtain the following message of message mm
95              mm.followingMessage->forAll(mm1:Message |
96                  mm1.callerObject = mm.calleeObject
97              )
98          )
99  )
```

Figure 6 Transformation Rule to Identify Method Messages

From the first line to the third line in Figure 6, we illustrate that if there is a `MethodCall` object `m2`, which is invoked by `MethodCall` object `m1`, a `MethodMessage` object `mm` can be generated with the information obtained from `m1` and `m2`. We defined the rule, presented in Figure 6, for transforming `m2` to `mm`. We

decided that the content of mm is equal to the statement (method name) of m2, which is shown in the 5th line in Figure 6. We transform the information about the caller object of m2 to the caller object of mm, which is described from the 8th line to 16th line in Figure 6. The information about the callee object of m2 is also transformed to the callee object of mm, which is shown from the 19th line to the 27th line in Figure 6. The 30th line shows that the return type of method m2 is equal to the attribute `returnType` of mm. From the 33rd line to the 39th line in Figure 6, we describe how the set of formal parameters of m2 are transformed to the set of formal parameters of mm. Lines 42 to 53 in Figure 6 shows that the condition of mm is composed of a set of condition clauses, which are transformed from the set of condition clauses of m2.

The iteration of a single method message mm, which is named `timesOfRepeat`, is determined by the number of executions of m2. Lines 56 to 92 in Figure 6 describe the transformation rule.

The rest of lines in Figure 6 describe the relationship between message mm, which is transformed to present the execution of the method (represented by `MethodCall` m2), and its following messages. The messages, which follow the method message mm, share the same caller object which is actually the callee object of mm. Those following messages are transformed to present executions of methods which have the same `caller` method m2.

A short example is provided to demonstrate the transformation rule for identifying a method message. We have the following traces:

```
.....
"0x0012F700", "A", "Method Entry", "void methodA()"
"0x0012F700", "A", "If", "condition1"
"0x0012F700", "A", "While", "condition2"
"0x0012A800", "B", "Method Entry", "void methodB()"
"0x0012A800", "B", "Method Exit", "void methodB()"
"0x0012F700", "A", "EndControl", "a control flow end"
"0x0012F700", "A", "EndControl", "a control flow end"
.....
```

Figure 7 Traces illustrating Transformation Rule 1

From Figure 7, we are able to obtain the corresponding instance of the TRACE metamodel, which is shown in Figure 8 below.

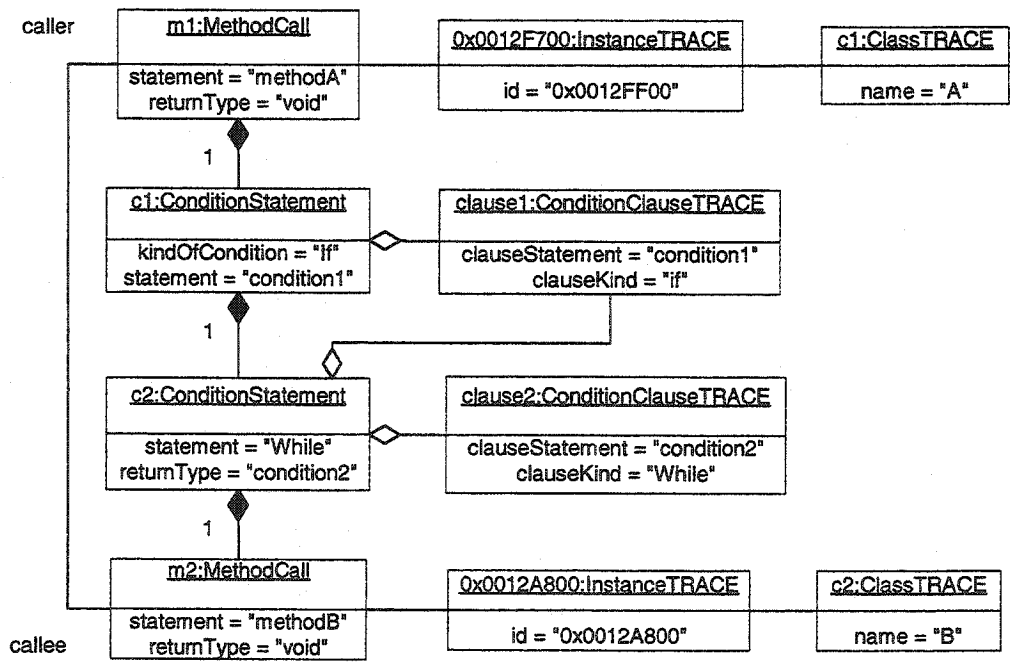


Figure 8 An instance of TRACE Metamodel to Illustrate Transformation Rule 1

According to the method message transformation rule, a method message is acquired. The corresponding instance of the SD metamodel is shown in Figure 9 below.

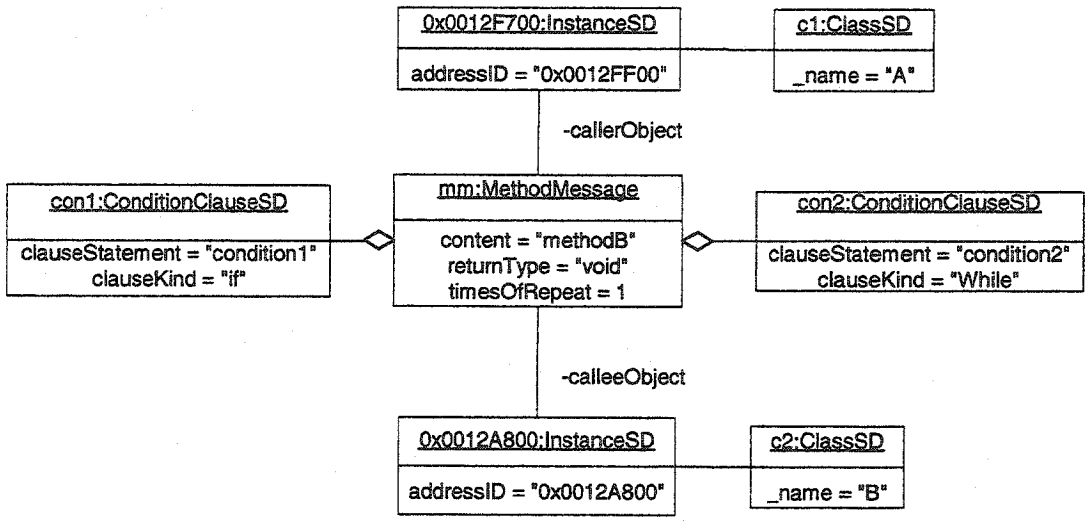


Figure 9 An instance of SD Metamodel to Illustrate Transformation Rule 1

From the relationship between `MethodCall` `m1` and `m2` in Figure 8 and the method message transformation rule, we are able to get a `MethodMessage` object, which is shown as `mm` in Figure 9. The caller object of `mm2` is object `0x0012F700`, and the callee object is `0x0012A800` of class `c2`. A condition, composed of object `con1` and object `con2` of class `ConditionClauseSD`, must be satisfied in order to send message `mm`. Since the condition of `m2` includes a "while" loop condition clause, the iteration of `mm`, which is 1, exists.

3.3.2 Rule 2 - Identifying a return message

To illustrate a `ReturnMessage`, information such as the statement of return message, sender object and receive object is needed. The information is stored in a `Return` object, which is created once the corresponding log lines have been analyzed. In this section, the return message transformation rule is introduced to define the method of transforming the information from a `Return` object in the `TRACE` metamodel to a

ReturnMessage of the SD metamodel. Figure 10 shows the transformation rule for return messages between the TRACE and SD metamodels.

```
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
8           then (
9                rm.callerObject.addressID = r.getContext().id
10               and
11               rm.callerObject.theClass.name = r.getContext().theClass.name
12          )
13          else (
14               rm.callerObject.name = r.getContext().name
15          )
16          and
17          // obtain the callee object of the message
18          if r.inMethod,context.oclType = InstanceTRACE
19          then (
20              rm.calleeObject.addressID = r.inMethod.context.id
21              and
22              rm.calleeObject.theClass.name = r.inMethod.context.theClass.name
23          )
24          else (
25              rm.calleeObject.name = r.inMethod.context.name
26      )
27  )
```

Figure 10 Transformation Rule to Identify Return Messages

In the first two lines of Figure 10, we indicate that if there is a Return object r in the TRACE metamodel, a corresponding ReturnMessage object rm exists in the SD metamodel. Line 4 in Figure 10 assigns the statement of r as equal to the content of rm. Lines 7 to 15 in Figure 10 describe a method for transforming the information of the sender object of r to the sender object of rm. The rest of the transformation rule is designed to transform the information of the receiver object of r to the receiver object of rm. The rules are applied to transform all of the Return objects in the TRACE metamodel into corresponding ReturnMessage objects in the SD metamodel.

A short example is provided below to demonstrate the transformation rule to identify return messages. For example, we have the following traces:

```
.....
"0x0012F700", "A", "Method Entry", "void methodA()"
"0x0012A800", "B", "Method Entry", "void methodB()"
"0x0012A800", "B", "Return", "Hello, world!"
.....
```

Figure 11 Traces illustrating Transformation Rule 2

From Figure 11, we obtain the corresponding instance of the TRACE metamodel, which is shown in Figure 12.

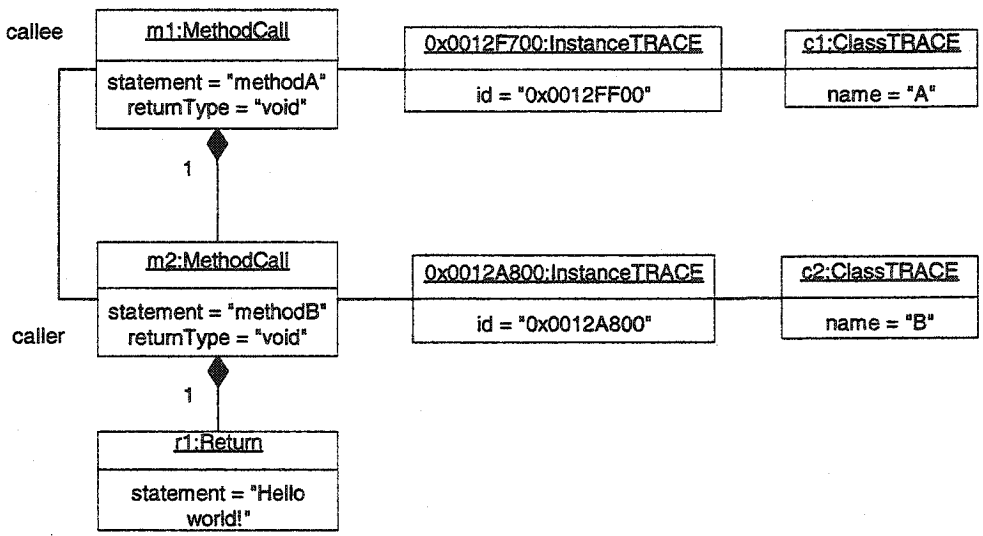


Figure 12 An instance of TRACE Metamodel to Illustrate Transformation Rule 2

According to the return message transformation rule, a return message is obtained. The corresponding instance of the SD metamodel is shown in Figure 13.

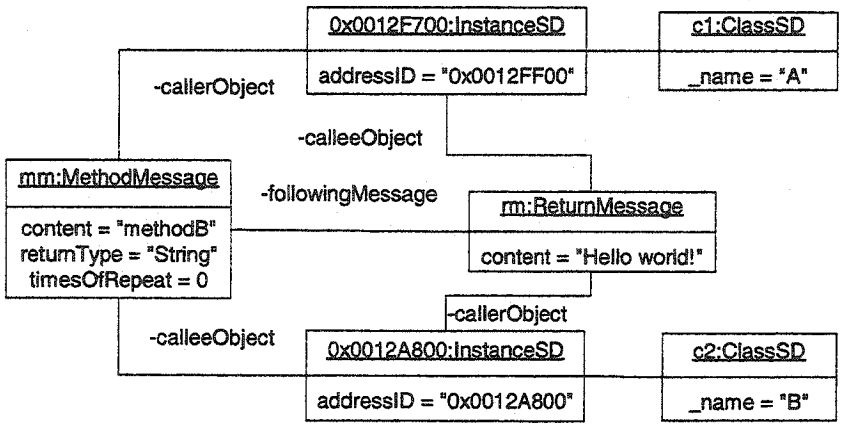


Figure 13 An instance of SD Metamodel to Illustrate Transformation Rule 2

A return statement is represented by `ReutrnMessage rm` with the statement "Hello, world!", which is sent by object `0x0012A800` and received by object `0x0012F700`.

3.3.3 Rule 3 - Identifying iteration of a group of messages

Because the method of presenting the iteration of a single message is different to that of the iteration of a group of messages in UML notation, we introduced the `IterationMessage` class to show that the iteration of a group of messages exists. In this section, we define the transformation rule to show the iteration of a group of messages.

```
1   conditionStatement.allInstances->forAll(c :ConditionStatement |
2      (c.isLoop = true and c.followingStatement->size > 1 )
3      implies
4      iterationMessage.allInstances->exists(im :IterationMessage |
5         // obtain the loop condition of the group of messages
6         im.content = c.statement
7         and
8         // obtain the number of iteration of the group of messages
9         im.iteration =
10           conditionStatement.allInstnces->select(c1:ConditionStatement|
11              c1.clauses->forAll(index:Integer |
12                 c1.clauses->at(index).clauseStatement =
13                       c.clauses->at(index).clauseStatement
14                 and
15                 c1.clauses->at(index).clauseKind =
16                       c.clauses->at(index).clauseKind
17                 )
18              and
19              c1.clauses->size = c.clauses->size
20              and
21              c1.kindOfCondition = c.kindOfCondition
22              and
23              c1.precedentStatement = c.precedentStatement
24           )->size
25        and
26        // obtain the condition clauses of the group of messages
27        im.clauses->forAll(index:Integer |
28           im.clauses->at(index).clauseStatement =
29                    c.clauses->at(index).clauseStatement
30           and
31           im.clauses->at(index).clauseKind = c.clauses->at(index).clauseKind
32        )
33        and
34        im.clauses->size = c.clauses->size
35     )
36  )
```

Figure 14 Transformation Rule to Identify Iteration of a Group of Messages

Figure 14 depicts the transformation rule for showing the iteration of a group messages between the TRACE metamodel and the SD metamodel. A `ConditionStatement` object `c`, which represents a loop condition, is studied. If `c` has more than one following statement, we consider that the iteration of a group of messages exists. A related `IterationMessage` `im` can then be created. The value of content of `im` is equal to the value of statement of `c`. The value of the `iteration` of `rm` is determined by the repetition times of the execution of `c`. Finally, the condition clauses of `rm` are transformed from the condition clauses of `c` in order.

A brief example is provided below to demonstrate the transformation rule to identify iteration in a group of messages. We have the following traces:

```
.....
"0x0012F700", "A", "Method Entry", "void methodA()"
"0x0012F700", "A", "While", "condition1"
"0x0012A800", "B", "Method Entry", "void methodB()"
"0x0012A800", "B", "Method Exit", "void methodB()"
"0x0012A800", "B", "Method Entry", "void methodC()"
"0x0012A800", "B", "Method Exit", "void methodC()"
"0x0012F700", "A", "EndControl", "a control flow end"
.....
```

Figure 15 Traces illustrating Transformation Rule 3

From Figure 15, we are able to obtain the corresponding instance of the TRACE metamodel, which is shown in Figure 16.

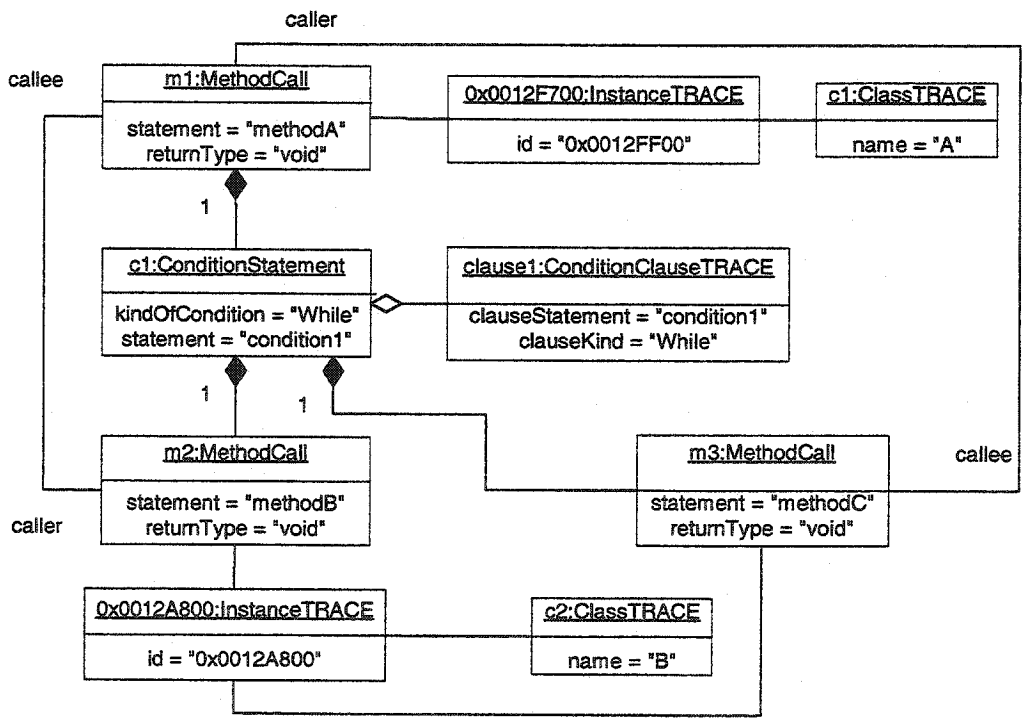


Figure 16 An instance of TRACE Metamodel to Illustrate Transformation Rule 3

According to Transformation Rule 3, the iteration of a group of messages is obtained. The corresponding instance of the SD metamodel is shown in Figure 17.

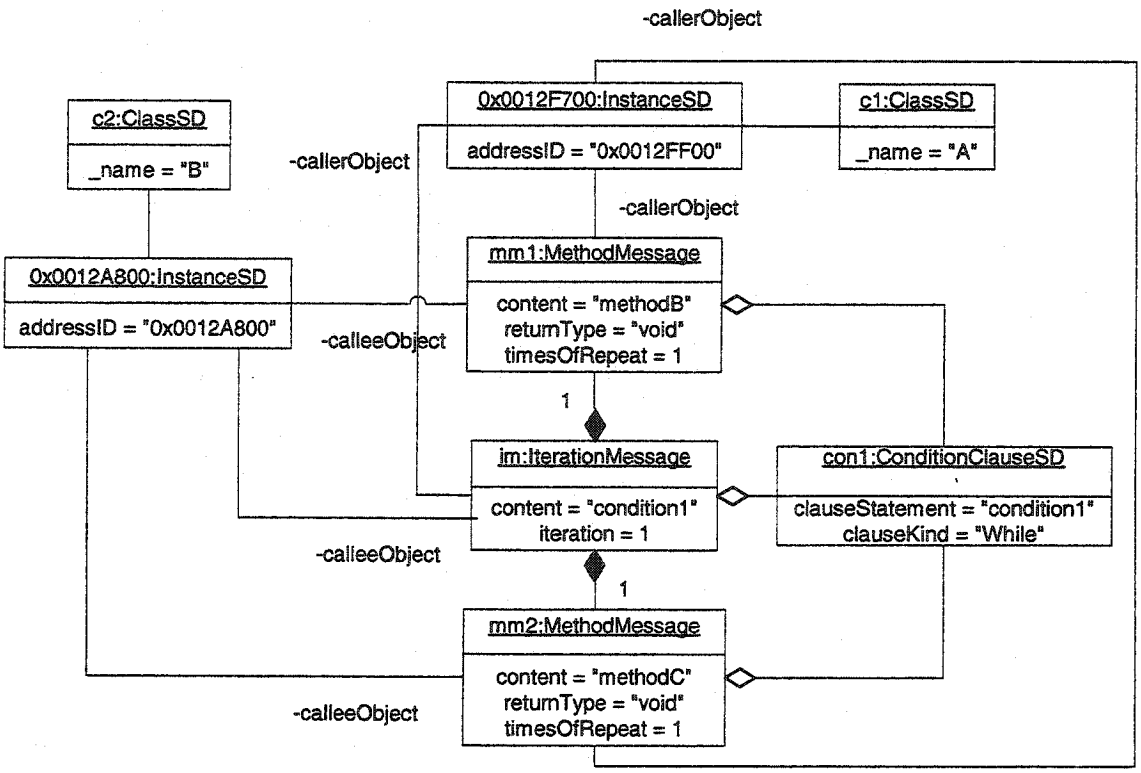


Figure 17 An instance of SD Metamodel to Illustrate Transformation Rule 3

From the relationship between `MethodCall` m1, m2 and m3 in Figure 16, we found that m2 and m3 are executed sequentially, under a "while" loop structure. According to the transformation rule, we can obtain an `IterationMessage` object. It indicates that method messages mm1 and mm2 show a group of messages with iteration. The iteration of the group messages is 1.

Chapter 4 Implementation of an Automation Tool

An automation tool was developed in order to recover scenario diagrams. The system behaviors produced by our automation tool depict a scenario diagram in a comprehensible text-base format, rather than graphic visualization.

According to our design, which is introduced in Chapter 3, our automation tool is composed of two parts: a code instrumentation tool and a model recovery tool.

4.1 Code Instrumentation Tool

Applying the methodology described in section 3.2, an instrumentation tool InstrTC++ (Instrumentation Tool for C++) was developed. InstrTC++ instruments source code for recording dynamic information about objects' interactions, which will be classified and analyzed for scenario diagram recovery. InstrTC++ is designed to instrument any C++ program. It augments source code with trace-generating statements for every non-static method, return statement and control flow structure statement.

4.1.1 Requirement

An instrumentation tool InstrTC++ was developed to instrument source files. InstrTC++ should meet the following functional requirements:

- Load the source file.
- Define the location of the trace file.
- Parse the source file.
- Identify each non-static method, return statement and control flow structure.

- Add trace-generating statements to detect the dynamic information of program executions.

4.1.2 Design of Code Instrumentation

The trace-generating statements are added at the beginning and end of the definition bodies of methods (illustrated in Appendix A) and control flow structures respectively. The return statements and keywords for the end of the control flow (e.g., break, continue) are detected and augmented by trace-generating statements as well.

The definition of method, return statements and control flow structures can be detected in the source code, based on the semantics of the C++ programming language [Deitel+ 98]. As discussed in section 3.2.1, the execution of the static method is not investigated.

The procedure of code instrumentation can be demonstrated by the following flow charts.

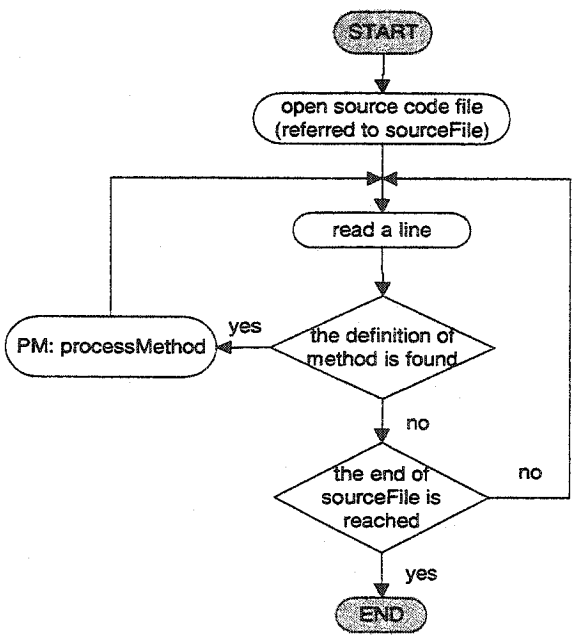


Figure 18 Flow Chart of Code Instrumentation Procedure (1)

Figure 18 describes the procedure for instrumenting code for a source code file of a program. PM represents the procedure for detecting and analyzing a method. The sub steps of the PM procedure are illustrated in Figure 19 below.

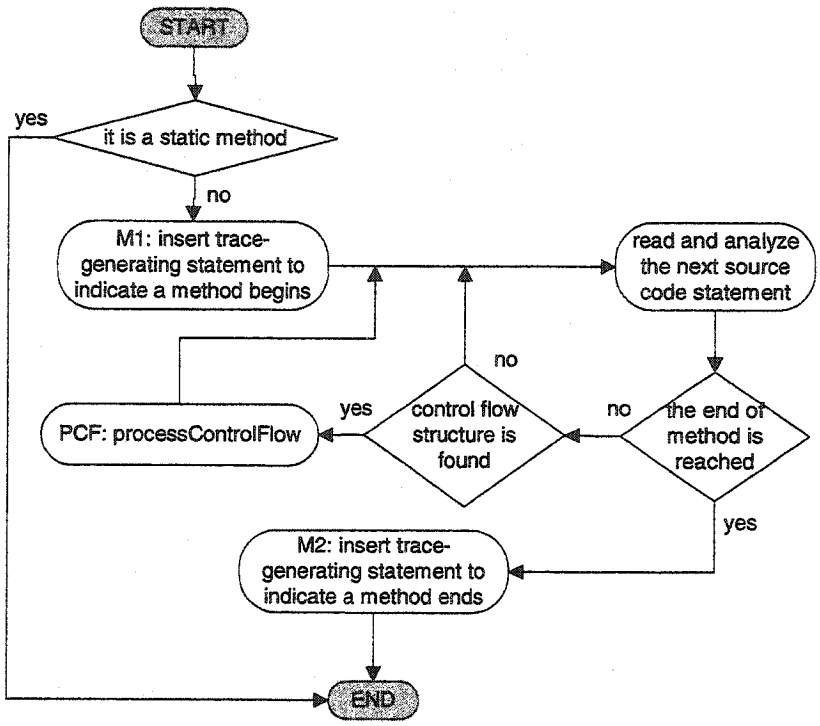


Figure 19 Flow Chart of Code Instrumentation Procedure (2)

M1 Add a trace-generating statement after the method definition to declare that a method is called;

M2 Add a trace-generating statement to declare that the execution of a method is about to finish;

PCF Procedure for detecting and analyzing control flow structures (see below).

The PCF procedure is designed to instrument control flow structures. The sub-steps of the PCF procedure are illustrated in Figure 20.

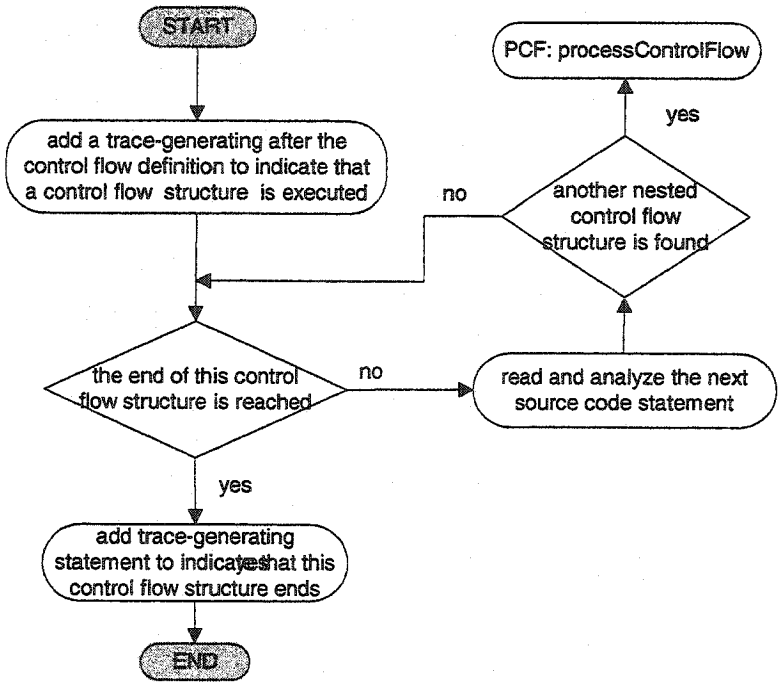


Figure 20 Flow Chart of Code Instrumentation Procedure (3)

A control flow structure can be defined in another control flow structure. During the analysis and instrumentation procedure of control flow structures, a recursion design structure is applied, as described in Figure 20. The detailed algorithms can be found in Appendix B.

4.1.3 Implementation

Perl, a powerful and efficient script language, was chosen to implement InstrTC++. Perl's pattern matching and textual manipulation capabilities outperform C++ or Java. Perl is also a convenient file manipulation language. It can help deal with the files themselves apart from their contents, moving them, renaming them, changing their permissions, and so on. With Perl, users don't have to worry about arbitrary restrictions on string length, input line length, or the number of elements in an array. Perl's regular expression handling is very powerful. The language has a proper set of regular expression beyond that of others. Perl's data types and operators are richer, such as scalars,

numerically indexed arrays (lists), and string-indexed (hashed) array. Each of these holds arbitrary data values.

There is an issue we'd like to discuss here. When capturing dynamic information, the file input and output stream is used. The trace file needs to be opened and closed every time a log is recorded. This may slow down the execution of the program. Traces can be collected in other ways which are not applied in our approaches. For example, an attribute can be added to the constructors of classes in the source code to take control of the writing of the trace file, so that the trace file doesn't have to be opened and closed frequently. Furthermore, a temporary buffer can be defined to collect dynamic information. Once the buffer is full, it can be emptied by writing the logs into the trace file. Although it is more efficient, more modifications to the source code must be made in order to carry out this methodology.

4.1.4 Usage

The Instrumentation tool is executed under an MS-DOS environment. Our instrumentation tool (InstrTC++) instruments one source file of the target system at a time. Source code files are instrumented one by one in order to obtain and analyze the behavior between objects. However, the user can only instrument the source files, which include the class that he/she is interested in. In this case, interactions involving the objects of classes not of interest will not be captured.

To execute the instrumentation tool correctly, the user needs to know: (1) the location of the instrumentation tool; (2) the location of the source code file which is to be instrumented; (3) the location of the trace file from which dynamic information can be collected; (4) the location of a file which stores the signatures of static methods during

the instrumentation. As discussed in section 3.2.1, a static method will not be instrumented, since it is independent of any objects. A static method can be declared in the declaration file (header file) but implemented in the definition file (.cpp file). In the definition file, it is not necessary to re-notify that a method (which has already been declared in the header file) is a static method. Since InstrTC++ instruments one source file at a time, we need to store the information about a static method (obtained from a header file). Therefore, we can ignore the definition body of the static method when analyzing and instrumenting the corresponding definition file.

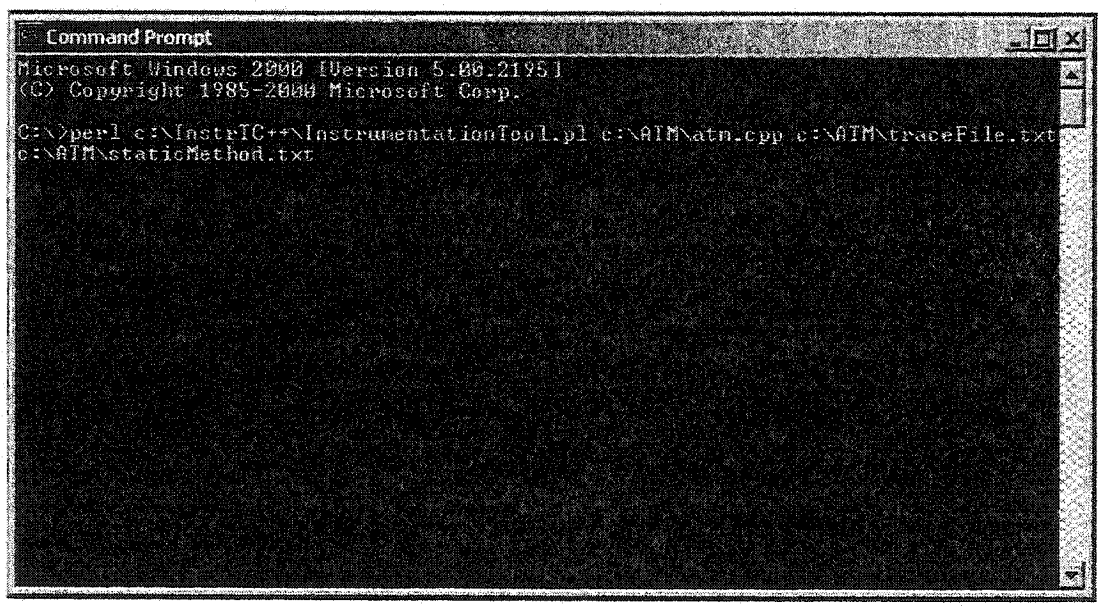


Figure 21 Execution of Instrumentation Tool

Figure 21 shows an example of the execution of the instrumentation tool. The instrumentation tool instruments "atm.cpp" of the ATM system. All of the dynamic information will be recorded in a trace file called "traceFile.txt", located at c:\ATM. The "staticMethod.txt" file collects the signatures of the static method during instrumentation. Since all dynamic information is stored in a trace file, the argument about the trace file

should not be changed when the instrumentations of the source code files are executed. So is the static method file.

An example of instrumented source code can be found in Appendix H.

4.2 Scenario Diagram Recovery Tool

A trace file containing the run time information is generated when the instrumented source code is executed. By applying the transformation rules described in section 3.3, a scenario diagram recovery tool (RESDTool) was designed to recover the scenario diagram of a target system from its dynamic information collected at run time.

4.2.1 Requirement

A trace file is loaded and analyzed by RESDTool. According to a set of transformation rules, RESDTool transforms the dynamic information in the trace file to the corresponding scenario diagram. In this section, we illustrate the use case diagram and activity diagram to describe the requirement of RESDTool.

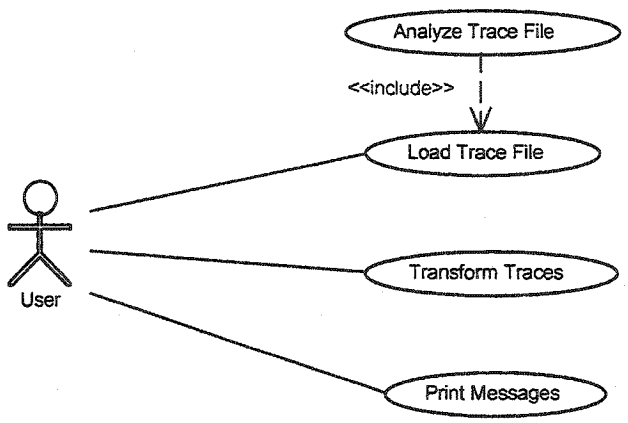


Figure 22 Use Case Diagram of RESDTool

Figure 22 illustrates the use case diagram of the RESDTool. It consists of four use cases: load a trace file, analyze the trace file, transform traces and print messages. The corresponding scenario diagrams, which describe the use cases in detail, are presented in Appendix C. The detailed use cases are explained as follows:

Use case name	**Analyze Trace File**
Participating actor	User
Entry condition	1. The original trace file has been loaded into the system.
	2. User selects Analyze Trace File use case.
Flow of events	3. Reads each log in the trace file.
	4. Looks into each log line; according to different type of dynamic information, each log line can be analyzed and translated into a `MethodCall`, `Return` and `ConditionStatement` object.
Exit condition	5. A set of objects, which represents the information of log lines in trace file, has been created.
	6. The analysis of the trace file is completed.

Use case name	**Load Log File**
Participating actor	User
Entry condition	1. The trace file containing the dynamic information has been generated after the target system has been run.
	2. User selects Load Trace File use case.
Flow of events	3. The system loads the trace file.
Exit condition	4. The dynamic information of the target system has been obtained.

Use case name	**Print Messages**
Participating actor	User
Entry condition	1. All the operations have been transformed.
	2. User selects Print Messages.
Flow of events	3. Presents message objects one by one.
Exit condition	4. Messages have been presented.

Use case name	**Transform Traces**
Participating actor	User
Entry condition	1. The original trace file has been analyzed, and all the dynamic information obtained from the log lines has been translated into a set of `ExecutionStatement` objects.
	2. User selects Transform Traces use case.
Flow of events	3. The system transforms the `MethodCall` and `Return` objects into `Message` objects, according to the defined transformation rules.
Exit condition	4. The `Message` objects, which compose the corresponding scenario diagram, have been created.

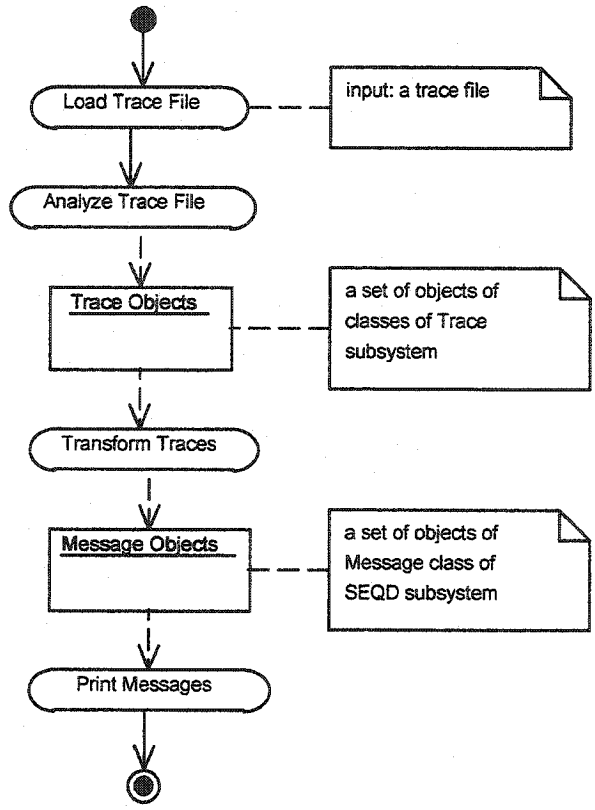


Figure 23 RESDTool Activity Diagram

The activity diagram can be used to describe the sequential dependencies among individual use cases. Figure 23 shows the activity diagram of the RESDTool. The user loads the trace file, which contains the dynamic information obtained from the target system at run time. Each log line in the trace file is analyzed to translate the useful information into the corresponding type of execution statement object. From the results of the analysis, the `MethodCall` and `Return` objects are transformed into messages according to the algorithms. A set of `Message` objects are created after processing the "Transform Traces" use case. The "Print Messages" use case presents all message objects to show the interactions.

4.2.2 Packages of RESDTool

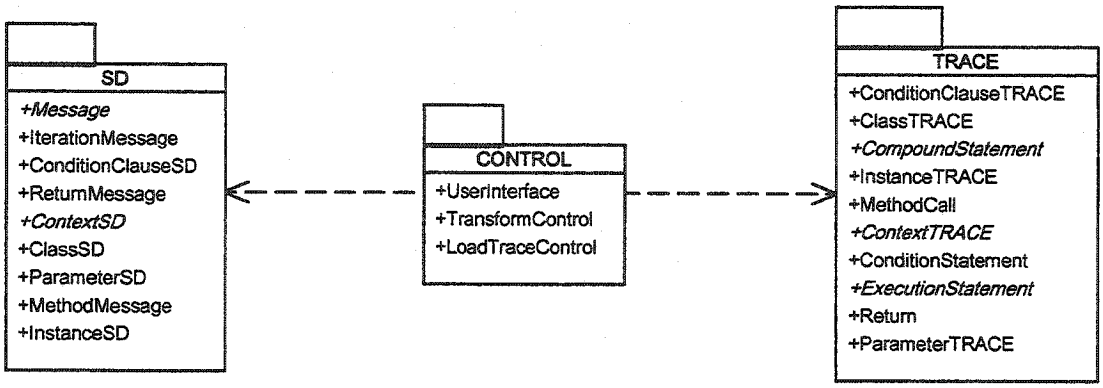


Figure 24 RESDTool Package Diagram

The RESDTool consists of three subsystems:

- The TRACE subsystem;
- The Scenario diagram (SD) subsystem;
- The Control subsystem

The TRACE subsystem, which is developed based on the TRACE metamodel described in section 3.2.3, is responsible for storing the information about the log lines in the trace file. The SD subsystem, developed based on the SD metamodel described in section 3.1.2, is responsible for storing interaction messages which build the corresponding scenario diagram. The transformation between the TRACE metamodel and the SD metamodel is accomplished by a set of transformation rules, which are defined in the Control subsystem. The Control subsystem is responsible for loading a trace file, translating the information from the log lines in the trace file into a set of ExecutionStatement objects, and transforming them into a set of Message objects, which represent the interaction information recorded in the trace file, by applying defined mapping rules. Figure 24 illustrates the package diagram of the RESDTool.

The RESDTool class diagram consists of the class diagram of the `Control` subsystem, the `TRACE` subsystem (designed according to the TRACE metamodel) and the `SD` subsystem (designed according to the SD metamodel). The relationships between those subsystems can be found in the RESDTool package diagram. The transformation algorithms between the `TRACE` subsystem and the `SD` subsystem are described in section 3.3. All the classes in the subsystems will be explained in the Data Dictionary in Appendix D.

Since the class diagrams of the `TRACE` subsystem and the `SD` subsystem are discussed in Chapter 3, here we will discuss only the `Control` subsystem.

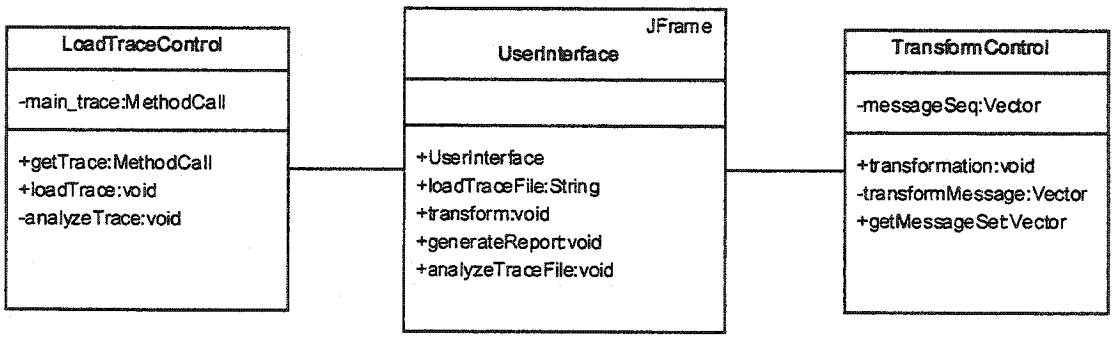


Figure 25 Control Subsystem Class Diagram

Figure 25 shows the class diagram of the `Control` subsystem, which is composed of three classes: the `UserInterface` class, the `TransformControl` class and the `LoadTraceControl` class.

The `LoadTraceControl` class is responsible for loading the trace file, processing and translating each log line in the trace file into an `ExecutionStatement`. The attribute `main_trace` is a `MethodCall` object. It represents a "main" method of the program as an invoker of followed interactions. The operation `loadTrace` obtains the name and location of a trace file to be analyzed. The

operation `analyzeTrace` is designed to generate objects of the `TRACE` subsystem to present the information in the trace file.

The `TransformControl` class is designed to transform the interaction information from the objects defined in the `TRACE` subsystem to objects defined in the `SD` subsystem. The attribute `messageSeq` presents a set of messages. The operation transform and `transformMessage` are implemented to transform the objects of the `TRACE` subsystem to the objects of the `SD` subsystem.

The `UserInterface` class is designed as a boundary class of the RESDTool.

4.2.3 Implementation

Java™ 2 Platform, Standard Edition, v1.4 [J2SE™ 1.4], a platform-independent language, is used to implement RESDTool. RESDTool's GUI was built using Java Swing components. The new regular expression feature in Java™ 2 Platform, Standard Edition, v1.4 was applied to find pattern matching in implementation. Detailed algorithms are provided in Appendix E.

4.2.4 Usage

Figure 26 shows the GUI of the RESDTool. The user needs to provide: (1) **Trace File**, the location and the file name of the trace file, which contains the run time information to be analyzed. It can be input either manually, from the keyboard, or by browsing the hard disk directory using the corresponding **Browse** button. (2) A list of derived classes of the target system. This should be input manually into the **Derived Classes** field. Class names are separated by semicolons.

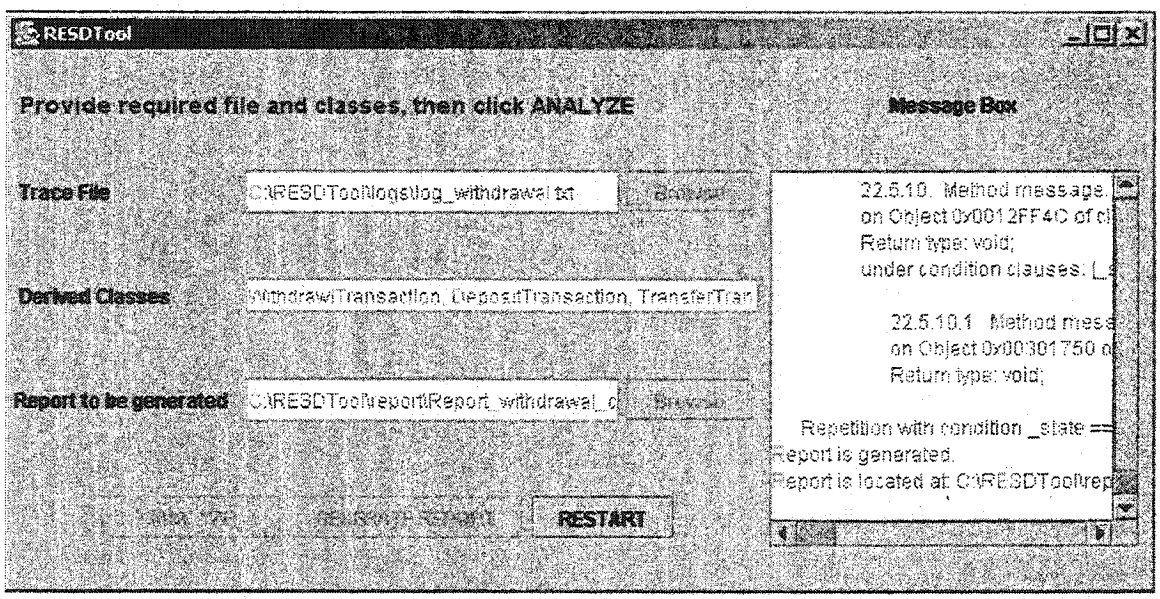


Figure 26 GUI of RESDTool

Once the two parameters have been provided, the user can press the **ANALYZE** button to analyze the trace file and translate the information into the corresponding messages.

A report containing all the messages can be created by pressing the **GENERATE REPORT** button. The location and the name of the report file can be offered by the user in the **Report to be generated** field. Otherwise, the default path and name of a report file is given, i.e., c:\RESDToolReport.txt.

The **RESTART** button can be used to start a new analysis. The user can monitor the process and result in the **Message Box** text area.

Chapter 5 Case Study

In this section, an ATM (Automated Teller Machine) banking system is used as a case study. This system was selected because it was not developed by the author, and both the C++ source code and the UML design documents were available. In particular, use cases are described by sequence diagrams, thus allowing us to validate our approach: i.e., how close from the expected sequence diagrams are the scenario diagrams generated by the approach? Withdrawal (section 5.2) and InvalidPIN (section 5.3) use cases are selected as examples to prove the feasibility of our methodology, the reverse engineering of the former being detailed. Other retrieved scenario diagrams of use cases can be found in Appendix L. Lessons which are learned from the case study are discussed in section 5.4.

5.1 ATM Banking System

The ATM system is a banking system which provides customers with money withdrawal, money deposit, money transfer and balance query services. The ATM services one customer at a time. A customer is required to insert an ATM card and enter a personal identification number (PIN) – both of which will be sent to the bank for validation as part of each transaction. The customer is then able to perform one or more transactions. The card is retained in the machine until the customer indicates that he/she desires no further transactions, at which point it is returned. If the bank determines that the customer's PIN is invalid, the customer is required to re-enter the PIN before a transaction can proceed. If the customer is unable to successfully enter the PIN after two

re-tries, the card is permanently retained by the machine, and customers have to contact the bank to get it back. If a transaction fails for any reason other than an invalid PIN, the ATM displays an explanation of the problem, and then asks the customer whether he/she wants to perform another transaction. The ATM provides the customer with a printed receipt for each successful transaction, showing the date, time, machine location, type of transaction, account(s), amount, and an available balance(s) of the affected account. The ATM allows an operator to remove deposit envelopes and reload the machine with cash, blank receipts, and so on.

The ATM application is implemented using C++ language. The main classes of the ATM system are (The class diagram of the ATM system can be found in Appendix F.):

1. Class ATM, which manages the ATM and component parts. It has the following component classes: CardReader, Display, Keyboard, CashDispenser, ReceiptPrinter, EnvelopeAcceptor and OperatorPanel.
2. Class Session, which performs a session use case. The ATM card number and customer pin number are obtained in the session use case. An appropriate transaction is initiated by the session.
3. Class Transaction, which abstracts common features of the various types of transactions, presented by WithdrawlTransaction, DepositTransaction, TransferTransaction and InquiryTransaction subclasses.
4. Class Bank, which manages communications with the bank.

The use cases of the ATM system include: System startup, System shutdown, Session, Transaction, Withdrawal, Deposit, Transfer, Inquiry and Invalid PIN. The use

case diagram of the ATM system is presented in Appendix G. The corresponding UML sequence diagrams are shown in section 5.2, section 5.3 and Appendix K respectively.

In order to obtain dynamic information at run time, we need to instrument all the source files separately, using InstrTC++. We use *session.cpp* as an example to present the usage of InstrTC++. The complete instrumented source code can be found in Appendix H.

5.2 Withdrawal Use Case

A withdrawal transaction asks the customer to choose a type of account (e.g. checking) from a menu of possible accounts, and to choose a dollar amount from a menu of possible amounts. The system verifies that it has sufficient money on hand to satisfy the request before sending the transaction to the bank. (If not, the customer is informed.) If the transaction is approved by the bank, the appropriate amount of cash is dispensed by the machine before it issues a receipt. The withdrawal use case consists of several scenarios such as withdrawal with success and withdrawal with failure due to insufficient money.

In the following section, we detail all the steps of the reverse engineering of the withdrawal use case. Only one possible scenario, i.e., a successful withdrawal, is used.

5.2.1 Trace File of Withdrawal Use Case

Once the instrumented source code has been executed, a trace file is generated. All of the dynamic information is recorded in the trace file. Figure 27 shows a fragment of the trace file of the withdrawal use case, which was obtained from the ATM system at run time. The complete trace file can be found in Appendix I.

```
1       ..............................
2.      "0x0012FF4C","ATM","Method Entry","ATM(int number,const char* location, Bank &bank)"
3.      "0x0012FF4C","ATM","Method Exit"," ATM(int number,const char* location, Bank &bank)"
4.      "0x0012FF4C","ATM","Method Entry","void serviceCustomers()"
5.      "0x0012FF4C","ATM","While","_state == RUNNING"
6.      "0x00301710","Display","Method Entry","void requestCard()"
7       "0x00301710","Display","Method Exit","void requestCard()"
8       "0x0012FF4C","ATM","Do","_state == RUNNING && readerStatus == CardReader::NO_CARD"
9       "0x00301740","CardReader","Method Entry","CardReader::ReaderStatus checkForCardInserted()"
10      "0x00301740","CardReader","Return"," _status"
11      "0x0012FF4C","ATM","EndControl","a control flow end"
12      "0x0012FF4C","ATM","If","_state == RUNNING && readerStatus ==
                    CardReader::CARD_HAS_BEEN_READ"
13      "0x00301740","CardReader","Method Entry","int cardNumber() const"
14      "0x00301740","CardReader","Return"," _cardNumberRead"
15      "0x00302710","Session","Method Entry","Session(int cardNumber, ATM &atm, Bank &bank)"
16.     "0x00302710","Session","Method Exit"," Session(int cardNumber, ATM &atm, Bank &bank)"
17.     "0x00302710","Session","Method Entry","void doSessionUseCase()"
18.     ..............................
19      "0x00301070","Transaction","Method Entry","Transaction(Session& session,ATM& atm,Bank& bank)"
20      "0x00301070","Transaction","Method Exit","Transaction(Session& session,ATM& atm,Bank& bank)"
21      "0x00301070","WithdrawlTransaction","Method Entry","WithdrawlTransaction(Session& session,ATM&
                    atm,Bank& bank)"
22      "0x00301070","WithdrawlTransaction","Method Exit","WithdrawlTransaction(Session& session,ATM&
                    atm,Bank& bank)"
23      "0x00301070","Transaction","Method Entry","Status::Code doTransactionUseCase()"
24      ..............
```

Figure 27 Fragment of Trace File of Withdrawal Use Case

Each log line has four fields (as described in Section 0): an *object ID* field, *class name* field, *kind of statement* field and *execution statement* field. The *Object ID* field indicates the unique ID of the callee object, which is actually the memory address of this object; the *class name* field indicates the class name in which the execution statement is defined; the *kind of statement* field indicates the type of statement execution, such as "Method Entry", "Method Exit", "While", "Return" and "EndControl"; the *execution statement* field records the detailed executed statement or a statement to indicate the end of the control flow.

As an example (Figure 27), lines 2 and 3 correspond to the creation of an instance of class ATM (its memory address is 0x0012FF4C): line 2 is the constructor's entry, and line 3 the constructor's exit. Similarly, lines 21 and 22 correspond to the creation of an instance of class WithdrawalTransaction (its memory address is 0x00301070). Note that, in line 23, method doTransactionUseCase, though defined in class Transaction (the second field in the log line is "Transaction"), is called on the WthdrawalTransaction instance created at lines 21 and 22: the address of the object in lines 21, 22 and 23 is the same.

5.2.2 An Instance of TRACE Metamodel for Withdrawal Use Case

Figure 28 shows a part of the instance of the TRACE metamodel, corresponding to the trace file in Appendix I. This part of the instance of TRACE metamodel corresponds to lines 4 to 17 in Figure 27. It shows a set of objects of classes MethodCall, ConditionStatement, ConditionClauseTRACE, ClassTRACE, InstanceTRACE and ParameterTRACE, and their relationships.

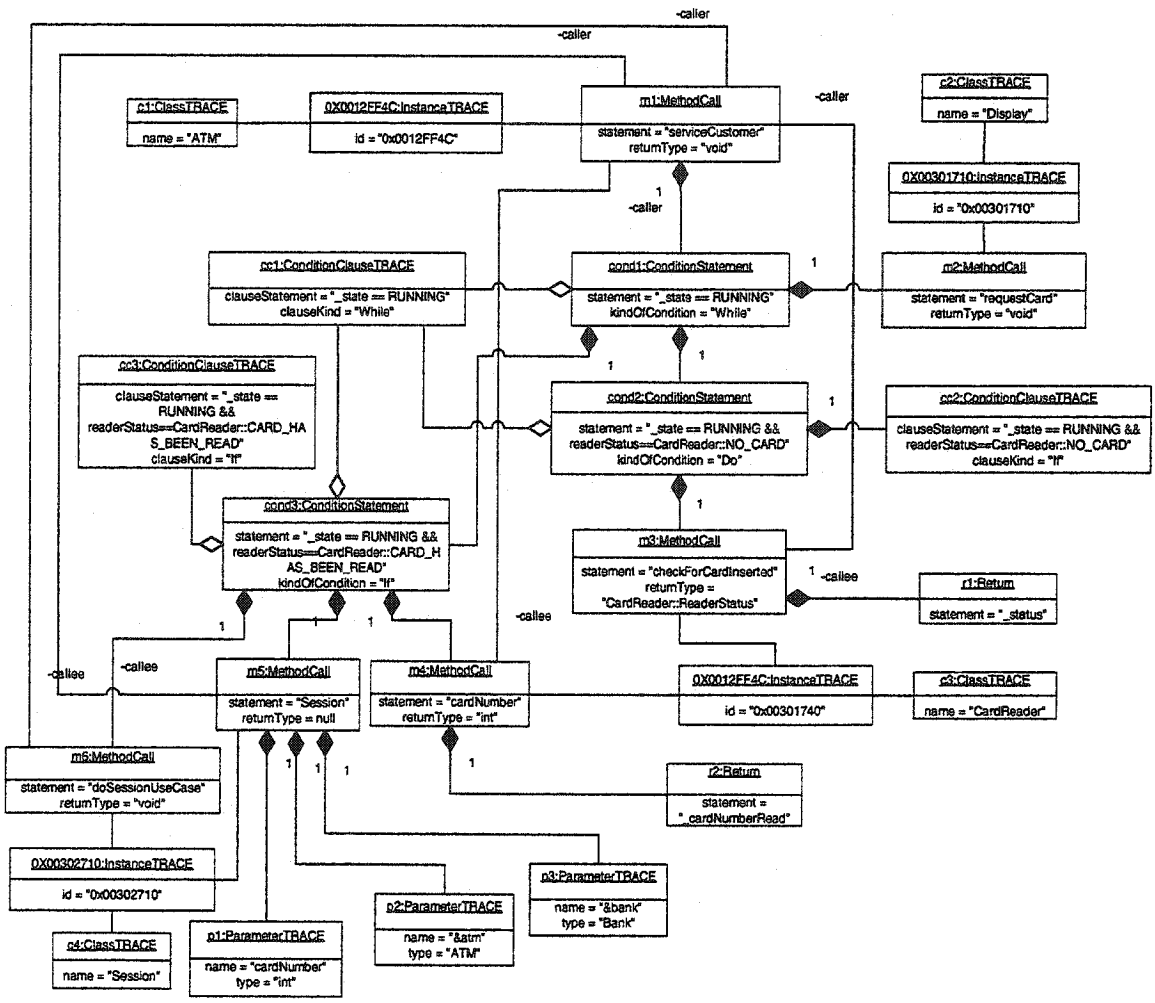


Figure 28 Fragment of an Instance of TRACE Metamodel for Withdrawal Use Case

We obtain the information about an operation, which is represented by `MethodCall` object `m1`, from the 4th log line. The attribute `statement` of `m1` shows that the name of the operation is "serviceCustomers". The attribute `returnType` is "void", which shows no particular value has been returned. The method has no parameter. We also know that the callee object of `m1` is the `InstanceTRACE` object `0x0012FF4C`, whose `id` is "0x0012FF4C", of class `c1` whose name is "ATM".

The 5th log line shows a "while" control flow structure, which is represented by a `ConditionStatement` object `cond1`. The object `cond1` is contained in `m1` object.

It has one condition clause, which is represented by object `cc1` of class `ConditionClauseTRACE`.

The 6th and 7th long lines record the execution of an operation "requestCard", which is represented by `MethodCall` object `m2`. The callee object of `m2` is `InstanceTRACE` object `0x00301710` which is an instance of the class Display. The caller method of `m2` is `m1`.

The 8th line describes the execution of a "do/while" control flow, which is represented by a `ConditionStatement` object `cond2`.

The 9th log line records the execution of an operation "checkForCardInserted", which is represented by `MethodCall` object `m3`. The caller method of `m3` is `m1`. The return message of the operation is described in the 10th log line, which is presented by a `Return` object `r1`.

The information in the 12th log line shows that an "if" control flow structure is executed. A `ConditionStatement` object `cond3` is instantiated.

The 13th log line records the execution of an operation "cardNumber", which is represented by `MethodCall` object `m4`. The callee object of `m4` is an `InstranceTRACE` object `0x00301740`. The caller method of `m4` is `m1`. The corresponding return message is recorded in line 14, which is presented by the `Return` object `r2`.

The 15th and 16th log lines show the execution information of an operation that is instantiated as `MethodCall` object `m5`, which has 3 parameters, represented by `p1`, `p2` and `p3`. It is a constructor of the Session class, whose return type is null. The callee object of `m2` is represented by `0x00302710` object whose `id` attribute is "0x302710" as

its physical address. `0x00302710` is an object of class `c4` whose `name` is "Session". The caller method of `m5` is `m1`.

The 17th log line shows that an operation named "doSessionUseCase" is executed with no parameter and "void" return type. It is instantiated as a `MethodCall` object `m3`. The caller object of `m3` is represented by `0x00302710`. It is invoked by `m1`.

5.2.3 An Instance of SD Metamodel for Withdrawal Use Case

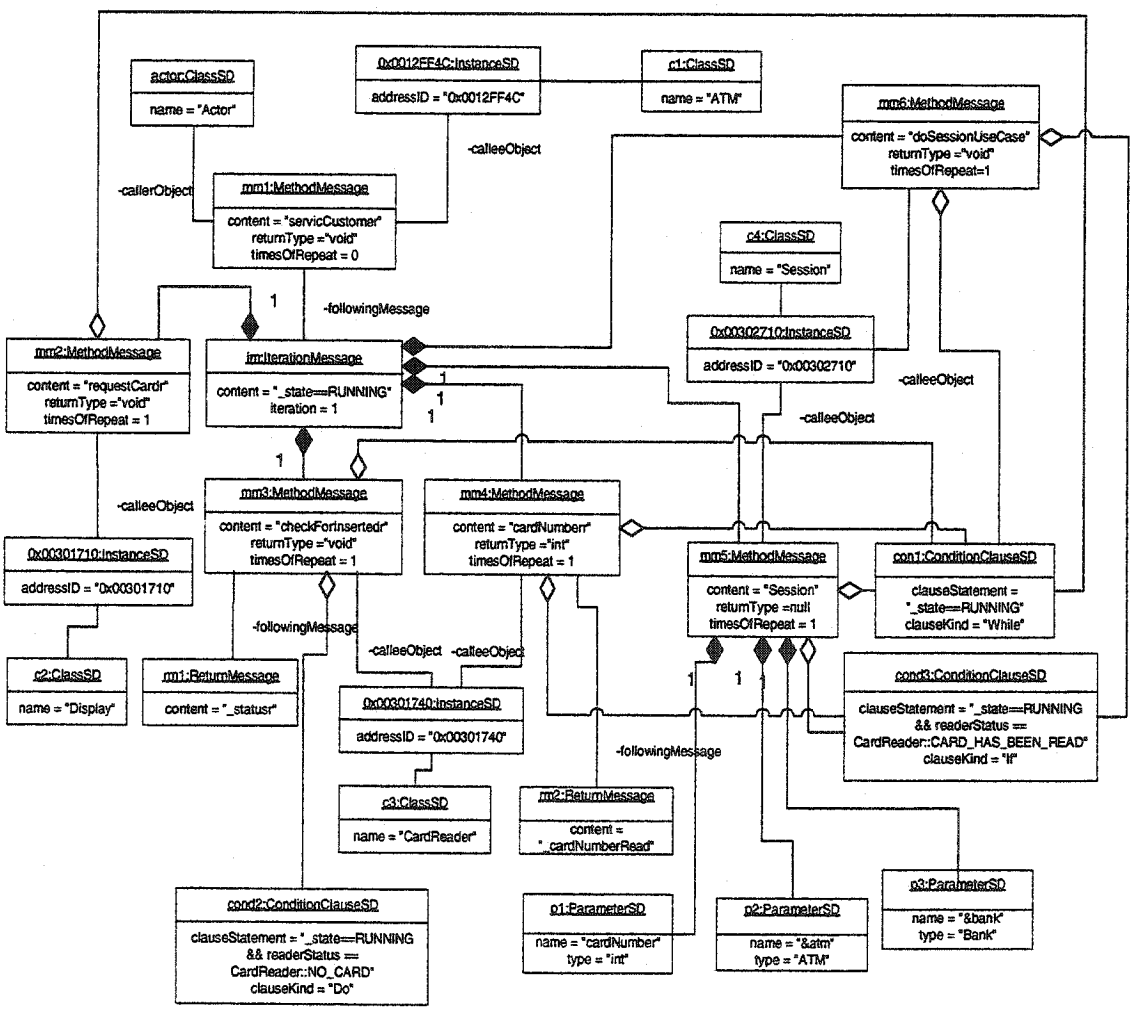


Figure 29 Fragment of an Instance of SD Metamodel for Withdrawal Use Case

Figure 29 shows part of the instance of SD metamodel, corresponding to the instance of TRACE metamodel shown in Figure 28. This instance of SD metamodel shows a set of objects of ClassSD, InstanceSD, MethodMessage, IterationMessage, ReturnMessage, ParameterSD and ConditionClauseSD classes and their relationships.

`MethodMessage mm1` represents an operation (named "serviceCustomers") which is transformed from the `MethodCall` object `m1` in Figure 28. The caller object of the operation, which could be an external actor, is not found in the trace file. We consider the caller to be an object of the Actor class. The callee object of the operation is object `0x0012FF4C` of ATM class, whose `addressID` is "0x0012FF4C". The method doesn't have any parameters. The object `0x0012FF4C` is the caller object of all other method messages. The callee objects of messages are presented as well.

`IterationMessage im` presents a group of messages sent under a "while" loop condition. The group of messages is composed of `MethodMessage mm2, mm3, mm4, mm5` and `mm6`.

Aside from the condition clause which is presented in `cond1`, the condition clause presented by `cond3` needs to be fulfilled to get message `mm3`. In addition, the message `mm4`, `mm5` and `mm6` are obtained only when condition clauses presented in `cond1` and `cond3` are fulfilled.

The return message `rm1` and `rm2` are created to present the corresponding return statement of method message `mm3` and `mm4` respectively.

5.2.4 Scenario Diagram of Withdrawal Use Case

After analyzing the trace file of the Withdrawal use case (i.e., transforming the information read into an object diagram), and producing an instance of a scenario

diagram, a result file, which records messages sequentially, is generated (Figure 30). Note that all theses steps are automated using the RESDTool.

```
9.  Method message: serviceCustomers
on Object 0x0012FF4C of class ATM from Class Actor;
Return type: void;

      Repetition with condition _state == RUNNING begins

            9.1.  Method message: requestCard
            on Object 0x00301710 of class Display from Object 0x0012FF4C of
            class ATM;
            Return type: void;
            under condition clauses: [_state == RUNNING(While)]
            repeated 1 times

            9.2.  Method message: checkForCardInserted
            on Object 0x00301740 of class CardReader from Object 0x0012FF4C
            of class ATM;
            Return type: CardReader::ReaderStatus;
            under condition clauses: [_state == RUNNING(While), _state ==
            RUNNING && readerStatus == CardReader::NO_CARD(Do)]
            repeated 1 times

            9.2.1. Return message: _status
            from Object 0x00301740 of class CardReader to Object 0x0012FF4C
            of class ATM

            9.3.  Method message: cardNumber
            on Object 0x00301740 of class CardReader from Object 0x0012FF4C
            of class ATM;
            Return type: int;
            under condition clauses: [_state == RUNNING(While), _state ==
            RUNNING && readerStatus == CardReader::CARD_HAS_BEEN_READ(If)]
            repeated 1 times

            9.3.1. Return message: _cardNumberRead
            from Object 0x00301740 of class CardReader to Object 0x0012FF4C
            of class ATM

            9.4.  Method message: Session
            on Object 0x00301130 of class Session from Object 0x0012FF4C of
            class ATM;
            Parameter Set: ["cardNumber" of int type, "atm" of ATM& type,
            "bank" of Bank& type]
            under condition clauses: [_state == RUNNING(While), _state ==
            RUNNING && readerStatus == CardReader::CARD_HAS_BEEN_READ(If)]
            repeated 1 times

            9.5.  Method message: doSessionUseCase
            on Object 0x00301130 of class Session from Object 0x0012FF4C of
            class ATM;
            Return type: void;
            under condition clauses: [_state == RUNNING(While), _state ==
            RUNNING && readerStatus == CardReader::CARD_HAS_BEEN_READ(If)]
            repeated 1 times
```

Figure 30 A Fragment of the Scenario Diagram Result File

The messages marked by sequence number 9, 9.1, 9.2, 9.3, 9.4, 9.5, 9.2.1 and 9.3.1 are depicted for `MethodMessage` `mm1`, `mm2`, `mm3`, `mm4`, `mm5`, `mm6`, `rm1` and `rm2` respectively. From Figure 30, we can see that mm2, mm3, mm4, mm5 and mm6 are a group of messages which are performed under a loop condition "_state==RUNNING". Their caller and callee object are shown in Figure 30 as well. The complete scenario diagram result file can be found in Appendix J.

5.2.5 Validity

Figure 31 shows the sequence diagram of Withdrawal use case as provided in the UML design documents. CustomerConsole represents Keyboard class and Display class.

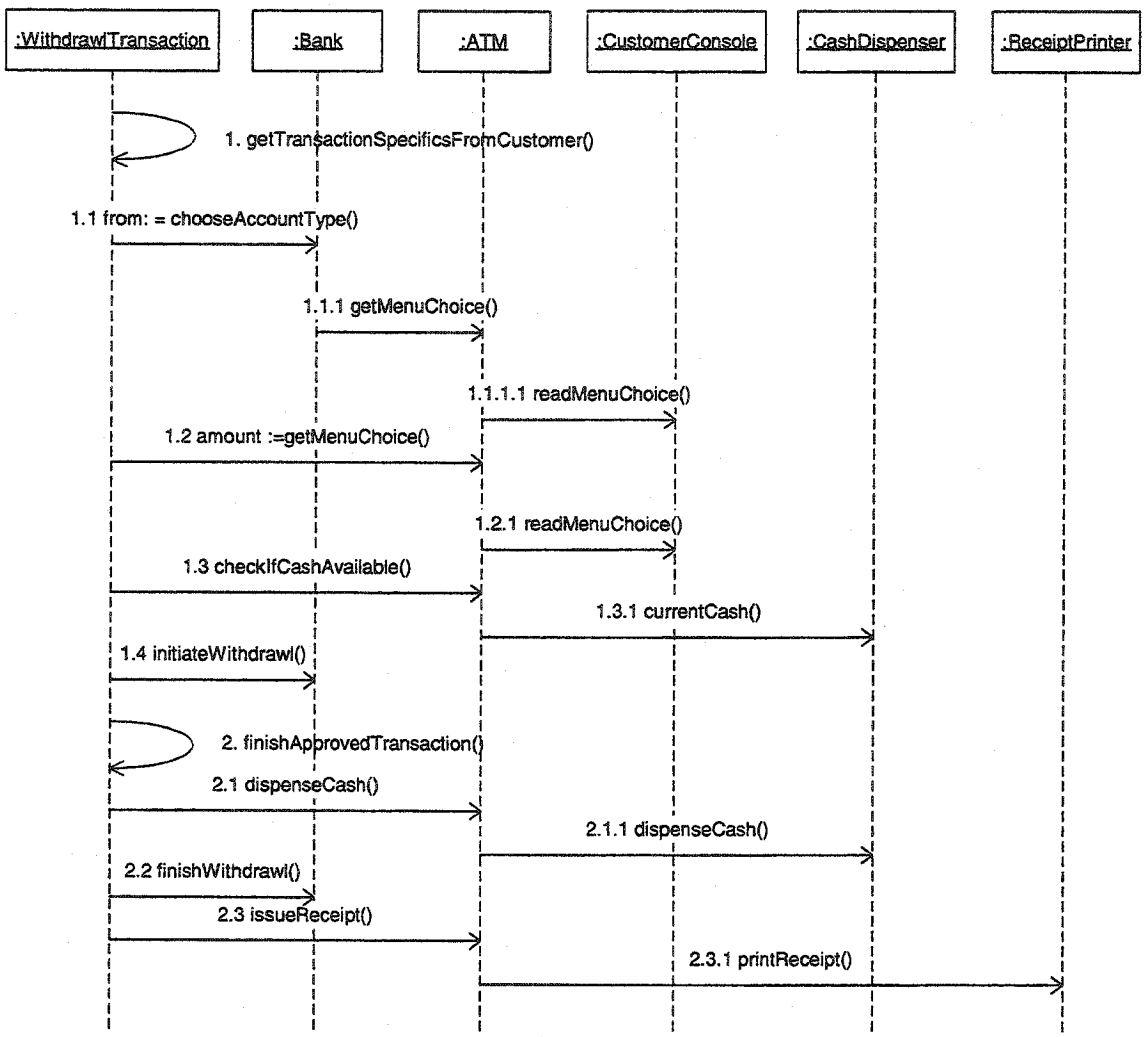


Figure 31 Sequence Diagram of Withdrawal Use Case

The reverse-engineered scenario diagram corresponding to a successful withdrawal is shown in Figure 32. It corresponds to the output file generated by the RESDTool (Appendix J).

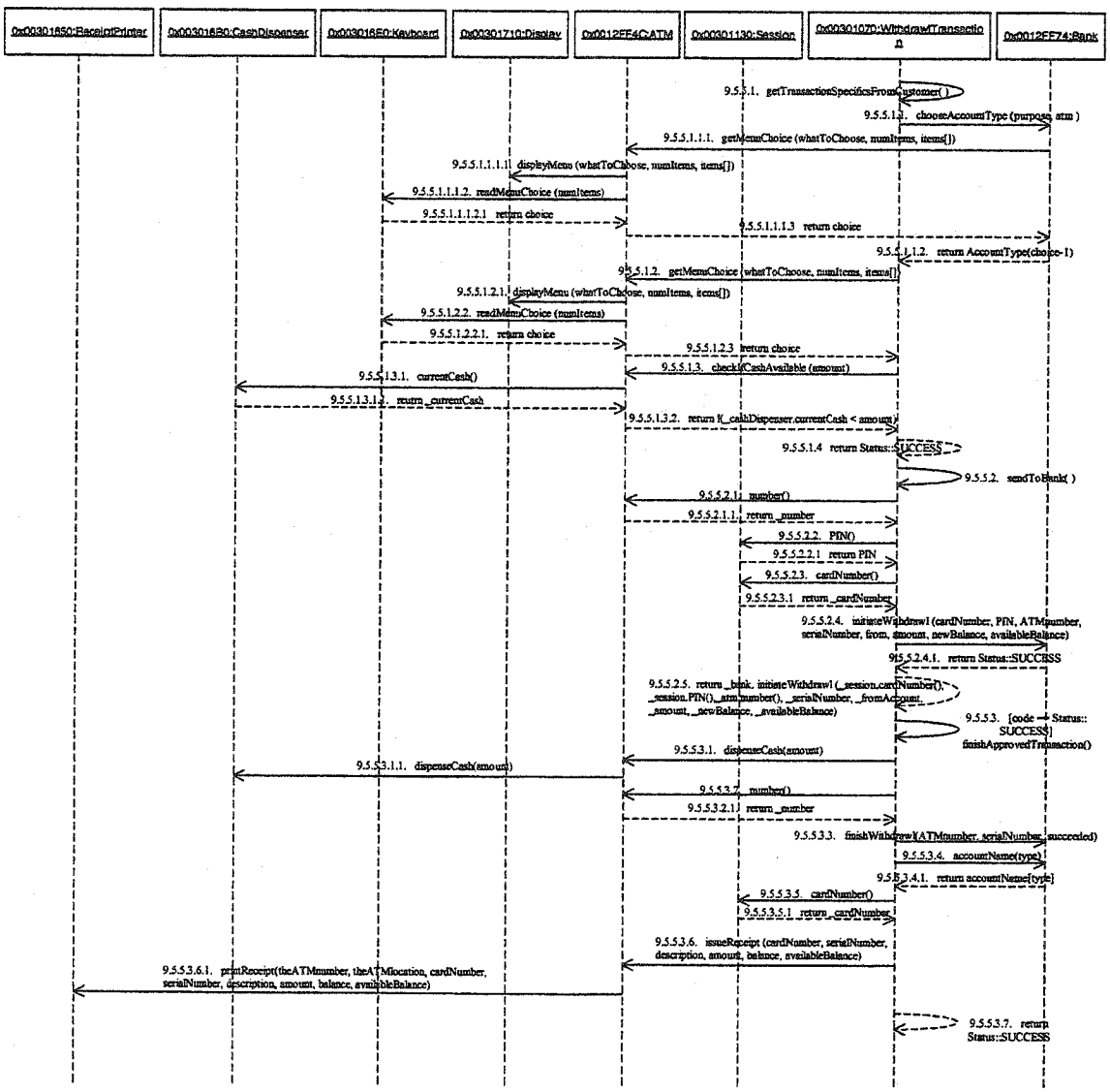


Figure 32 Retrieved Scenario Diagram of Withdrawal Use Case

Comparing the above two diagrams, we find that they are equivalent. Both of the diagrams describe the sequential event flows of withdrawal use case in same order. All the messages in Figure 31, such as *getTransactionSpecificsFromCustomer*, can be found in Figure 32. It proves that our approach is valid and applicable. In addition, the retrieved scenario diagram is more specific. Some detailed messages are missing in Figure 31, which shows that the sequence diagram provided for the ATM system was not complete.

For example, the interactions between objects of *WithdrawlTransaction* and *Session* class, which are shown in the retrieved scenario diagram, are ignored by sequence diagram of withdrawal use case (Figure 31). Objects, which participate in interactions, are represented by a set of unique physical memory addresses in retrieved scenario diagram. Conditions of interactions, return messages and parameters of messages are illustrated in our retrieved scenario diagram as well.

5.3 InvalidPIN Use Case

An invalid PIN use case extension is started within a transaction when the bank reports that the customer's transaction is disapproved due to an invalid PIN. The customer is required to re-enter the PIN and the original request is sent to the bank again. If the customer fails three times to enter the correct PIN, the card is permanently retained, a screen is displayed informing the customer of this and suggesting he/she contact the bank, and the entire customer session is aborted.

Figure 33 shows the sequence diagram of InvalidPIN use case. It is an extension of *Transaction* use case, which denotes that *Transaction* use case extends the flow of events when the invalid PIN was entered.

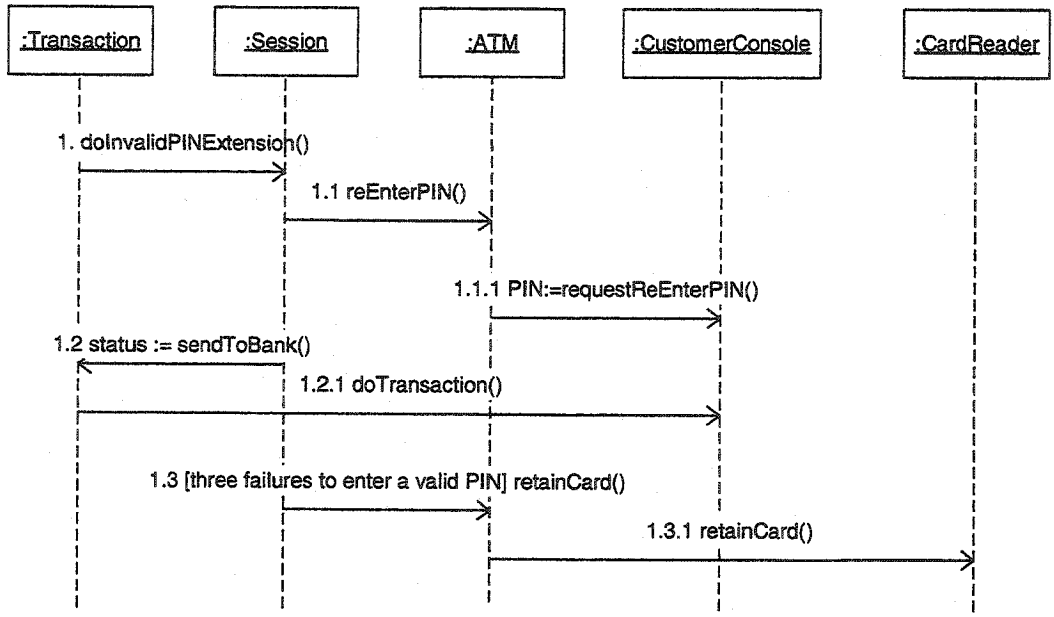


Figure 33 Sequence Diagram of InvalidPIN Use Case

Figure 34 depicts the retrieved scenario diagram of InvalidPIN use case. Since InvalidPIN use case is started within a transaction, the scenario diagram describes the whole flow of events which starts from the withdrawal transaction. Again, the reverse engineered scenario diagram is equivalent to the sequence diagram described in the UML documents, and shows that the provided UML sequence diagram for the InvalidPIN use case was not complete.

From Figure 34, we find that a group of messages are performed under a condition "int i=0; i<3; i++". Although it is shown in the figure, we can also know (from the scenario diagram result file) that this group of messages is executed three times.

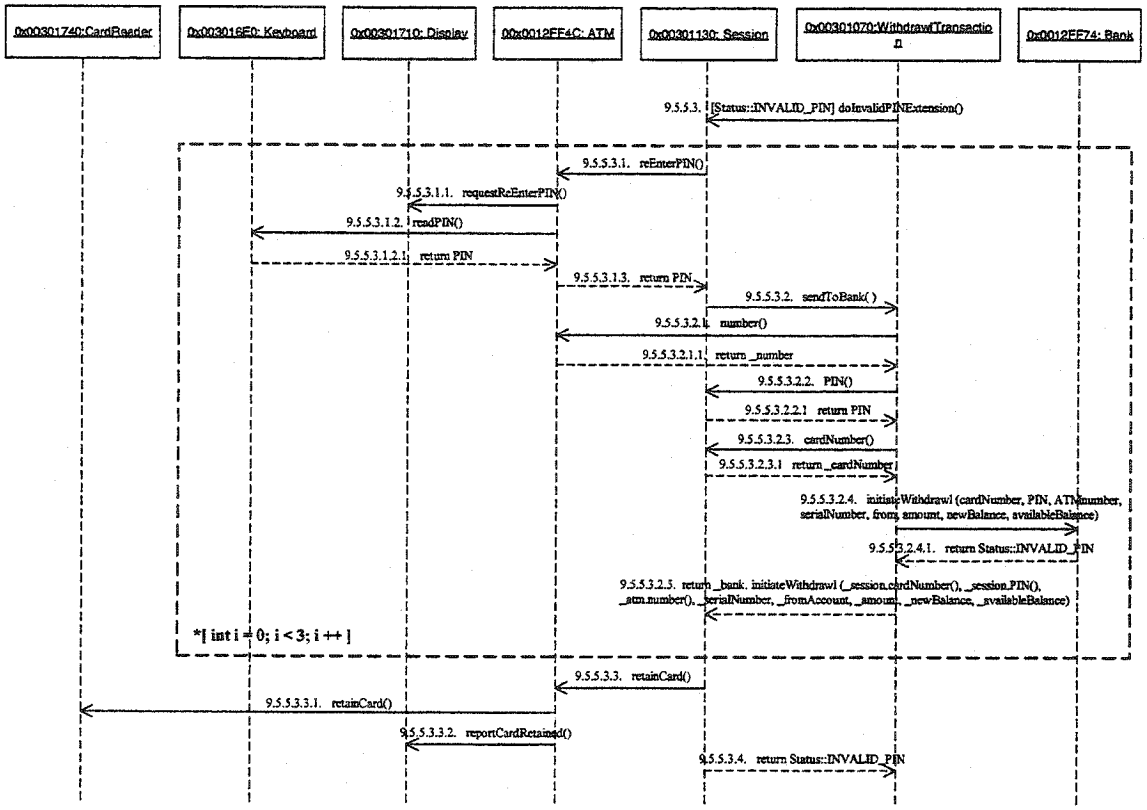


Figure 34 Retrieved Scenario Diagram of InvalidPIN use case

5.4 Lessons Learned

In the previous sections, we used use cases Withdrawal and InvalidPIN as examples to illustrate our approach. Here we summarize the lessons learnt from these use cases and the ones that can be found in Appendix K and Appendix L. The reverse-engineered scenario diagrams (described below as scenario diagrams for simplicity) provided by our approach are equivalent to the sequence diagrams (described below as sequence diagrams), which were available in the UML documentation of ATM system.

On the one hand, the scenario diagrams are much more specific than the sequence diagrams. Every detailed interaction is presented. Also, important interactions depicted in scenario diagrams are not identified clearly in the corresponding sequence diagrams.

After investigation, it appeared that the sequence diagrams should have shown these interactions: the sequence diagrams were not complete. On the other hand, since scenario diagrams only identify objects by their memory addresses (instead of role names for instance), it is difficult do use them along with the source code (it also makes the mapping between scenario and sequence diagrams difficult): the mapping is not obvious.

The scenario diagrams present complete flow of events of use case scenarios, which can correspond to more than one sequence diagram: it is common to split a UML sequence diagram into several sequence diagrams so as not to clutter diagrams (e.g., one wants diagrams to fit in a page when printed). For example, the InvalildPIN use case starts within the Transaction use case. In the forward engineering process, we can split the InvalidPIN use case by depicting two sequence diagrams. In the reverse engineering process, one scenario diagram describes the complete scenario, which make it looks more complex (Figure 34). However, we can divide a complex scenario diagram into smaller ones manually according to the use case diagram. Also, this could be performed by selecting the classes (or methods) of interest, during the instrumentation of the source code, or during the generation of the scenarios.

Chapter 6 Conclusions

Reverse engineering techniques are required to understand the structure and dynamic behavior of a software system whose documentation is missing or out of date. This thesis concentrates on the reverse engineering of the behavior of a software system, rather than its structure. A UML (Unified Modeling Language) sequence diagram is one of the possible notations one can use to describe the behavior of a system. When used during the analysis or design, a sequence diagram is usually associated with a use case of the system and describes a set of (and possibly all the) scenarios of the use case. Another approach is to use the UML notation for sequence diagrams to describe each of the scenarios individually. In the later case, we define the diagram a scenario diagram. The purpose of this thesis is to reverse-engineer scenario diagrams.

The characteristics of the UML sequence diagram notation is first described and analyzed: e.g., object names, flow of control, repetition of messages, conditions. This information guides the source code instrumentation strategy: what are the statements that must be added, and where to be added. A platform and compiler independent approach to instrument C++ source code is described. Trace-generating statements are added into the source codes to record the interaction behaviors of a C++ software system. Although our instrumentation tool is designed based on analysis of C++, it can be extended to instrument Java source code (Java and C++ programming languages are very similar). A powerful and efficient script language, Perl, was chosen to implement the code instrumentation approach.

Once the instrumented source code (possibly several files) has been re-compiled and executed, dynamic information about the execution of the operations, return statements and control flow structures, which are generated by trace-generating statements, is collected in a trace file.

Our methodology for the production of scenario diagrams from traces has been formally defined. Both traces and scenario diagrams are first modeled using a class diagram (metamodels). Then rules, defined in OCL (Object Constraint Language [Warmer+ 99]), specify how a trace (i.e., an instance of the TRACE metamodel) is transformed into a scenario (i.e., an instance of the SD metamodel). These rules specify how operations between objects, as well as their repetition (loops), and the condition under which they are triggered, are identified in a trace and how the corresponding messages are created in a scenario diagram.

These OCL rules were a formal foundation for the development of precise algorithms, and a prototype tool has been developed. This tool has been used to reverse-engineer several use cases of an Automated Teller Machine, thus showing the feasibility of the approach.

In our approach, a reverse-engineered scenario diagram is presented in a human readable text form, which can be easily transformed into any case tool specific format (e.g., Rational Rose), or transformed into an XMI file (an inter-change format between UML case tools).

In our approach, we assume that the physical memory address of an object will not be changed during the life time of the object. Therefore, we use memory address to identify objects. However, in some cases, the memory address of an object can be

changed or swapped, e.g., an object in virtual memory. Our code instrumentation methodology is applied to the sequential, non-distributed system only. More research is needed to apply our methodology to real-time, concurrent and distributed system in future.

Two main research directions can be investigated in the future:

(1) Extensions to the instrumentation step

(a) More instrumentation can be done to get as many information as possible. This includes the use of reference names (e.g., attribute names, local variable names) instead of memory addresses to identify objects. This would help a lot the understanding of the system under study as the mapping between the scenario (or sequence) diagrams and the source code would be more straightforward. This is however a challenge because of object aliasing (different variable names referencing the same object). A similar issue is the use of actual parameters instead of formal parameters when operations are detected at run time. Note that these two issues require a more substantial static analysis of the source code, than the one performed in this thesis.

(b) The instrumentation of the source code, and the run time generation of execution traces can be improved. For instance, instead of opening and closing the trace file each time a log is recorded (e.g., each time there is an operation, each time there is a control flow structure), a static method (of a GenerateTrace class for instance) can be used to handle the storage of trace information into the trace file.

(2) Extension to the scenario/sequence diagram retrieval

(a) In our approach, 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 it would be interesting to simplify it.

(b) A query system can be developed to limit the range of investigation, that is select classes (or interaction) of interest, thus concentrating only on the major interactions. For instance the user may not want to report on classes that are part of the GUI, or would like a report only on specific classes. Note that such functionality can also be used at the instrumentation phase (the use may choose not to instrument GUI classes). An approach similar to the Collaboration Browser (see the Related Work section) could then be developed. This query system would also be used to group scenario diagrams into sequence diagrams.

References

[Booch+ 99] Grandy Booch, James Rumbaugh, Ivar Jacobson, The Unified Modeling Language User Guide, Addisino-Wesley, 1999.

[Bruegge+ 00] Bernd Bruegge, Allen H. Dutoit, Object-Oriented Software Engineering: Conquering Complex and Changing Systems, Prentice Hall, 2000.

[Christiansen 98] Tom Christiansen, Nathan Torkingston, Perl Cookbook, O'Reilly & Associates, Inc. 1998.

[Deitel+ 98] H. M. Deitel, P. J. Deitel, C++ How To Program, 2nd Ed., Prentice-Hall, Inc. 1998.

[Erikssion+ 98] Hans-Erik Eriksson, Magnus Penker, UML Toolkit, John Wiley & Sons, Inc. 1998.

[Gamma+ 98] Enrich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Pattern, Elements of Reusable Object-Oriented Software, Addision-Wesley, 1998.

[J2SE™ 1.4] http://java.sun.com/j2se/1.4/

[Jerding+ 97a] Dean Jerding, John T. Stasko, and Thomas Ball, "*Visualizing Interactions in Program Executions*", Proceeding of the International Conference on Software Engineering, 1997, pp. 360-370.

[Jerding+ 97b] Dean Jerding and Spencer Rugaber, "*Using Visualization for Architectural Localization and Extraction*", In Proceedings of the Working Conference on Reverse Engineering (WCRE) 1997, pp. 56-65.

[Lange+ 95a] Danny B. Lange and Yuichi Nakamura, "*Interactive Visualization of Design Patterns Can Help in Framework Understanding*", Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 1995, pp.342-357.

[Lange+ 95b] Danny B. Lange and Yuichi Nakamura, *"Program Explorer: A Program Visualizer for C++"*, Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS), 1995, pp.39-54.

[Mendonca+ 96] Nabor C. Mendonca, Jeff Kramer, "*Requirements for an Effective Architecture Recovery Framework*", Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops, San Francisco, California, United States, pp101-105, 1996.

[Rational Rose] http://www.rational.com/products/rose/

[Richner+ 02] Tamar Richner and Stephance Ducasse, "*Using Dynamic Information for the Iterative Recovery of Collaborations and Roles*", Proceeding of International Conference on Software Maintenance, 2002, pp. 34-43.

[Systä 99a] Tarja Systä, "*On the Relationships between Static and Dynamic Models in Reverse Engineering Java Software*", Proceeding of Working Conference on Reverse Engineering (WCRE), 1999, pp. 304-313.

[Systä 99b] Tarja Systä, "*Dynamic reverse engineering of Java software*", Proceeding of the European Conference on Object-Oriented Programming (ECOOP) Workshop on Experiences in Object-Oriented Re-Engineering, 1999.

[Systä 00] Tarja Systä, "*Understanding the Behavior of Java Programs*", Proceeding of Working Conference on Reverse Engineering (WCRE), 2000, pp. 214-223.

[TogetherSoft] http://www.togethersoft.com

[UML] http://www.omg.org/technology/documents/formal/uml.htm

[Walker+ 98] R.Walker, G.Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak, "*Visualizing dynamic software system information through high-level models*", Proceeding of the 1998 ACM Conference on Object-Oriented Programming, System, Language, and Application (OOPSLA), 1998, pp. 271-283.

[Wall 00] Larry Wall, Tom Christiansen, and Jon Orwant, Programming Perl, 3rd Ed., O'Reilly & Associates, Inc. 2000.

[Warmer+ 99] J. Warmer and A. Kleppe, The Object Constraint Language: Precise Modeling with UML, Addison-Wesley, 1999.

[XMI] http://www.omg.org/technology/documents/formal/xmi.htm

Appendix A Semantics of Method Definition in C++

In any computer language, there are always certain patterns to express methods. So is there in C++. There are two ways to define a method: methodDefinition1 and methodDefinition2.

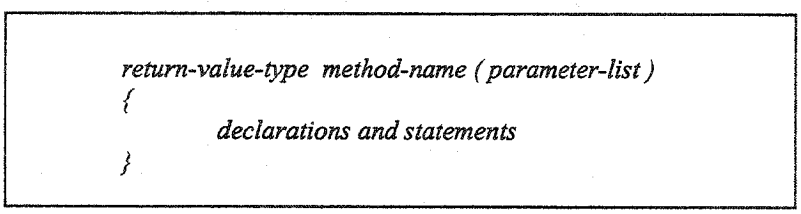


Figure 35 Format of methodDefinition1

Figure 35 shows the format of methodDefinition1. The *method-name* is any valid identifier. The *return-value-type* is the data type of the result returned from the method to the caller. The *return-value-type* can be **void** which indicates that a method does not return a value. The *parameter-list* is a comma-separated list containing the declarations of the parameters received by the method when it is called. If a method does not receive any values, *parameter-list* is void or simply left empty. The *declarations and statements* in *braces* form the method body. A method cannot be defined inside another method under any circumstances; otherwise, it will be a syntax error.

As an object-oriented high-level language, the focus of C++ is on classes rather than methods. Interface is usually separated from implementation. The definitions of the class member functions in a separate implementation source file, which is methodDefinition2, can be expressed as follows:

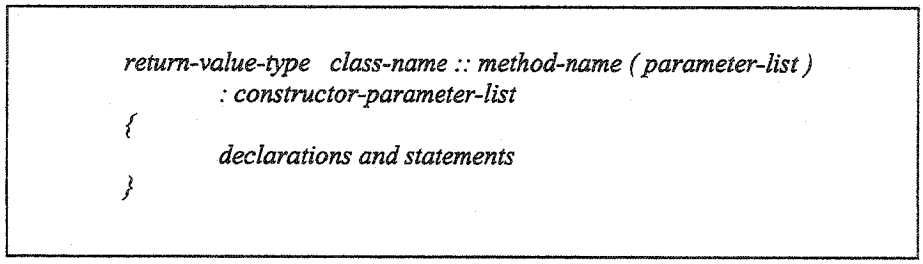


Figure 36 Format of methodDefinition2

The *class-name* is specified to which class's scope the method belongs. All the member functions are declared in the class definition. The *constructor-parameter-list* is optional in member function definition. It will be ignored while executing our instrumentation tool.

Appendix B Algorithm of Code InstrTC++

B.1 Instrumentation Algorithm

This algorithm is designed base on the flow chart (1) of code instrumentation, which is described in Figure 18. It finds and analyzes definitions of methods in C++ source code. To avoid modifying the original source code file, all the source codes and trace-generating statements are written into a new file. We name the new file after the original source code file, and rename the source code file as well after code instrumentation is completed. So, user can re-compile and run the program without any awareness.

Input/Output: **var** sourceFile :File;

Algorithm:

```
Procedure Instrumentation (sourceFile :File);
var  remainBuffer, thePattern :String;
     newFile :File;
begin
     open sourceFile;
     create and open newFile;
     read a line in sourceFile and assign it to remainBuffer;
     while (not the end of sourceFile) or (remainBuffer is not empty) do
     begin
          thePattern := findPattern(sourceFile,newFile,remainBuffer, className);
          if thePattern is a method pattern then
               processMethod(sourceFile,newFile,remainBuffer,thePattern,className);
     end
     close sourceFile and rename it as sourceFile.org;
     close newFile and rename it as sourceFile;
end
```

B.2 Procedure processMethod algorithm

This algorithm is created based on the flow chart (2) of code instrumentation, which is described in Figure 19. It analyzes a method and augments the method with trace-generating statements. The "elseConditionStack" stores all the condition of "if" control flow structures. The opposite of those conditions can be the conditions of the corresponding "else" control flow structures.

Input: **var** sourceFile: File;

var thePattern, className :String;

Input/Output: **var** newFile: File;

var remainBuffer :String;

Algorithm

```
Procedure processMethod (sourceFile, newFile :File; remainBuffer, thePattern, className
:String);
var theBuffer :String;
    elseConditionStack :Stack;
begin
       augments the source codes with trace-generating statements at the beginning of the
       class definition body to indicate that a method call begins;
       while the end of this method is not reached do
       begin
         if remainBuffer is null then
                    readln(sourceFile,newFile,theBuffer, remainBuffer);
         else
                    processBuffer(theBuffer, remainBuffer);
        if theBuffer contains a keyword of any control flow structure or return
              statement then begin
            prefix the remainBuffer with theBuffer;
            processControlFlow(sourceFile, newFile, remainBuffer, className,
                                          elseConditionStack);
         end
         else begin
             write the string of theBuffer into newFile;
             if remainBuffer is not null then
                 processComment(sourceFile, newFile, remainBuffer);
         end
       end
       augments the source code with trace-generating statements at the end of the class
       definition body to indicate that a method call completes;
end
```

B.3 Procedure processControlFlow algorithm

This algorithm is designed based on the flow chart (3) of code instrumentation which is described in Figure 20. It analyzes a control flow structure and augments it with trace-generating statements. Different control flow structures will be analyzed using corresponding sub-algorithms, which are described later.

A control flow structure can be defined in another control flow structure. A recursive analysis can be applied.

Input: var sourceFile: File;
var lassName :String;

Input/Output: var newFile: File;
var remainBuffer :String;
var elseConditionStack :Stack;

Algorithm

```
Procedure processControlFlow (sourceFile, newFile :File; remainBuffer, className :String;
elseConditionStack :Stack);
var theBuffer :String;
begin
    if remainBuffer is null then
            readln(sourceFile,newFile,theBuffer, remainBuffer);
    else
            processBuffer(theBuffer, remainBuffer);
    write the string before the keyword in theBuffer;
    prefix the remainBuffer with the string after the keyword in Buffer;
    if the keyword = "if" then
        processIf(sourceFile,newFile,remainBuffer, className, elseConditionStack);
    else if the keyword = "else" then
        processElse(sourceFile,newFile,remainBuffer,className,elseConditionStack);
    else if the keyword = "for" then
        processFor(sourceFile,newFile,remainBuffer, className,elseConditionStack);
    else if the keyword = "while" then
        processWhile(sourceFile,newFile,remainBuffer,className,elseConditionStack);
    else if the keyword = "do" then
        processDo(sourceFile,newFile,remainBuffer, className, elseConditionStack);
    else if the keyword = "switch" then
        processSwitch(sourceFile,newFile,remainBuffer,className,elseConditionStack);
    else if the keyword = "return" then
        processReturn(sourceFile,newFile,className);
    else if the keyword = "break" then
        processBreak(newFile, className);
    else if the keyword = "continue" then
        processContinue(newFile, className);
end
```

B.4 Auxiliary algorithms

Several algorithms are designed to perform the *instrumentation*, *processMethod* and *processControlFlow* algorithms.

B.4.1 Function findPattern

This is a function to detect the definition pattern of a methodDefinition1 or methodDefinition2 according to the C++ semantics. A matched pattern will be returned when the algorithm is performed.

Input: **var** sourceFile: File;
Output: **var** thePattern :String;
Input/Output: **var** newFile: File;
var remainBuffer, className :String;

Algorithm

```
Function findPattern (sourceFile, newFile :File; remainBuffer, className :String)
thePattern :String;
var theBuffer, methodBuffer, aString :String;
    found :Boolean;
begin
    methodBuffer := "";
    found := false;
    while ((not the end of the sourceFile) or (remainBuffer is not empty)) and not
       found do begin
        if remainBuffer is null then
                    readln(sourceFile,newFile,theBuffer, remainBuffer);
        else
                    processBuffer(theBuffer, remainBuffer);
        methodBuffer := methodBuffer + theBuffer;
        if methodBuffer contains the definition of a class then
             className := name of the class indicated in the definition;
        if methodBuffer contains the definition of a method then
        begin
             aString:= the string after the definition format part in theBuffer;
             write the string in theBuffer except aString into newFile;
             thePattern := the string rpresents the match pattern;
             make a prefix of remainBuffer with aString;
             found := true;
             if thePattern is a methodDefinition2 then
                 className = "";
        end
        else begin
             write theBuffer into newFile;
             if remainBuffer is not null then
                  processComment(sourceFile, newFile, remainBuffer);
             if methodBuffer contains ";" then
                  assign the string which is after the ";" to methodBuffer;
        end
    end
end
```

B.4.2 Procedure readln

This algorithm reads a source code line, omits preprocessor commands and separates the statement from the comment in this line.

Input: **var** sourceFile: File;

Input/Output: **var** newFile: File;

var theBuffer, remainBuffer :String;

Algorithm

```
Procedure readln (sourceFile, newFile :File; theBuffer,remainBuffer :String);
var bufferRead :String;
begin
    assign a code line of sourceFile to bufferRead;
    if bufferRead starts with preprocessor command sign "#" then
    begin
        write the string of bufferRead into newFile;
        theBuffer := "";
        remainBuffer := "";
    end
    else if bufferRead contains comment sign "//" or "/*" then begin
        theBuffer := string before comment sign in bufferRead;
        remainBuffer := string starts with comment sign in bufferRead;
    end
    else begin
        theBuffer := bufferRead;
        remainBuffer := "";
    end
end
```

B.4.3 Procedure processBuffer

A string is analyzed to separate the statement from the comment in this string.

Input/Output: **var** theBuffer, remainBuffer :String;

Algorithm

```
Procedure processBuffer (theBuffer, remainBuffer :String);
var bufferProcessing :String;
begin
    assign the string of remainBuffer to bufferProcessing;
    if bufferProcessing contains comment sign "//" or "/*" then begin
        theBuffer := string before comment sign in bufferProcessing;
        remainBuffer := string starts with comment sign in bufferProcessing;
    end
    else begin
        theBuffer := bufferProcessing;
        remainBuffer := "";
    end
end
```

B.4.4 Procedure processComment

Comments are omitted by applying this algorithm.

Input: **var** sourceFile: File;

Input/Output: **var** newFile: File;

var remainBuffer :String;

Algorithm

```
Procedure processComment (sourceFile, newFile :File; remainBuffer :String);
var commentEnd :Boolean;
begin
    commentEnd := false;
    if remainBuffer starts with "/*" then
    begin
        while not commentEnd do
        begin
            if remainBuffer contains the comment end sign "*/" then
            begin
                write the string before and including "*/" in remainBuffer into
                newFile;
                remainBuffer := the string after "*/" in remainBuffer;
                commentEnd := true;
            end
            else begin
                print remainBuffer into newFile;
                assign the next code line in sourceFile to remainBuffer;
            end
        end
    end
    else begin
        write the string of remainBuffer into newFile;
        remainBuffer := "";
    end
end
```

B.4.5 Procedure processIf

This algorithm is designed to process an "if" control flow structure, augment it with trace-generating statements. Another control flow structure may be analyzed inside this control flow structure.

Input: var sourceFile: File;

var className :String;

Input/Output: var newFile: File;

var remainBuffer :String;

var elseConditionStack :Stack;

Algorithm

```
Procedure processIf (sourceFile, newFile :File; remainBuffer, className :String;
elseConditionStack :Stack);
var theBuffer :String;
begin
     find the condition statement;
     store the condition into elseConditionStack;
     write the definition format of the control flow structure into newFile;
     write the trace-generating statements into newFile to indicate that a control flow
     structure begins;
     while the end of this control flow structure is not reached do
     begin
        if remainBuffer is null then
                    readln(sourceFile,newFile,theBuffer, remainBuffer);
        else
                    processBuffer(theBuffer, remainBuffer);
        if theBuffer contains any keyword of control flow structure or return statement
        then begin
             prefix the string of remainBuffer with the string of theBuffer;
             processControlFlow(sourceFile, newFile, remainBuffer, className,
                              elseConditionStack);
        end
        else begin
                write theBuffer into newFile;
                if remainBuffer is not null then
                    processComment(sourceFile, newFile, remainBuffer);
        end
     end
     write the trace-generating statements into newFile to indicate that a control flow
     structure completes;
end
```

B.4.6 Procedure processElse

This algorithm is designed to process a "else" control flow structure, augment it with trace-generating statements. Another control flow structure may be analyzed inside this control flow structure.

Input: var sourceFile: File;

var className :String;

Input/Output: var newFile: File;

var remainBuffer :String;

var elseConditionStack :Stack;

Algorithm

```
Procedure processElse (sourceFile, newFile :File; remainBuffer, className :String;
elseConditionStack :Stack);
var theBuffer :String;
begin
    take the condition from elseConditionStack, make it opposite to its original value;
    write the definition format of the control flow structure into newFile;
    write the trace-generating statements into newFile to indicate that a control flow
    structure begins;
    while the end of this control flow structure is not reached do
    begin
          if remainBuffer is null then
              readln(sourceFile,newFile,theBuffer, remainBuffer);
          else
              processBuffer(theBuffer, remainBuffer);
          if theBuffer contains any keyword of control flow structure or return statement
          then begin
              prefix the string of remainBuffer with the string of theBuffer;
              processControlFlow(sourceFile, newFile, remainBuffer, className,
                    elseConditionStack);
          end
          else begin
              write theBuffer into newFile;
              if remainBuffer is not null then
                  processComment(sourceFile, newFile, remainBuffer);
          end
    end
    write the trace-generating statements into newFile to indicate that a control flow
    structure completes;
end
```

B.4.7 Procedure processFor

This algorithm is designed to process a "for" control flow structure, augment it with trace-generating statements. Another control flow structure may be analyzed inside this control flow structure.

Input: **var** sourceFile: File;

var className :String;

Input/Output: **var** newFile: File;

var remainBuffer :String;

var elseConditionStack :Stack;

Algorithm

```
Procedure processFor (sourceFile, newFile :File; remainBuffer, className :String;
elseConditionStack :Stack);
var theBuffer :String;
begin
    find the condition statement;
    write the definition format of the control flow structure into newFile;
    write the trace-generating statements into newFile to indicate that a control flow
    structure begins;
    while the end of this control flow structure is not reached do
    begin
          if remainBuffer is null then
              readln(sourceFile,newFile,theBuffer, remainBuffer);
          else
              processBuffer(theBuffer, remainBuffer);
          if theBuffer contains any keyword of control flow structure or return statement
          then begin
              prefix the string of remainBuffer with the string of theBuffer;
              processControlFlow(sourceFile, newFile, remainBuffer, className,
                    elseConditionStack);
          end
          else begin
              write theBuffer into newFile;
              if remainBuffer is not null then
                  processComment(sourceFile, newFile, remainBuffer);
          end
    end
    write the trace-generating statements into newFile to indicate that a control flow
    structure completes;
end
```

B.4.8 Procedure processDo

This algorithm is designed to process a "do/while" control flow structure, augment it with trace-generating statements. Another control flow structure may be analyzed inside this control flow structure.

Input: var sourceFile: File;

var className :String;

Input/Output: var newFile: File;

var remainBuffer :String;

var elseConditionStack :Stack;

Algorithm

```
Procedure processDo (sourceFile, newFile :File; remainBuffer, className :String;
elseConditionStack :Stack);
var theBuffer :String;
begin
    find the condition statement;
    write the definition format of the control flow structure into newFile;
    write the trace-generating statements into newFile to indicate that a control flow
    structure begins;
    while the end of this control flow structure is not reached do
    begin
          if remainBuffer is null then
              readln(sourceFile,newFile,theBuffer, remainBuffer);
          else
              processBuffer(theBuffer, remainBuffer);
          if theBuffer contains any keyword of control flow structure or return statement
          then begin
              prefix the string of remainBuffer with the string of theBuffer;
              processControlFlow(sourceFile, newFile, remainBuffer, className,
                    elseConditionStack);
          end
          else begin
              write theBuffer into newFile;
              if remainBuffer is not null then
                  processComment(sourceFile, newFile, remainBuffer);
          end
    end
    write the trace-generating statements into newFile to indicate that a control flow
    structure completes;
end
```

B.4.9 Procedure processWhile

This algorithm is designed to process a "while" control flow structure, augment it with trace-generating statements. Another control flow structure may be analyzed inside this control flow structure.

Input: var sourceFile: File;

var className :String;

Input/Output: var newFile: File;

var remainBuffer :String;

var elseConditionStack :Stack;

Algorithm

```
Procedure processWhile (sourceFile, newFile :File; remainBuffer, className :String;
elseConditionStack :Stack);
var theBuffer :String;
begin
    find the condition statement;
    write the definition format of the control flow structure into newFile;
    write the trace-generating statements into newFile to indicate that a control flow
    structure begins;
    while the end of this control flow structure is not reached do
    begin
          if remainBuffer is null then
              readln(sourceFile,newFile,theBuffer, remainBuffer);
          else
              processBuffer(theBuffer, remainBuffer);
          if theBuffer contains any keyword of control flow structure or return statement
          then begin
              prefix the string of remainBuffer with the string of theBuffer;
              processControlFlow(sourceFile, newFile, remainBuffer, className,
                    elseConditionStack);
          end
          else begin
              write theBuffer into newFile;
              if remainBuffer is not null then
                  processComment(sourceFile, newFile, remainBuffer);
          end
    end
    write the trace-generating statements into newFile to indicate that a control flow
    structure completes;
end
```

B.4.10 Procedure processSwitch

This algorithm is designed to process a "switch/case" control flow structure, augment it with trace-generating statements. Another control flow structure may be analyzed inside this control flow structure.

Input:	**var**	sourceFile: File;
	var	className :String;
Input/Output:	**var**	newFile: File;
	var	remainBuffer :String;
	var	elseConditionStack :Stack;

Algorithm

```
Procedure processSwitch (sourceFile, newFile :File; remainBuffer, className :String;
elseConditionStack :Stack);
var theBuffer :String;
begin
    find the control expression;
    write the "switch" definition format of the control flow structure into newFile;
    while the end of this control flow structure is not reached do
    begin
          if remainBuffer is null then
              readln(sourceFile,newFile,theBuffer, remainBuffer);
          else
              processBuffer(theBuffer, remainBuffer);
          if theBuffer contains keyword "case" then
          begin
              write the "case :" into newFile;
              write the trace-generating statements into newFile to indicate that a case
              statement is going to be performed.
              write the case statement into newFile;
          end
          else if theBuffer contains keyword "default" then
          begin
              write the "default:" into newFile;
              write the trace-generating statements into newFile to indicate that a
              default statement of a "switch/case" control flow structure is going to be
              performed;
              write the default statement into newFile;
          end
          else if theBuffer contains any keyword of control flow structure or return
          statement then begin
              prefix the string of remainBuffer with the string of theBuffer;
              processControlFlow(sourceFile, newFile, remainBuffer, className,
                      elseConditionStack);
          end
          else begin
              write theBuffer into newFile;
              if remainBuffer is not null then
                  processComment(sourceFile, newFile, remainBuffer);
          end
    end
    write the trace-generating statements into newFile to indicate that a control flow
    structure completes;
end
```

B.4.11 Procedure processReturn

This algorithm is designed to process a return statement and augments it with trace-generating statements.

Input: var sourceFile: File;

var className :String;

Input/Output: var newFile: File;

Algorithm

```
Procedure processReturn (sourceFile, newFile :File; className :String);
begin
      find the statement of the return statement;
      write the trace-generating statements into newFile to indicate that a return
      statement is going to be performed;
      write the return statementn into the newFile;
end
```

B.4.12 Procedure processBreak

This algorithm is designed to augments a "break" command with trace-generating statements.

Input: var className :String;

Input/Output: var newFile: File;

Algorithm

```
Procedure processBreak (newFile :File; className :String);
begin
    write the trace-generating statements into newFile to indicate that a "break" command
    is going to be performed;
    write the "break" source code into the newFile;
end
```

B.4.13 Procedure processContinue

This algorithm is designed to augments a "continue" command with trace-generating statements.

Input: var className :String;

Input/Output: var newFile: File;

Algorithm

```
Procedure processBreak (newFile :File; className :String);
begin
    write the trace-generating statements into newFile to indicate that a "continue"
    command is going to be performed;
    write the "continue" source code into the newFile;
end
```

Appendix C Sequence Diagrams of RESDTool

C.1 Analyze Trace File

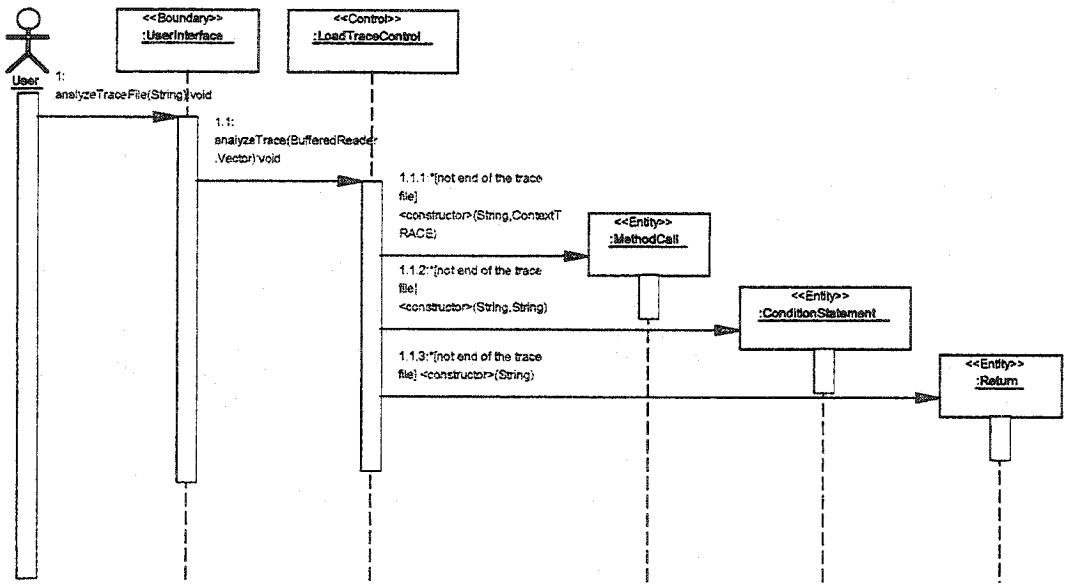


Figure 37 Sequence Diagram: Analyze Trace File

C.2 Load Trace File

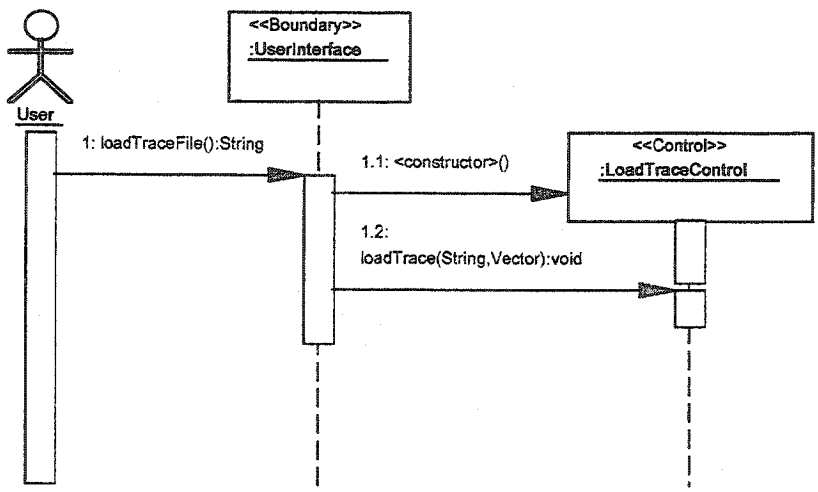


Figure 38 Sequence Diagram: Load Trace File

C.3 Print Messages

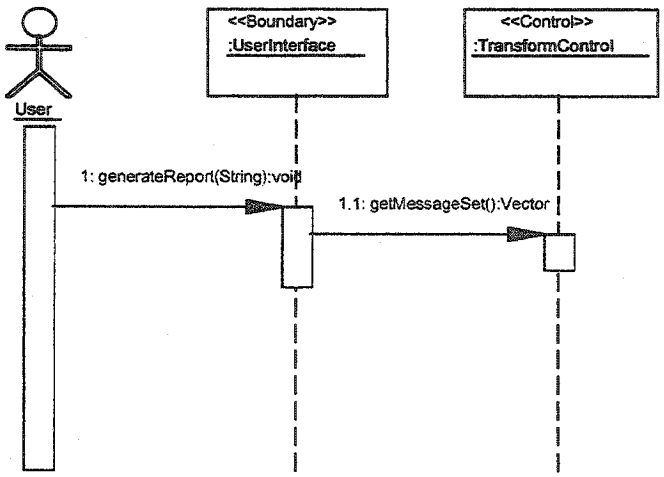


Figure 39 Sequence Diagram: Print Messages

C.4 Transform Traces

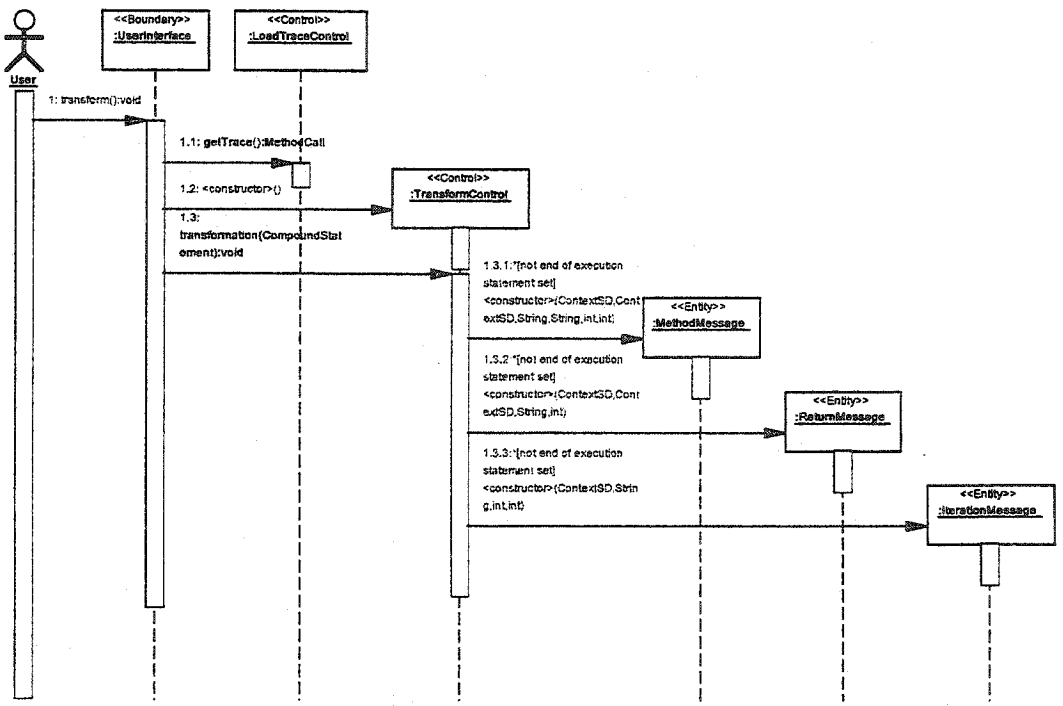


Figure 40 Sequence Diagram: Transform Traces

Appendix D Data Dictionary of RESDTool

D.1 Entity Classes

ClassSD	The class of object which is involved in the message interaction. The attribute of this class is `name` to represent the name of the class.
ClassTRACE	The class of object which is involved in the statement execution. The attribute of this class is `name` to represent the name of the class.
CompoundStatement	An abstract class. The attribute `executionStatementSeq` indicates that this object is followed by some `MethodCall`, `ConditionStatement` and `Return` objects
ConditionClauseSD	`ConditionClauseSD` class stores the information of a condition clause in SD subsystem. The attribute of this class is `cluaseKind` and `clauseStatement`.
ConditionClauseTRACE	`ConditionClauseSD` class stores the information of a condition clause in TRACE subsystem. The attribute of this class is `cluaseKind` and `clauseStatement`.
ConditionStatement	The class indicates that a control flow structure is executed. It has an attributes `kindOfCondition` to show the kind of the control flow structure such as "if" type.
ContextSD	An abstract class to indicate that a message is made in some context which can be objects or classes.
ContextTRACE	An abstract class to indicate that a method call is made between in some context which can be objects or classes.
ExecutionStatement	ExecutionStatement is an abstract class. It indicates that a certain statement has been executed. The attribute of this class is `statement`.
InstanceSD	The class represents the objects which involve in the interactions in a target system. This class is part of `SD` subsystem. The attribute of this class is `addressID`.
InstanceTRACE	The class represents the objects which involve in the interactions in the target system. This class is part of `Trace` subsystem. The attribute of this class is `id` which is the physical address of this instance.
IterationMessage	The class indicates that a group of iterative messages exist.
Message	An abstract class to indicate that a message is sent. The attribute of this class is `content` to present the statement of a message.

MethodCall	It is a subclass of `ExecutionStatement`. It indicates that a method call is executed. It has an attribute `returnType`. If there is any parameter taken in the method call, it will be stored in attribute `parameterSet`.
MethodMessage	The `MethodMessage` class is responsible to provide the information of method calls messages. The attribute of this class are `returnType` and `timesOfRepeat`.
ParameterSD	The class stores the arguments of a method call message. The attributes of this class are `name` and `type`.
ParameterTRACE	The class stores the arguments of a method call. The attributes of this class are `name` and `type`.
Return	It is a subclass of `ExecutionStatement`. It indicates that a return statement is executed.
ReturnMessage	The class represents that a return message is sent

D.2 Boundary Class

UserInteraface	This class provides the user interface. The user can select the desired functions from the menu.

D.3 Control Classes

LoadTraceControl	LoadTraceControl loads a trace file into the system. It also manages the parse function to convert and translate the information of log lines into `ExecutionStatement` object. The attribute of this class is `main_trace`.
TransformControl	TransformControl defines the transformation rules, transform the information of method executions and return execution statements into messages. The attributes of this class are `messageSeq`.

D.4 Attributes

addressID	A string presents the memory address of an instance object in SD subsystem, which is unique ID of the instance object.
ConditinoClauseSD::clauseKind	The type of a condition clause in SD subsystem, such as "if".
ConditionClauseTRACE::clauseKind	The type of a condition clause in TRACE subsystem, such as "if".
ConditionClauseSD::clauseSt	The content of a condition clause in SD subsystem.

atement	
ConditionClauseTRACE::cla useStatement	The content of a condition clause in TRACE subsystem.
Content	A string which indicates the statement of invocated messages.
Id	A string presents the memory address of an instance object in TRACE package, which is unique ID of the instance object.
isLoop	A boolean variable indicates whether the control flow structure is a loop structure.
Iteration	An integer to show the iteration of a group of iterative messages.
kindOfCondition	The kind of a control flow structure.
main_trace	A method call which represents the "main" method call in a program. It is the basis of all the statement execution.
messageSeq	A set of messages, which compose a scenario diagram.
ClassSD::name	A string represents the name of the `Class` object in SD subsystem.
ClassTRACE::name	A string represents the name of the `Class` object in TRACE subsystem.
ParameterSD::name	A string represents the name of a `Parameter` object in SD subsystem.
ParameterTRACE::name	A string represents the name of a `Parameter` object in TRACE subsystem.
MethodCall::returnType	A string represents the return type of a method call.
MethodMessage::returnType	A string represents the return type of a method call message.
Statement	A string represents the source code statement of an executed statement.
timesOfRepeat	An integer to show the iteration of a method call.
ParameterSD::type	A string represents the type of a parameter in SD subsystem.
ParameterTRACE::type	A string represents the type of a parameter in TRACE subsystem.

D.5 Operations

analyzeTrace(BufferReader, Vector, Vector):void	Analyze a trace file.
analyzeTraceFile(String):void	The request to analyze each log line of a trace file, it takes a string parameter which indicates the path of the trace file.
generateReport():void	The request to get a set of messages which compose a scenario diagram.
ConditionClauseSD::getClauseKind():String	Gets the kind of a condition clause such as "While" in SD subsystem.
ConditionClauseTRACE::getClauseKind():Str	Gets the type of a condition clause such as "While" in

ing	TRACE subsystem.
ConditionClauseSD::getClauseStatement():String	Gets the statement of a condition clause in SD subsystem.
ConditionClauseTRACE::getClauseStatement():String	Gets the statement of a condition clause in TRACE subsystem.
getContent():String	Get the statement of a message.
getContext():ContextTRACE	Gets the context in which a method call is made.
InstanceSD::getID():String	Gets the unique id of an InstanceSD object.
InstanceTRACE::getID():String	Gets the unique id of an InstanceTRACE object.
getKindOfCondition():String	Gets the kind of the a condition statement.
getMessageSet():Vector	Gets a set of message which compose a scenario diagram.
getMyClass:String	Gets the name of ClassTRACE object.
ClassSD::getName():String	Gets the name of ClassSD object.
ParameterSD::getName():String	Gets the name of a parameter.
ParameterTRACE::getName():String	Gets the name of a parameter.
MethodCall::getReturnType():Sting	Gets the return type of a method call.
MethodMessge::getReturnType():Sting	Gets the return type of a method call message.
getStatement():String	Gets the statement of an executed statement.
getTrace():MethodCall	Gets the main_trace method call which is followed by other analyzed ExecutionStatement objects.
ParameterSD::getType():String	Gets the type of a parameter.
ParameterTRACE::getType():String	Gets the type of a parameter.
loadTrace(BufferedReader, Vector, Vector):void	Loads a trace file, a list of derived classes name of a target system and a list of names of classes to be analyzed.
loadTraceFile():void	The request to load a trace file into system.
transform():void	The request to transform the objects of the subclasses of ExecutionStatement into Message objects.
transformation(CompoundStatement):void	Transforms a method call or a condition statement into corresponding message.
transformMessage(CompoundStatement, int):void	Transforms the objects of the subclasses of ExecutionStatement into Message objects.

Appendix E Algorithms of RESDTool

E.1 Algorithm analyzeTrace

This algorithm analyzes a trace file which stores dynamic information collected at run time. According to characteristic of information of each log line, a set of `MethodCall`, `Return` and `ConditionStatement` objects are created. They are stored for further analysis. The context of a `MethodCall` object is also investigated.

During the analysis of trace file, the inheritance feature of C++ programming language also needs to be taken care. Due to the inheritance mechanism, all the public or protected member functions of a base class can be accessed by an object of the base class or its derived class. We use the id (physical address) of an object to recognize a method function is accessed whether by an object of the base class or by an object of the derived classes.

Input: **var** theTraceFile :File;

var derivedClassSet :Set of String;

var classToBeAnalyzed :Set of String;

Input/Output: **var** main_trace: `MethodCall`;

Algorithm

```
Procedure analyzeTrace(theTraceFile :File; derivedClassSet: Set of Strings;
classToBeAnalyzed: Set of String) main_trace : MethodCall
var  theObjectID, theClassName, theKind, theStatement :String;
        currentCompound :CompoundStatement;
        classSet :Set of ClassTRACE objects;
        instanceSet: Set of InstanceTRACE objects;
        anInstance :InstanceTRACE;
begin
     // to indicate that the beginning of all the interactions
     currentCompound := main_trace;
     foreach log line l1 in theTraceFile do
     begin
        theObjectID := first part of l1, which is present the id of callee object
        theClassName := second part of l1, in which the execution statement is defined
        theKind := third part of l1, the type of the execution
        theStatement := last part of l1, the statement of the execution
        if theKind = "Method Entry" then begin
              // identify the instance which the statement is executed on
              anInstance = findCallee(classSet, instanceSet, derivedClassSet, theObjectID,
                                      theClassName);
              // translate the information of the executed method call
              currentCompound := methodExecutionAnalysis(theStatement,anInstance,
                                                         currentCompound);
        end
        else if theKind = "Return" then begin
              // translate the information of the executed message return
              currentCompound := returnExecutionAnalysis(theStatement, currentCompound);
        end
        else if theKind = "While" or "If" or "Else" or "For" or "Do" or "Case"  then begin
              // translate the information of the executed control flow structure
              currentCompound := controlExecutionAnalysis(theKind, theStatement,
                                                          currentCompound);
        end
        else if theKind = "Method Exit then begin
              MethodCall m := a compoundStatement which is a corresponding MethodCall
                              object at upper levels;
              currentCompound := m.compoundStatement;
        end
        else // theType = "EndControl";
              currentComopound := endControlExecutionAnalysis(theStatement);
     end
     // refine the traces collection, make it neater especially for some unused
     // ConditionStatement object, correct the position of Return object if necessary
     Vector newSequence;
     foreach element e1 in main_trace.executionStatement do
     begin
          e1 = refineTraces(e1);
          if e1 is not null then
              newSequence->including(e1);
     end
     main_trace.executionStatement := newSequence;
     newSequence := selectTrace(main_trace, classToBeAnalyzed);
end
```

E.1.1 Internal Function findCallee

Description: This algorithm is designed to find a callee instance which a method is executed on.

Input: **var** cSet: a set of ClassTRACE objects;
var iSet: a set of InstanceTRACE objects;
var dSet: a set of string to hold the class names of derived classes of the target system;
var cName: a string to represent the class name field of a log line;
var objectID: a string to represent the objectID field of a log line;

Output: **var** theInstance: InstanceTRACE;

```
Function findCallee (cSet :Set of ClassTRACE; iSet :Set of InstanceTRACE; dSet :Set of
String; cName :String; objectID :String) theInstance :InstanceTRACE

var aClass :ClassTRACE;

begin
     if  iSet->exists(i: iSet | i.getID() = objectID then
     begin
          // identify the class of the instance
          if dSet->includes(cName) and i.classTRACE.getMyClass() != cName then
          begin
               if cSet->exists(c:cSet | c.getMyClass() = cName) then
                    aClass :=c;
               else begin
                    aClass := new ClassTRACE(cName);
                    cSet->including(aClass);
               end
               i.setMyClass(aClass);
          end
          theInstance := i;
     end
     else begin
          if cSet->exists(c:cSet | c.getMyClass() = cName) then
               aClass := c;
          else begin
               aClass = new ClassTRACE(cName);
               cSet->including(aClass);
          end
          theInstance := new InstanceTRACE(objectID, aClass);
          iSet->including(theInstance);
     end
end
```

E.1.2 Internal Function methodExecutionAnalysis

Description: This algorithm is to translate the dynamic information into a `MethodCall` object. The current compoundStatement is updated.

Input: var s: a string to represent the source statement field of a log line;
var i: InstanceTRACE, which a method call is executed on;

Input/Output: var c: CompoundStatement, which represents the current compoundStatement;

```
Function methodExecutionAnalysis (s :String; i :InstanceTRACE; c: CompoundStatement)
c:CompoundStatement

var mc :MethodCall;

begin
    mc := new MethodCall(s, i);
    // relationship with currentCompound
    c.executionStatement->including(mc);
    mc.compoundStatement := c;
    // reset currentCompound;
    c := mc;
end
```

E.1.3 Internal Function returnExecutionAnalysis

Description: This algorithm is to translate the dynamic information into a `Return` object. The current compoundStatement is updated.

Input: var s: a string to represent the source statement field of a log line;

Input/Output: var c: CompoundStatement, which represents the current compoundStatement;

```
Function returnExecutionAnalysis (s :String; c: CompoundStatement) c:CompoundStatement

var r :Return;

begin
    r := new Return(s);
    MethodCall m = the method call which makes the return execution;
    m.executionStatement->including(r);
    r.compoundStatement := m;
    // reset currentCompound;
    c := m.compoundStatement;
end
```

E.1.4 Internal Function controlExecutionAnalysis

Description: This algorithm is to translate the dynamic information into a `ControlStatement` object. The current compoundStatement is updated.

Input: **var** k: a string to represent the kind field of a log line;

var s: a string to represent the source statement field of a log line;

Input/Output: **var** c: CompoundStatement, which represents the current compoundStatement;

```
Function controlExecutionAnalysis (k: String; s :String; c: CompoundStatement)
c:CompoundStatement

var cond :ConditionClauseTRACE;
    cs :ConditionStatement;

begin
    ConditionClauseTRACE cond := new ConditionClauseTRACE(k, s);
    ConditionStatement cs := new ConditionStatement(k, s);
    c.executionStatement->including(cs);
    cs.compoundStatement := c;
    if currentCompound is a ConditionStatement object then
    begin
        copy every condition clause in currentCompound.conditionClauseTRACE to
        cs.conditionClauseTRACE set;
        cs.conditionClauseTRACE->including(cond);
    end
    // reset currentCompound
    c := cs;
end
```

E.1.5 Internal Function endControlExecutionAnalysis

Description: This algorithm is to process an end of control flow structure. The current compoundStatement is updated.

Input: **var** s: a string to represent the source statement field of a log line;

Input/Output: **var** c: CompoundStatement, which represents the current compoundStatement;

```
Function endControlExecutionAnalysis(s :String; c: CompoundStatement) c:CompoundStatement
begin
    if s = "breakLog" then
    begin
        go to upper compoundStatement levels until find a ConditionStatement whose
        kindOfCondition is "while" or "for" or "do" or "case";
        c := the compoundStatement of the found ConditionStatement;
    end
    else if s = "continueLog" then
    begin
        go to upper compoundStatement levels until find a ConditionStatement whose
        kindOfCondition is "while" or "for" or "do" ;
        c := the compoundStatement of found ConditionStatement;
    end
    else begin// regular the end of control
        c := c.compoundStatement;
   end
end
```

E.1.6 Internal Function refineTraces

Description: This algorithm is designed to refine the result of `analyzeTrace` algorithm. The result of `analyzeTrace` algorithm contains the trace information about method calls, return messages and control flow structures. Some of control flow structures, which do not affect the execution of methods and return messages, are recorded as well. Method calls are allowed to be executed in a return message. But according to our instrumentation tool, the return message is recorded before it is actually executed. In that case, a return message is recorded before the execution of method calls, which are stated inside this return statement, has been recorded. Therefore, the execution order is not correct in that way. We find a way to dispose useless control flow structures and solve the potential execution order problem during the return message execution.

Input/Output: var e:`ExecutionStatement`

```
Function refineTraces (e:ExecutionStatement)   e: ExecutionStatement
begin
    if e is a ConditionStatement then
    begin
        if e.executionStatement->size >0 then
        begin
            Vector newSequence;
            foreach element e1 in e.exeuctionStatement do
            begin
                 // refine the following traces recursively
                e1 := refineTraces(e1)
                if e1 is not null then
                    newSequence->including(e1);
            end
            e.executionStatement := newSequence;
        end
        else
            e := null;    // dispose the useless control flow structure
    end
    else if e is a MethodCall then
    begin
        if e.executionStatement->size >0 then
        begin
            Vector newSequence;
            foreach element e1 in e.executionStatement do
            begin
                if e1 is a Return then
                begin
                    study the return statement;
                    if there are some method calls stated in the return statement,
                    remember the numbers of the calls and names of the calls, keep them
                    in a collection cr;
                    if the names of n elements in the e.executionStatement collection,
                    which is kept after e1, are equal to the elements in the collection
                    cr with the existing order, refine those n elements by using
                    refineTraces algorithm one by one. If the result of refineTraces is
                    not null, move them into newSequence collection.
                    newSequence->including(e1);
                end
                else begin
                    e1 := refineTraces(e1);
                    if e1 is not null then
                        newSequence->including(e1);
                end
            end
        end
    end
end
```

E.2 Algorithm transformMessage

This algorithm transforms `MethodCall` and `Return` objects of `TRACE` package into `Message` objects of `SD` package, which represent the interaction behaviors of target system, according to the transforming rules which are defined ahead. If a group of methods are executed repeatedly under a loop condition, an `IterationMessage` is created. The iteration of a single method call is also presented. The set of generated `Message` objects compose the corresponding scenario diagram. The algorithm processes objects transformation recursively.

Input: **var** e:`CompoundStatement`

var iter:Integer;

Output: **var** messageSet:Set of `Messages`;

Algorithm

```
Founction transformMessage(e :CompoundStatement; iter :integer)  messageSet : Set of
Messages;
var  callerCT, calleeCT: ContextTRACE;
     callerCS, calleeCS: ContextSD;
begin
     foreach element e1 in the theTraces do
     begin
        if e1 is a MethodCall object then
        begin
              MethodMessage mm = methodTransformation(e1, iter);
              // process objects in e1.executionStatement set cursively
              if e1.executionStatement->size >0 then
                   mm.followedMessage := transformMessage(e1,0);
              // store the method message in the message set
                   messageSet->including(mm);
        end
        else if e1 is a Return object then
        begin
              ReturnMessage rm = returnTransformation(e1);
              // store the return message in the message set
              messageSet->including(rm);
        end
        else begin // e1 is a ConditionStatement object
              if e1.kindOfCondition is equal to "while", "do" or "for" then
              begin
                   Message m := loopControlTransformation(e1, iter);
                   // store the return message in the message set
                   messageSet->including(m);
              end
              else begin
                   Vector v := transformMessage(e1, iter);
                   foreach element v1 in v do
                   messageSet->including(v1);
               end
          end
     end
end
```

E.2.1 Internal Function methodTransformation

Description: This algorithm is designed to transform a `MethodCall` to a `MethodMessage`.

Input: var es:ExecutionStatement;

var i:Integer to indicate the iteration of this method message

Output: var m:MethodMessage

```
Function methodTransformation (es: ExecutionStatement; i:Integer) mm: MethodMessage
var callerCT, calleeCT :ContextTRACE;
    callerCS, calleeCS :ContextSD;
begin
       // identify the caller object of the method call message;
       callerCT := es.compoundStatement.getContext();
       if callerCT is an InstanceTRACE object then
              callerCS = new InstanceSD(callerCT.id, callerCT.classTRACE.name);
       else
              callerCS = new ClassSD(callerCT.name);
       // identify the callee object of the method call message;
       calleeCT := es.getContext();
       if calleeCT is an InstanceTRACE object then
              calleeCS = new InstanceSD(calleeCT.id, calleeCT.classTRACE.name);
       else
              calleeCS = new ClassSD(calleeCT.name);
       // create a method message
       m = new MethodMessage(callerCS, calleeCS, e1.statement, t.returnType, i);
       // formal parameter set of the method message
       foreach element p in the e1.parameterTRACE set do
       begin
              m.parameterSD->including(new ParameterSD(p.type, p.name));
       end
       // condition clause set of the method message
       foreach element c in the e1.conditionClauseTRACE set do
       begin
              m.conditionClauseSD->including(new ConditionClauseTRACE(c.clauseKind,
                                                  c.clauseStatement));
       end
end
```

E.2.2 Internal Function returnTransformation

Description: This algorithm is designed to transform a `Return` to a `ReturnMessage`.

Input: var es:ExecutionStatement;

Output: var rm:ReturnMessage

```
Function returnTransformation (es: ExecutionStatement) rm: ReturnMessage
var  callerCT, calleeCT :ContextTRACE;
     callerCS, calleeCS :ContextSD;
begin
        // identify the caller object of the method call message;
        callerCT := es.getContext();
        if callerCT is an InstanceTRACE object then
        begin
                callerCS = new InstanceSD(callerCT.id, callerCT.classTRACE.name);
        end
        else begin
                callerCS = new ClassSD(callerCT.name);
        end
        // identify the callee object of the method call message;
        calleeCT := es.compoundStatement.getContext();
        if calleeCT is an InstanceTRACE object then
        begin
                calleeCS = new InstanceSD(calleeCT.id, calleeCT.classTRACE.name);
        end
        else begin
                calleeCS = new ClassSD(calleeCT.name);
        end
        // create a return message
        rm := new ReturnMessage(callerCS, calleeCS, es.statement);
end
```

E.2.3 Internal Function loopConditionTransformation

Description: This algorithm is designed to analyze a ConditionStatement with loop condition, and investigate if it is a condition of a group of messages.

Input: var es:ExecutionStatement;
var i:Integer to indicate the iteration of es;

Output: var m:Message

```
Function loopConditionTransformation (es: ExecutionStatement, i: Integer) m: Message
var callerCT, calleeCT :ContextTRACE;
    callerCS, calleeCS :ContextSD;
    times:Integer;
begin
    seek if there are elements in the rest of es.compoundStatement.executionStatement set,
    which has same statement, kindOfCondition and conditionClauseTRACE set as es;
    times = the appearance of e1;
    if i >0 then
        times = i * times;
    if es.executionStatement->size =1 then
    begin //could be iteration of single method or iteration of a group of methods
        ExecutionStatement es1 = es.executionStatement->at(1);
        if es1 is a MethodCall then
        begin // iteration of a single method
              Vector v := transformMessage(es1, times);    // v->size = 1;
              m:= (MethodMessage)(v->at(1));
        end
        else
        begin // es1 is a ConditionStatement
              Vector v := transformMessage(es1, times);
              if v->size = 1 then // single method iteration
                   m:= (MethodMessage)(v->at(1));
              else begin // the iteration of a group of methods
                   callerCT := es.getContext();
                   if callerCT is an InstanceTRACE object then
                        callerCS = new InstanceSD(callerCT.id, callerCT.classTRACE.name);
                   else
                        callerCS = new ClassSD(callerCT.name);
                   // create an IterationMessage object
                   m = new IterationMessage(callerCS, es.statement, times);
                   m.message := v;
                   // callee objects
                   foreach element m1 in m.message do
                        m.calleeObject->including(m1.calleeObject);
                   // condition clause
                   foreach element c1 in es.getConditionClause() do
                   begin
                        ConditionClauseSD cct = new ConditionClauseSD(c1.clauseKind,
                                                                      c1.clauseStatement);
                        m.conditionClauseSD->including(cct);
                   end
              end
        end
    end
    else begin // iteration of a group of method messages
         callerCT := es.getContext();
         if callerCT is an InstanceTRACE object then
              callerCS = new InstanceSD(callerCT.id, callerCT.classTRACE.name);
         else
              callerCS = new ClassSD(callerCT.name);
         m := new IterationMessage(callerCS, es.statement, times)
         m.message := transformMessage(es, times);
         // callee objects
         foreach element m1 in m.message do
               m.calleeObject->including(m1.calleeObject);
         // condition clause
         foreach element c1 in es.getConditionClause() do
         begin
              ConditionClauseSD cct = new ConditionClauseSD (c1.clauseKind,
                                                             c1.clauseStatement);
              m.conditionClauseSD->including(cct);
         end
    end
end
```

Appendix F Class Diagram of ATM Banking System

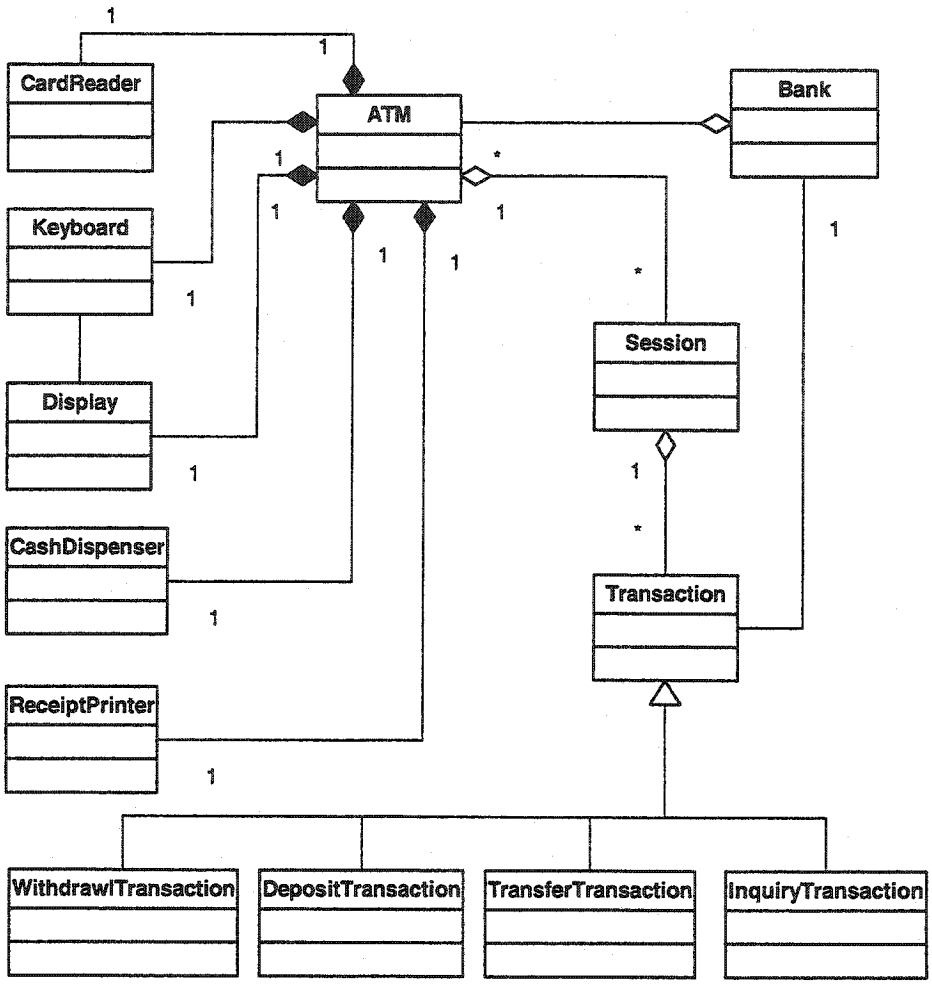


Figure 41 Class Diagram of ATM Banking System

Appendix G Use Case Diagram of ATM Banking System

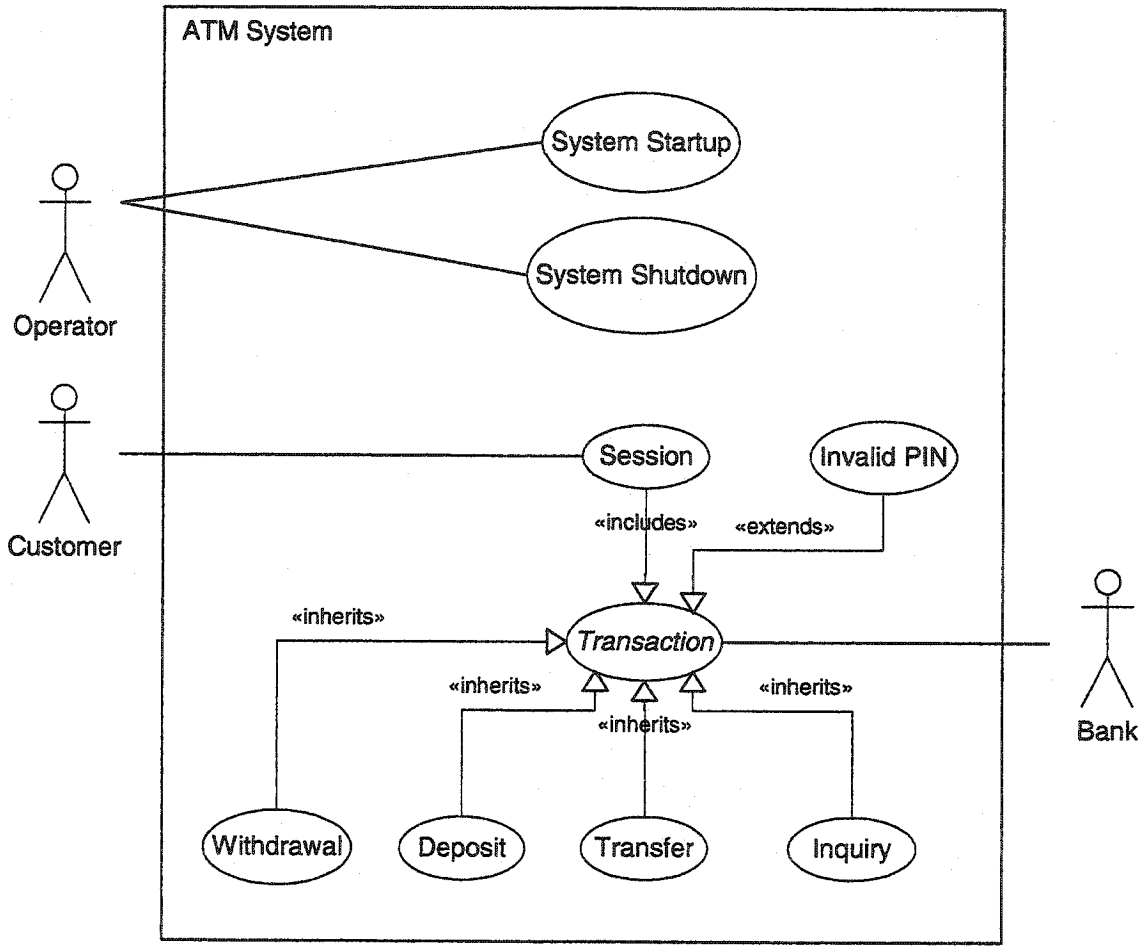


Figure 42 Use Case Diagram of ATM Banking System

Appendix H Instrumented session.cpp file of ATM Banking System

The statements in *Italic* font are trace-generating statements augmented by executing InstrTC++. To collect run time information in a file on the hard disk, a file attribute *OutFile* is augmented and defined in every method in the source file. A header file "<fstream.h>" is also included.

```
#include <fstream.h>
/*
 * Example ATM simulation - file session.cc
 *
 * This file implements the class that represents a single customer session
 * with the ATM, declared in session.h
 *
 * Copyright (c) 1996,1997 - Russell C. Bjork
 *
 */


#include "status.h"
#include "money.h"
#include "bank.h"
#include "session.h"
#include "transaction.h"
#include "atmparts.h"
#include "atm.h"
#include <iostream.h>


Session::Session(int cardNumber, ATM & atm, Bank & bank)
     :_cardNumber(cardNumber),
      _atm(atm),
      _bank(bank),
      _state(RUNNING),
      _PIN(0),
      _currentTransaction(0)
{ofstream OutFile;

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Method Entry\",\"Session(int
cardNumber,ATM& atm,Bank& bank)\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Method Exit\",\"Session(int
cardNumber,ATM& atm,Bank& bank)\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
}


void Session::doSessionUseCase(){
ofstream OutFile;

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
```

```
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Method Entry\",\"void
doSessionUseCase()\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

    _PIN = _atm.getPIN();

    do {
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Do\",\"_state == RUNNING\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

        const char * anotherMenu[] = { "Yes", "No" };
        _currentTransaction = Transaction::chooseTransaction(*this,_atm,_bank);
        Status::Code status = _currentTransaction -> doTransactionUseCase();

        switch (status){
            case  Status::SUCCESS :
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Case\",\"status ==
Status::SUCCESS\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

                if (1 != _atm.getMenuChoice("Do you want to perform another
                                                transaction?",2,anotherMenu)){
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"If\",\"1 != _atm.getMenuChoice (\"Do
you want to perform another transaction?\",2,anotherMenu)\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

                    _state = FINISHED;
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"EndControl\",\"a control flow
end\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
                }

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"EndControl\",\"breakLog\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
            break;

            case  Status::INVALID_PIN :
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Case\",\"status ==
Status::INVALID_PIN\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

                _state = ABORTED;

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"EndControl\",\"breakLog\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
            break;

            default :
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
```

```
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Case\",\"NOT(status== Status::SUCCESS
AND status== Status::INVALID_PIN)\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

                bool doAnother = doFailedTransactionExtension(status);
                if (! doAnother){
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"If\",\"! doAnother\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

                    _state = FINISHED;
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"EndControl\",\"a control flow
end\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
                }

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"EndControl\",\"a control flow
end\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
        }

        delete _currentTransaction;

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"EndControl\",\"a control flow
end\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
    }
    while (_state == RUNNING);

    if (_state != ABORTED){
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"If\",\"_state != ABORTED\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

        _atm.ejectCard();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"EndControl\",\"a control flow
end\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
    }

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Method Exit\",\"void
doSessionUseCase()\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
}


Status::Code Session::doInvalidPINExtension()
{ofstream OutFile;

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
```

```
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Method Entry\",\"Status::Code
doInvalidPINExtension()\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
    Status::Code code;
    for (int i = 0; i < 3; i ++){
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"For\",\"int i = 0; i < 3; i ++\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

        _PIN = _atm.reEnterPIN();
        code = _currentTransaction -> sendToBank();
        if (code != Status::INVALID_PIN){
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"If\",\"code !=
Status::INVALID_PIN\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Return\",\" code\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
            return  code;
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"EndControl\",\"a control flow
end\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
        }

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"EndControl\",\"a control flow
end\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
    }
    _atm.retainCard();
    _state = ABORTED;

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Return\",\" Status::INVALID_PIN\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
    return  Status::INVALID_PIN;

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Method Exit\",\"Status::Code
doInvalidPINExtension()\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
}


bool Session::doFailedTransactionExtension(Status::Code reason)
{ofstream OutFile;

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Method Entry\",\"bool
doFailedTransactionExtension(Status::Code reason)\"\n";
OutFile.close();
```

```
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

    switch (reason){
        case  Status::TOO_LITTLE_CASH :
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Case\",\"reason ==
Status::TOO_LITTLE_CASH\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Return\",\"
_atm.reportTransactionFailure( \"Sorry, there is not enough cash available to satisfy
your request\")\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
            return  _atm.reportTransactionFailure("Sorry, there is not enough cash
                                                  available to satisfy your request");

        case  Status::ENVELOPE_DEPOSIT_TIMED_OUT :
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Case\",\"reason ==
Status::ENVELOPE_DEPOSIT_TIMED_OUT\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Return\",\"
_atm.reportTransactionFailure( \"Envelope not deposited - transaction cancelled\")\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
            return  _atm.reportTransactionFailure("Envelope not deposited - transaction
                                                  cancelled");

        default :
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Case\",\"NOT(reason==
Status::TOO_LITTLE_CASH AND reason== Status::ENVELOPE_DEPOSIT_TIMED_OUT)\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Return\",\"
_atm.reportTransactionFailure( _bank.rejectionExplanation(reason))\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
            return  _atm.reportTransactionFailure(_bank.rejectionExplanation(reason));

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"EndControl\",\"a control flow
end\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
    }

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Method Exit\",\"bool
doFailedTransactionExtension(Status::Code reason)\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
}
```

```
int Session::cardNumber() const
{ofstream OutFile;

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Method Entry\",\"int cardNumber()
const\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Return\",\" _cardNumber\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
    return  _cardNumber;
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Method Exit\",\"int cardNumber()
const\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
}


int Session::PIN() const
{ofstream OutFile;

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Method Entry\",\"int PIN()
const\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Return\",\" _PIN\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
    return  _PIN;
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
OutFile.open("log.dat", ios::app);
OutFile << "\"" <<  this << "\"" << ",\"Session\",\"Method Exit\",\"int PIN() const\"\n";
OutFile.close();
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
}
```

Appendix I Trace File of Withdrawal Use Case

```
"0x0012FF74","Bank","Method Entry","Bank()"
"0x0012FF74","Bank","Method Exit","Bank()"
"0x00301740","CardReader","Method Entry","CardReader()"
"0x00301740","CardReader","Method Exit","CardReader()"
"0x00301710","Display","Method Entry","Display()"
"0x00301710","Display","Method Exit","Display()"
"0x003016E0","Keyboard","Method Entry","Keyboard()"
"0x003016E0","Keyboard","Method Exit","Keyboard()"
"0x003016B0","CashDispenser","Method Entry","CashDispenser()"
"0x003016B0","CashDispenser","Method Exit","CashDispenser()"
"0x00301680","ReceiptPrinter","Method Entry","ReceiptPrinter()"
"0x00301680","ReceiptPrinter","Method Exit","ReceiptPrinter()"
"0x0012FF50","ATM","Method Entry","ATM(int number,const char* location,Bank& bank)"
"0x0012FF50","ATM","Method Exit","ATM(int number,const char* location,Bank& bank)"
"0x0012FF50","ATM","Method Entry","void serviceCustomers()"
"0x0012FF50","ATM","While","_state == RUNNING"
"0x00301710","Display","Method Entry","void requestCard()"
"0x00301710","Display","Method Exit","void requestCard()"
"0x0012FF50","ATM","Do","_state == RUNNING && readerStatus == CardReader::NO_CARD"
"0x00301740","CardReader","Method Entry","CardReader::ReaderStatus
            checkForCardInserted()"
"0x00301740","CardReader","Return"," _status"
"0x0012FF50","ATM","EndControl","a control flow end"
"0x0012FF50","ATM","If","_state == RUNNING && readerStatus ==
            CardReader::CARD_HAS_BEEN_READ"
"0x00301740","CardReader","Method Entry","int cardNumber() const"
"0x00301740","CardReader","Return"," _cardNumberRead"
"0x00301160","Session","Method Entry","Session(int cardNumber,ATM& atm,Bank& bank)"
"0x00301160","Session","Method Exit","Session(int cardNumber,ATM& atm,Bank& bank)"
"0x00301160","Session","Method Entry","void doSessionUseCase()"
"0x0012FF50","ATM","Method Entry","int getPIN() const"
"0x00301710","Display","Method Entry","void requestPIN()"
"0x00301710","Display","Method Exit","void requestPIN()"
"0x003016E0","Keyboard","Method Entry","int readPIN()"
"0x003016E0","Keyboard","Return"," PIN"
"0x0012FF50","ATM","Return"," PIN"
"0x00301160","Session","Do","_state == RUNNING"
"0x0012FF50","ATM","Method Entry","int getMenuChoice(const char* whatToChoose,int
            numItems,const char* items[]) const"
"0x00301710","Display","Method Entry","void displayMenu(const char* whatToChoose,int
            numItems,const char* items[])"
"0x00301710","Display","For","int i=0; i< numItems; i++"
"0x00301710","Display","EndControl","a control flow end"
"0x00301710","Display","For","int i=0; i< numItems; i++"
"0x00301710","Display","EndControl","a control flow end"
"0x00301710","Display","For","int i=0; i< numItems; i++"
"0x00301710","Display","EndControl","a control flow end"
"0x00301710","Display","For","int i=0; i< numItems; i++"
"0x00301710","Display","EndControl","a control flow end"
"0x00301710","Display","Method Exit","void displayMenu(const char* whatToChoose,int
            numItems,const char* items[])"
"0x003016E0","Keyboard","Method Entry","int readMenuChoice(int numItems)"
"0x003016E0","Keyboard","Return"," choice"
"0x0012FF50","ATM","Return"," choice"
"0x003010A0","Transaction","Method Entry","Transaction(Session& session,ATM& atm,Bank&
            bank)"
"0x003010A0","Transaction","Method Exit","Transaction(Session& session,ATM& atm,Bank&
            bank)"
"0x003010A0","WithdrawlTransaction","Method Entry","WithdrawlTransaction(Session&
            session,ATM& atm,Bank& bank)"
"0x003010A0","WithdrawlTransaction","Method Exit","WithdrawlTransaction(Session&
            session,ATM& atm,Bank& bank)"
"0x003010A0","Transaction","Method Entry","Status::Code doTransactionUseCase()"
"0x003010A0","WithdrawlTransaction","Method Entry","Status::Code
            getTransactionSpecificsFromCustomer()"
"0x0012FF74","Bank","Method Entry","Bank::AccountType chooseAccountType(const char*
            purpose,ATM& atm)"
```

```
"0x0012FF50","ATM","Method Entry","int getMenuChoice(const char* whatToChoose,int
              numItems,const char* items[]) const"
"0x00301710","Display","Method Entry","void displayMenu(const char* whatToChoose,int
              numItems,const char* items[])"
"0x00301710","Display","For","int i=0; i< numItems; i++"
"0x00301710","Display","EndControl","a control flow end"
"0x00301710","Display","For","int i=0; i< numItems; i++"
"0x00301710","Display","EndControl","a control flow end"
"0x00301710","Display","For","int i=0; i< numItems; i++"
"0x00301710","Display","EndControl","a control flow end"
"0x00301710","Display","Method Exit","void displayMenu(const char* whatToChoose,int
              numItems,const char* items[])"
"0x003016E0","Keyboard","Method Entry","int readMenuChoice(int numItems)"
"0x003016E0","Keyboard","Return"," choice"
"0x0012FF50","ATM","Return"," choice"
"0x0012FF74","Bank","Return"," AccountType(choice - 1)"
"0x0012FF50","ATM","Method Entry","int getMenuChoice(const char* whatToChoose,int
              numItems,const char* items[]) const"
"0x00301710","Display","Method Entry","void displayMenu(const char* whatToChoose,int
              numItems,const char* items[])"
"0x00301710","Display","For","int i=0; i< numItems; i++"
"0x00301710","Display","EndControl","a control flow end"
"0x00301710","Display","For","int i=0; i< numItems; i++"
"0x00301710","Display","EndControl","a control flow end"
"0x00301710","Display","For","int i=0; i< numItems; i++"
"0x00301710","Display","EndControl","a control flow end"
"0x00301710","Display","For","int i=0; i< numItems; i++"
"0x00301710","Display","EndControl","a control flow end"
"0x00301710","Display","For","int i=0; i< numItems; i++"
"0x00301710","Display","EndControl","a control flow end"
"0x00301710","Display","For","int i=0; i< numItems; i++"
"0x00301710","Display","EndControl","a control flow end"
"0x00301710","Display","Method Exit","void displayMenu(const char* whatToChoose,int
              numItems,const char* items[])"
"0x003016E0","Keyboard","Method Entry","int readMenuChoice(int numItems)"
"0x003016E0","Keyboard","Return"," choice"
"0x0012FF50","ATM","Return"," choice"
"0x003010A0","WithdrawlTransaction","Case","_atm.getMenuChoice("Please choose an
              amount:", 7, menu) == 2"
"0x003010A0","WithdrawlTransaction","EndControl","breakLog"
"0x0012FF50","ATM","Method Entry","bool checkIfCashAvailable(Money amount) const"
"0x0012FF50","ATM","Return"," ! (_cashDispenser.currentCash() < amount)"
"0x003016B0","CashDispenser","Method Entry","Money currentCash() const"
"0x003016B0","CashDispenser","Return"," _currentCash"
"0x003010A0","WithdrawlTransaction","If","_atm.checkIfCashAvailable(_amount)"
"0x003010A0","WithdrawlTransaction","Return"," Status::SUCCESS"
"0x003010A0","WithdrawlTransaction","Method Entry","Status::Code sendToBank()"
"0x003010A0","WithdrawlTransaction","Return","
              _bank.initiateWithdrawl(_session.cardNumber(), _session.PIN(),
              _atm.number(), _serialNumber, _fromAccount, _amount, _newBalance,
              _availableBalance)"
"0x0012FF50","ATM","Method Entry","int number() const"
"0x0012FF50","ATM","Return"," _number"
"0x00301160","Session","Method Entry","int PIN() const"
"0x00301160","Session","Return"," _PIN"
"0x00301160","Session","Method Entry","int cardNumber() const"
"0x00301160","Session","Return"," _cardNumber"
"0x0012FF74","Bank","Method Entry","Status::Code initiateWithdrawl(int cardNumber,int
              PIN,int ATMnumber,int serialNumber,AccountType from,Money amount,Money&
              newBalance,Money& availableBalance)"
"0x0012FF74","Bank","Return"," Status::SUCCESS"
"0x003010A0","Transaction","If","code == Status::SUCCESS"
"0x003010A0","WithdrawlTransaction","Method Entry","Status::Code
              finishApprovedTransaction()"
"0x0012FF50","ATM","Method Entry","void dispenseCash(Money amount) const"
"0x003016B0","CashDispenser","Method Entry","void dispenseCash(Money amount)"
"0x003016B0","CashDispenser","Method Exit","void dispenseCash(Money amount)"
"0x0012FF50","ATM","Method Exit","void dispenseCash(Money amount) const"
"0x0012FF50","ATM","Method Entry","int number() const"
```

```
"0x0012FF50","ATM","Return"," _number"
"0x0012FF74","Bank","Method Entry","void finishWithdrawl(int ATMnumber,int
            serialNumber,bool succeeded)"
"0x0012FF74","Bank","If","succeeded"
"0x0012FF74","Bank","EndControl","a control flow end"
"0x0012FF74","Bank","Method Exit","void finishWithdrawl(int ATMnumber,int
            serialNumber,bool succeeded)"
"0x0012FF74","Bank","Method Entry","const char* accountName(AccountType type) const"
"0x0012FF74","Bank","Return"," accountNames[type]"
"0x00301160","Session","Method Entry","int cardNumber() const"
"0x00301160","Session","Return"," _cardNumber"
"0x0012FF50","ATM","Method Entry","void issueReceipt(int cardNumber,int
            serialNumber,const char* description,Money amount,Money balance,Money
            availableBalance) const"
"0x00301680","ReceiptPrinter","Method Entry","void printReceipt(int theATMnumber,const
            char* theATMlocation,int cardNumber,int serialNumber,const char*
            description,Money amount,Money balance,Money availableBalance)"
"0x00301680","ReceiptPrinter","Else","NOT(amount == Money(0))"
"0x00301680","ReceiptPrinter","EndControl","a control flow end"
"0x00301680","ReceiptPrinter","For","int i=0; i<=6; i++"
"0x00301680","ReceiptPrinter","EndControl","a control flow end"
"0x00301680","ReceiptPrinter","For","int i=0; i<=6; i++"
"0x00301680","ReceiptPrinter","EndControl","a control flow end"
"0x00301680","ReceiptPrinter","For","int i=0; i<=6; i++"
"0x00301680","ReceiptPrinter","EndControl","a control flow end"
"0x00301680","ReceiptPrinter","For","int i=0; i<=6; i++"
"0x00301680","ReceiptPrinter","EndControl","a control flow end"
"0x00301680","ReceiptPrinter","For","int i=0; i<=6; i++"
"0x00301680","ReceiptPrinter","EndControl","a control flow end"
"0x00301680","ReceiptPrinter","For","int i=0; i<=6; i++"
"0x00301680","ReceiptPrinter","EndControl","a control flow end"
"0x00301680","ReceiptPrinter","For","int i=0; i<=6; i++"
"0x00301680","ReceiptPrinter","EndControl","a control flow end"
"0x00301680","ReceiptPrinter","Method Exit","void printReceipt(int theATMnumber,const
            char* theATMlocation,int cardNumber,int serialNumber,const char*
            description,Money amount,Money balance,Money availableBalance)"
"0x0012FF50","ATM","Method Exit","void issueReceipt(int cardNumber,int serialNumber,const
            char* description,Money amount,Money balance,Money availableBalance) const"
"0x003010A0","WithdrawlTransaction","Return"," Status::SUCCESS"
"0x003010A0","Transaction","EndControl","a control flow end"
"0x003010A0","Transaction","Return"," code"
"0x00301160","Session","Case","status == Status::SUCCESS"
"0x0012FF50","ATM","Method Entry","int getMenuChoice(const char* whatToChoose,int
            numItems,const char* items[]) const"
"0x00301710","Display","Method Entry","void displayMenu(const char* whatToChoose,int
            numItems,const char* items[])"
"0x00301710","Display","For","int i=0; i< numItems; i++"
"0x00301710","Display","EndControl","a control flow end"
"0x00301710","Display","For","int i=0; i< numItems; i++"
"0x00301710","Display","EndControl","a control flow end"
"0x00301710","Display","Method Exit","void displayMenu(const char* whatToChoose,int
            numItems,const char* items[])"
"0x003016E0","Keyboard","Method Entry","int readMenuChoice(int numItems)"
"0x003016E0","Keyboard","Return"," choice"
"0x0012FF50","ATM","Return"," choice"
"0x00301160","Session","If","1 != _atm.getMenuChoice ("Do you want to perform another
            transaction?",2,anotherMenu)"
"0x00301160","Session","EndControl","a control flow end"
"0x00301160","Session","EndControl","breakLog"
"0x00301160","Session","EndControl","a control flow end"
"0x00301160","Session","If","_state != ABORTED"
"0x0012FF50","ATM","Method Entry","void ejectCard() const"
"0x00301740","CardReader","Method Entry","void ejectCard()"
"0x00301740","CardReader","Method Exit","void ejectCard()"
"0x0012FF50","ATM","Method Exit","void ejectCard() const"
"0x00301160","Session","EndControl","a control flow end"
"0x00301160","Session","Method Exit","void doSessionUseCase()"
"0x0012FF50","ATM","EndControl","a control flow end"
"0x0012FF50","ATM","EndControl","a control flow end"
"0x0012FF50","ATM","Method Exit","void serviceCustomers()"
```

Appendix J Scenario Diagram Result File for Withdrawal Use Case

```
1.  Method message: Bank
on Object 0x0012FF74 of class Bank from Class Actor;


2.  Method message: CardReader
on Object 0x00301740 of class CardReader from Class Actor;


3.  Method message: Display
on Object 0x00301710 of class Display from Class Actor;


4.  Method message: Keyboard
on Object 0x003016E0 of class Keyboard from Class Actor;


5.  Method message: CashDispenser
on Object 0x003016B0 of class CashDispenser from Class Actor;


6.  Method message: ReceiptPrinter
on Object 0x00301650 of class ReceiptPrinter from Class Actor;

7.  Method message: EnvelopeAcceptor
on Object 0x00301680 of class EnvelopeAcceptor from Class Actor;

8.  Method message: ATM
on Object 0x0012FF50 of class ATM from Class Actor;
Parameter Set: ["number" of int type, "location" of const char* type, "bank" of Bank& type]


9.  Method message: serviceCustomers
on Object 0x0012FF50 of class ATM from Class Actor;
Return type: void;

     Repetition with condition _state == RUNNING begins

          9.1.  Method message: requestCard
          on Object 0x00301710 of class Display from Object 0x0012FF50 of class ATM;
          Return type: void;
          under condition clauses: [_state == RUNNING(While)]
          repeated 1 times
```

```
9.2. Method message: checkForCardInserted
on Object 0x00301740 of class CardReader from Object 0x0012FF50 of class ATM;
Return type: CardReader::ReaderStatus;
under condition clauses: [_state == RUNNING(While), _state == RUNNING && readerStatus == CardReader::NO_CARD(Do)]
repeated 1 times

9.2.1. Return message: _status
from Object 0x00301740 of class CardReader to Object 0x0012FF50 of class ATM

9.3. Method message: cardNumber
on Object 0x00301740 of class CardReader from Object 0x0012FF50 of class ATM;
Return type: int;
under condition clauses: [_state == RUNNING(While), _state == RUNNING && readerStatus ==
CardReader::CARD_HAS_BEEN_READ(If)]
repeated 1 times

9.3.1. Return message: _cardNumberRead
from Object 0x00301740 of class CardReader to Object 0x0012FF50 of class ATM

9.4. Method message: Session
on Object 0x00301160 of class Session from Object 0x0012FF50 of class ATM;
Parameter Set: ["cardNumber" of int type, "atm" of ATM& type, "bank" of Bank& type]
under condition clauses: [_state == RUNNING(While), _state == RUNNING && readerStatus ==
CardReader::CARD_HAS_BEEN_READ(If)]
repeated 1 times

9.5. Method message: doSessionUseCase
on Object 0x00301160 of class Session from Object 0x0012FF50 of class ATM;
Return type: void;
under condition clauses: [_state == RUNNING(While), _state == RUNNING && readerStatus ==
CardReader::CARD_HAS_BEEN_READ(If)]
repeated 1 times

    9.5.1.  Method message: getPIN
    on Object 0x0012FF50 of class ATM from Object 0x00301160 of class Session;
    Return type: int;

        9.5.1.1.  Method message: requestPIN
        on Object 0x00301710 of class Display from Object 0x0012FF50 of class ATM;
        Return type: void;

        9.5.1.2.  Method message: readPIN
        on Object 0x003016E0 of class Keyboard from Object 0x0012FF50 of class ATM;
        Return type: int;

        9.5.1.2.1. Return message: PIN
        from Object 0x003016E0 of class Keyboard to Object 0x0012FF50 of class ATM
```

```
9.5.1.3. Return message: PIN
from Object 0x0012FF50 of class ATM to Object 0x00301160 of class Session

Repetition with condition _state == RUNNING begins

        9.5.2.      Method message: getMenuChoice
        on Object 0x0012FF50 of class ATM from Object 0x00301160 of class Session;
        Return type: int;
        Parameter Set: ["whatToChoose" of const char* type, "numItems" of int type, "items[]" of const char* type]
        under condition clauses: [_state == RUNNING(Do)]
        repeated 1 times

             9.5.2.1.    Method message: displayMenu
             on Object 0x00301710 of class Display from Object 0x0012FF50 of class ATM;
             Return type: void;
             Parameter Set: ["whatToChoose" of const char* type, "numItems" of int type, "items[]" of const char*
             type]

             9.5.2.2.    Method message: readMenuChoice
             on Object 0x003016E0 of class Keyboard from Object 0x0012FF50 of class ATM;
             Return type: int;
             Parameter Set: ["numItems" of int type]

             9.5.2.2.1. Return message: choice
             from Object 0x003016E0 of class Keyboard to Object 0x0012FF50 of class ATM

        9.5.2.3.    Return message: choice
        from Object 0x0012FF50 of class ATM to Object 0x00301160 of class Session

        9.5.3.      Method message: Transaction
        on Object 0x003010A0 of class WithdrawlTransaction from Object 0x00301160 of class Session;
        Parameter Set: ["session" of Session& type, "atm" of ATM& type, "bank" of Bank& type]
        under condition clauses: [_state == RUNNING(Do)]
        repeated 1 times

        9.5.4.      Method message: WithdrawlTransaction
        on Object 0x003010A0 of class WithdrawlTransaction from Object 0x00301160 of class Session;
        Parameter Set: ["session" of Session& type, "atm" of ATM& type, "bank" of Bank& type]
        under condition clauses: [_state == RUNNING(Do)]
        repeated 1 times

        9.5.5.      Method message: doTransactionUseCase
        on Object 0x003010A0 of class WithdrawlTransaction from Object 0x00301160 of class Session;
        Return type: Status::Code;
        under condition clauses: [_state == RUNNING(Do)]
        repeated 1 times
```

```
9.5.5.1.   Method message: getTransactionSpecificsFromCustomer
on Object 0x003010A0 of class WithdrawlTransaction from Object 0x003010A0 of class WithdrawlTransaction;
Return type: Status::Code;

    9.5.5.1.1.   Method message: chooseAccountType
    on Object 0x0012FF74 of class Bank from Object 0x003010A0 of class WithdrawlTransaction;
    Return type: Bank::AccountType;
    Parameter Set: ["purpose" of const char* type, "atm" of ATM& type]

        9.5.5.1.1.1. Method message: getMenuChoice
        on Object 0x0012FF50 of class ATM from Object 0x0012FF74 of class Bank;
        Return type: int;
        Parameter Set: ["whatToChoose" of const char* type, "numItems" of int type, "items[]" of
        const char* type]

            9.5.5.1.1.1.1.   Method message: displayMenu
            on Object 0x00301710 of class Display from Object 0x0012FF50 of class ATM;
            Return type: void;
            Parameter Set: ["whatToChoose" of const char* type, "numItems" of int type, "items[]" of
            const char* type]

            9.5.5.1.1.1.2.   Method message: readMenuChoice
            on Object 0x003016E0 of class Keyboard from Object 0x0012FF50 of class ATM;
            Return type: int;
            Parameter Set: ["numItems" of int type]

            9.5.5.1.1.1.2.1. Return message: choice
            from Object 0x003016E0 of class Keyboard to Object 0x0012FF50 of class ATM

        9.5.5.1.1.1.3.   Return message: choice
        from Object 0x0012FF50 of class ATM to Object 0x0012FF74 of class Bank

    9.5.5.1.1.2. Return message: AccountType(choice - 1)
    from Object 0x0012FF74 of class Bank to Object 0x003010A0 of class WithdrawlTransaction

    9.5.5.1.2.   Method message: getMenuChoice
    on Object 0x0012FF50 of class ATM from Object 0x003010A0 of class WithdrawlTransaction;
    Return type: int;
    Parameter Set: ["whatToChoose" of const char* type, "numItems" of int type, "items[]" of const
    char* type]

        9.5.5.1.2.1.   Method message: displayMenu
        on Object 0x00301710 of class Display from Object 0x0012FF50 of class ATM;
        Return type: void;
        Parameter Set: ["whatToChoose" of const char* type, "numItems" of int type, "items[]" of
        const char* type]
```

```
            9.5.5.1.2.2.   Method message: readMenuChoice
            on Object 0x003016E0 of class Keyboard from Object 0x0012FF50 of class ATM;
            Return type: int;
            Parameter Set: ["numItems" of int type]

            9.5.5.1.2.2.1. Return message: choice
            from Object 0x003016E0 of class Keyboard to Object 0x0012FF50 of class ATM

      9.5.5.1.2.3. Return message: choice
      from Object 0x0012FF50 of class ATM to Object 0x003010A0 of class WithdrawlTransaction

      9.5.5.1.3.   Method message: checkIfCashAvailable
      on Object 0x0012FF50 of class ATM from Object 0x003010A0 of class WithdrawlTransaction;
      Return type: bool;
      Parameter Set: ["amount" of Money type]

            9.5.5.1.3.1.   Method message: currentCash
            on Object 0x003016B0 of class CashDispenser from Object 0x0012FF50 of class ATM;
            Return type: Money;

            9.5.5.1.3.1.1. Return message: _currentCash
            from Object 0x003016B0 of class CashDispenser to Object 0x0012FF50 of class ATM

      9.5.5.1.3.2. Return message: ! (_cashDispenser.currentCash() < amount)
      from Object 0x0012FF50 of class ATM to Object 0x003010A0 of class WithdrawlTransaction

9.5.5.1.4. Return message: Status::SUCCESS
from Object 0x003010A0 of class WithdrawlTransaction to Object 0x003010A0 of class WithdrawlTransaction

9.5.5.2.   Method message: sendToBank
on Object 0x003010A0 of class WithdrawlTransaction from Object 0x003010A0 of class WithdrawlTransaction;
Return type: Status::Code;

      9.5.5.2.1.   Method message: number
      on Object 0x0012FF50 of class ATM from Object 0x003010A0 of class WithdrawlTransaction;
      Return type: int;

      9.5.5.2.1.1. Return message: _number
      from Object 0x0012FF50 of class ATM to Object 0x003010A0 of class WithdrawlTransaction

      9.5.5.2.2.   Method message: PIN
      on Object 0x00301160 of class Session from Object 0x003010A0 of class WithdrawlTransaction;
      Return type: int;

      9.5.5.2.2.1. Return message: _PIN
      from Object 0x00301160 of class Session to Object 0x003010A0 of class WithdrawlTransaction
```

9.5.5.2.3. Method message: cardNumber
on Object 0x00301160 of class Session from Object 0x003010A0 of class WithdrawlTransaction;
Return type: int;

9.5.5.2.3.1. Return message: _cardNumber
from Object 0x00301160 of class Session to Object 0x003010A0 of class WithdrawlTransaction

9.5.5.2.4. Method message: initiateWithdrawl
on Object 0x0012FF74 of class Bank from Object 0x003010A0 of class WithdrawlTransaction;
Return type: Status::Code;
Parameter Set: ["cardNumber" of int type, "PIN" of int type, "ATMnumber" of int type, "serialNumber" of int type, "from" of AccountType type, "amount" of Money type, "newBalance" of Money& type, "availableBalance" of Money& type]

9.5.5.2.4.1. Return message: Status::SUCCESS
from Object 0x0012FF74 of class Bank to Object 0x003010A0 of class WithdrawlTransaction

9.5.5.2.5. Return message: _bank.initiateWithdrawl(_session.cardNumber(), _session.PIN(), _atm.number(), _serialNumber, _fromAccount, _amount, _newBalance, _availableBalance)
from Object 0x003010A0 of class WithdrawlTransaction to Object 0x003010A0 of class WithdrawlTransaction

9.5.5.3. Method message: finishApprovedTransaction
on Object 0x003010A0 of class WithdrawlTransaction from Object 0x003010A0 of class WithdrawlTransaction;
Return type: Status::Code;
under condition clauses: [code == Status::SUCCESS(If)]

9.5.5.3.1. Method message: dispenseCash
on Object 0x0012FF50 of class ATM from Object 0x003010A0 of class WithdrawlTransaction;
Return type: void;
Parameter Set: ["amount" of Money type]

9.5.5.3.1.1. Method message: dispenseCash
on Object 0x003016B0 of class CashDispenser from Object 0x0012FF50 of class ATM;
Return type: void;
Parameter Set: ["amount" of Money type]

9.5.5.3.2. Method message: number
on Object 0x0012FF50 of class ATM from Object 0x003010A0 of class WithdrawlTransaction;
Return type: int;

9.5.5.3.2.1. Return message: _number
from Object 0x0012FF50 of class ATM to Object 0x003010A0 of class WithdrawlTransaction

9.5.5.3.3. Method message: finishWithdrawl
on Object 0x0012FF74 of class Bank from Object 0x003010A0 of class WithdrawlTransaction;
Return type: void;

```
Parameter Set: ["ATMnumber" of int type, "serialNumber" of int type, "succeeded" of bool type]

9.5.5.3.4.   Method message: accountName
on Object 0x0012FF74 of class Bank from Object 0x003010A0 of class WithdrawlTransaction;
Return type: const char*;
Parameter Set: ["type" of AccountType type]

9.5.5.3.4.1. Return message: accountNames[type]
from Object 0x0012FF74 of class Bank to Object 0x003010A0 of class WithdrawlTransaction

9.5.5.3.5.   Method message: cardNumber
on Object 0x00301160 of class Session from Object 0x003010A0 of class WithdrawlTransaction;
Return type: int;

9.5.5.3.5.1. Return message: _cardNumber
from Object 0x00301160 of class Session to Object 0x003010A0 of class WithdrawlTransaction

9.5.5.3.6.   Method message: issueReceipt
on Object 0x0012FF50 of class ATM from Object 0x003010A0 of class WithdrawlTransaction;
Return type: void;
Parameter Set: ["cardNumber" of int type, "serialNumber" of int type, "description" of const char*
type, "amount" of Money type, "balance" of Money type, "availableBalance" of Money type]

     9.5.5.3.6.1.   Method message: printReceipt
     on Object 0x00301680 of class ReceiptPrinter from Object 0x0012FF50 of class ATM;
     Return type: void;
     Parameter Set: ["theATMnumber" of int type, "theATMlocation" of const char* type,
     "cardNumber" of int type, "serialNumber" of int type, "description" of const char* type,
     "amount" of Money type, "balance" of Money type, "availableBalance" of Money type]

9.5.5.3.7. Return message: Status::SUCCESS
from Object 0x003010A0 of class WithdrawlTransaction to Object 0x003010A0 of class WithdrawlTransaction

9.5.5.4. Return message: code
from Object 0x003010A0 of class WithdrawlTransaction to Object 0x00301160 of class Session

9.5.6.   Method message: getMenuChoice
on Object 0x0012FF50 of class ATM from Object 0x00301160 of class Session;
Return type: int;
Parameter Set: ["whatToChoose" of const char* type, "numItems" of int type, "items[]" of const char* type]
under condition clauses: [_state == RUNNING(Do), status == Status::SUCCESS(Case)]
repeated 1 times

9.5.6.1.   Method message: displayMenu
on Object 0x00301710 of class Display from Object 0x0012FF50 of class ATM;
Return type: void;
```

```
                    Parameter Set: ["whatToChoose" of const char* type, "numItems" of int type, "items[]" of const char*
                    type]

                    9.5.6.2.   Method message: readMenuChoice
                    on Object 0x003016E0 of class Keyboard from Object 0x0012FF50 of class ATM;
                    Return type: int;
                    Parameter Set: ["numItems" of int type]

                    9.5.6.2.1. Return message: choice
                    from Object 0x003016E0 of class Keyboard to Object 0x0012FF50 of class ATM

            9.5.6.3. Return message: choice
            from Object 0x0012FF50 of class ATM to Object 0x00301160 of class Session

        Repetition with condition _state == RUNNING ends

        9.5.7.   Method message: ejectCard
        on Object 0x0012FF50 of class ATM from Object 0x00301160 of class Session;
        Return type: void;
        under condition clauses: [_state != ABORTED(If)]

            9.5.7.1. Method message: ejectCard
            on Object 0x00301740 of class CardReader from Object 0x0012FF50 of class ATM;
            Return type: void;

Repetition with condition _state == RUNNING ends
```

Appendix K Sequence Diagram of ATM System

K.1 Sequence Diagram of Session Use Case

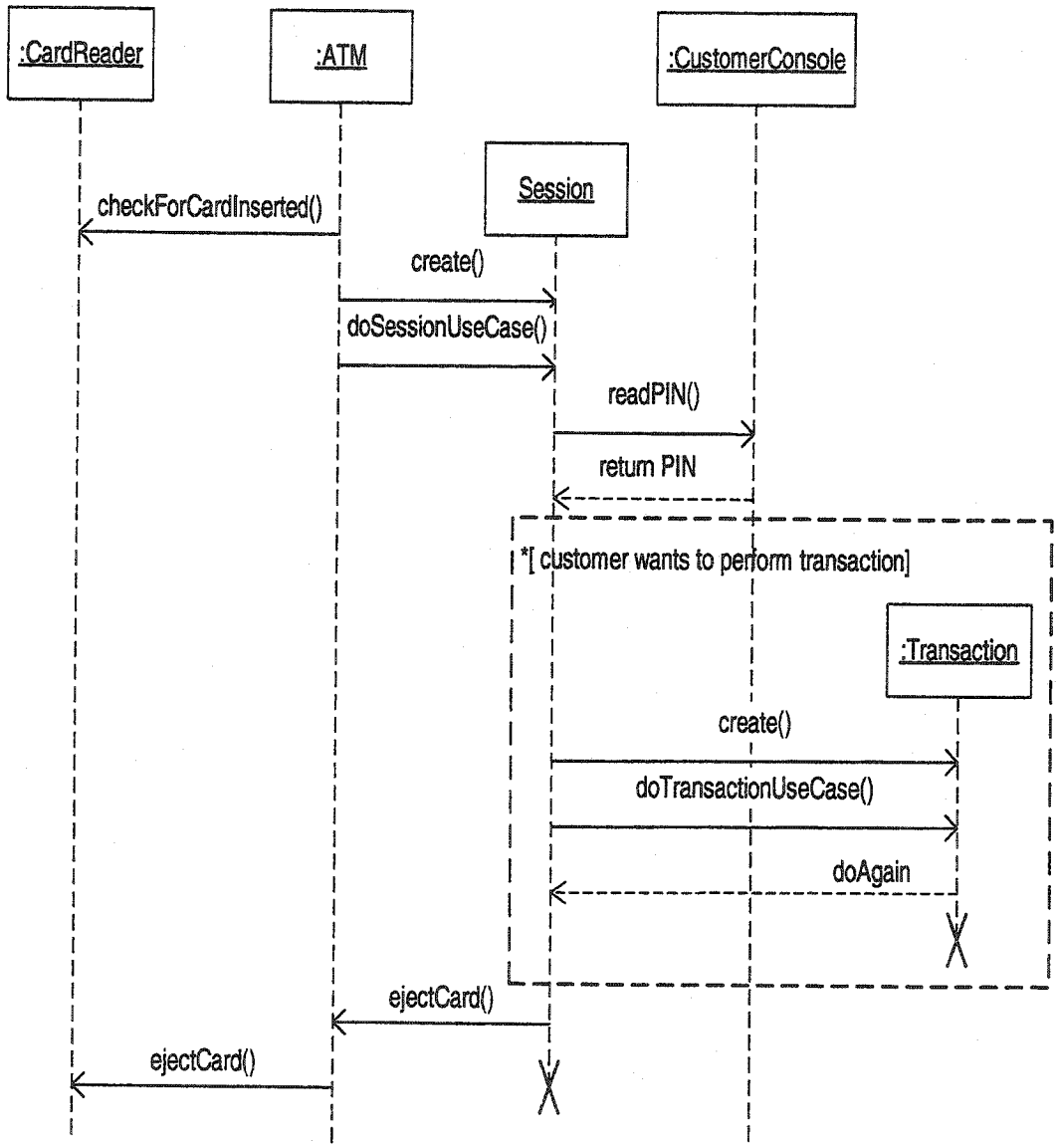


Figure 43 Sequence Diagram of Session Use Case

K.2 Sequence Diagram of Deposit Use Case

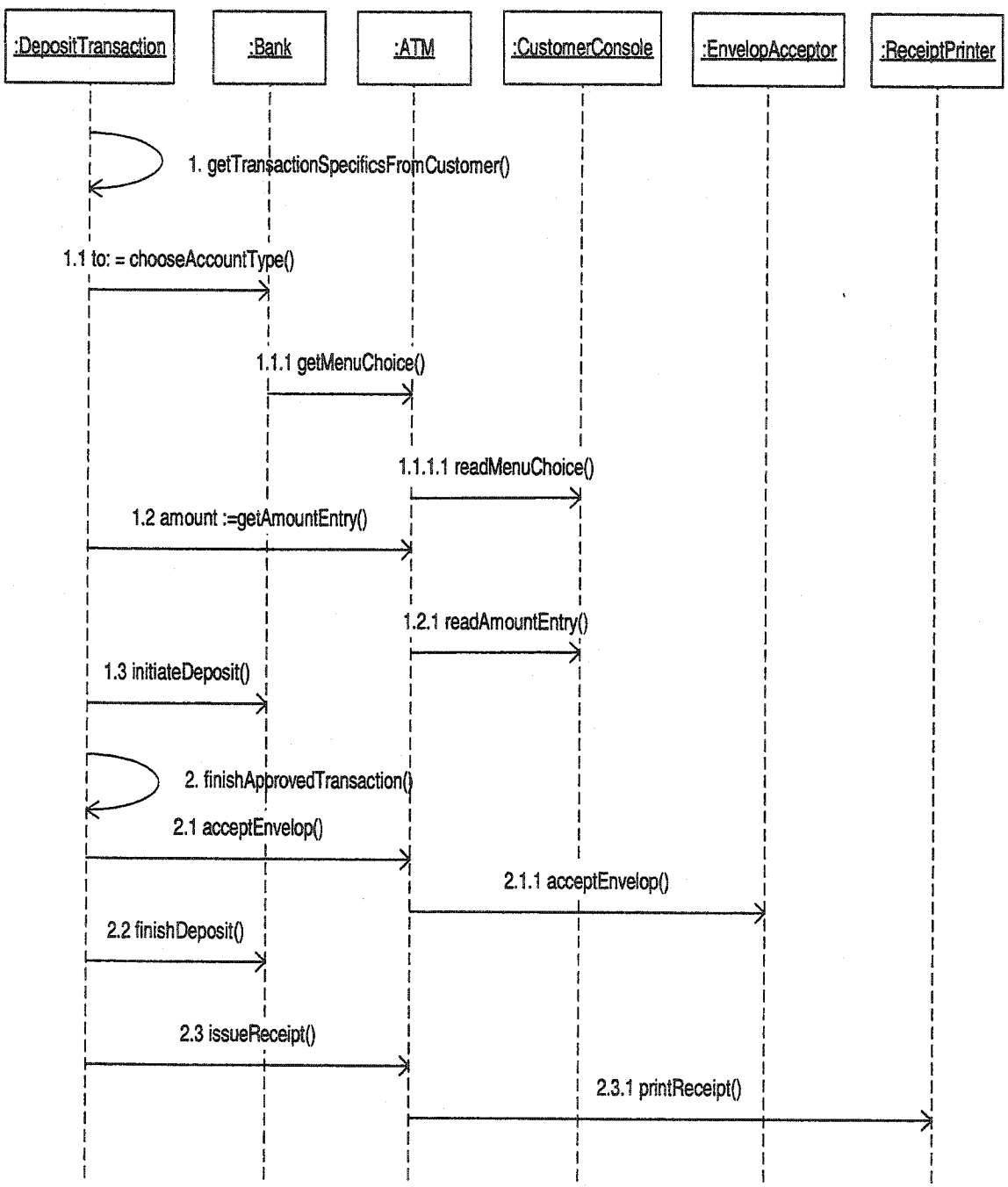


Figure 44 Sequence Diagram of Deposit Use Case

K.3 Sequence Diagram of Transfer Use Case

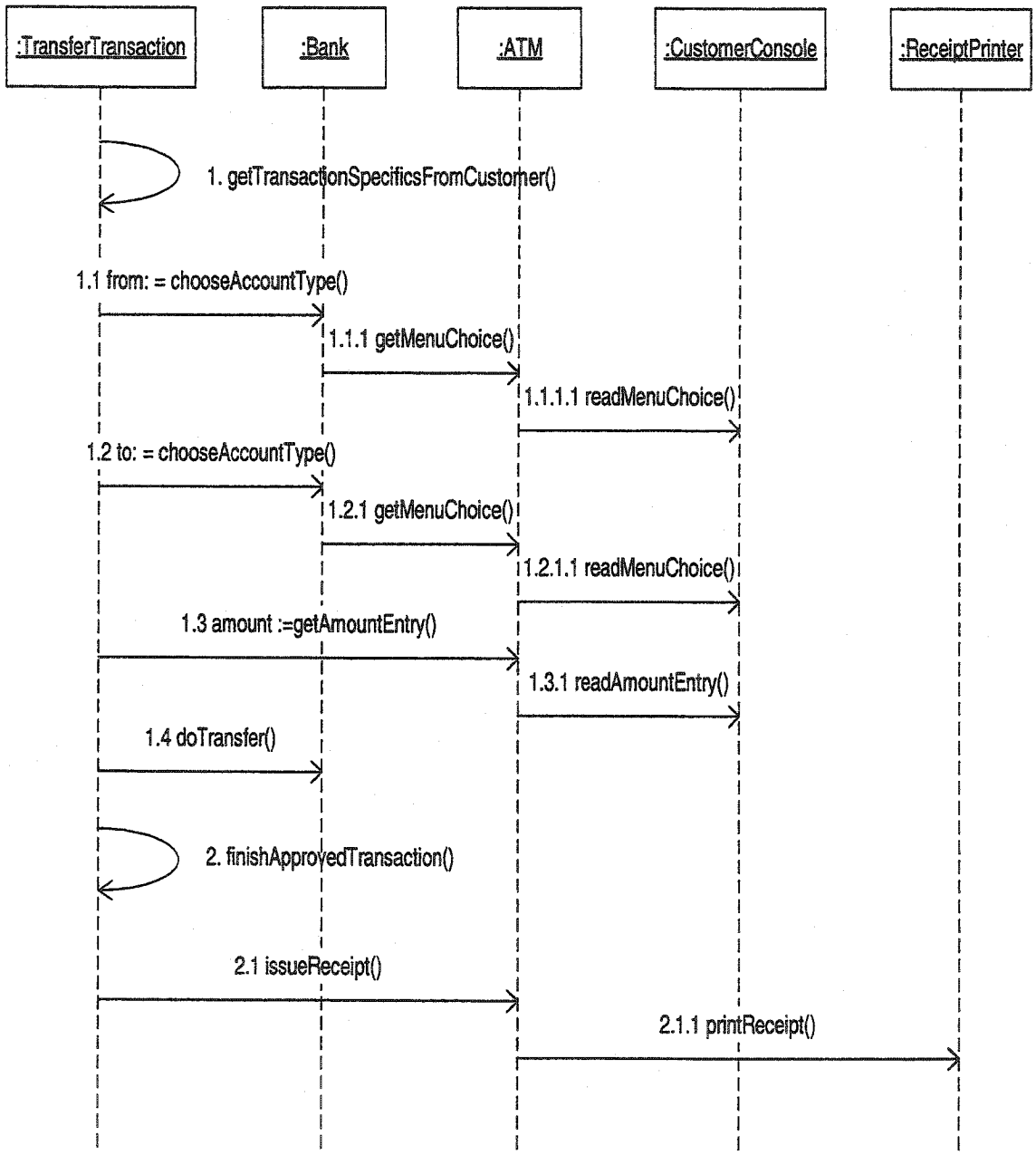


Figure 45 Sequence Diagram of Transfer Use Case

K.4 Sequence Diagram of Inquiry Use Case

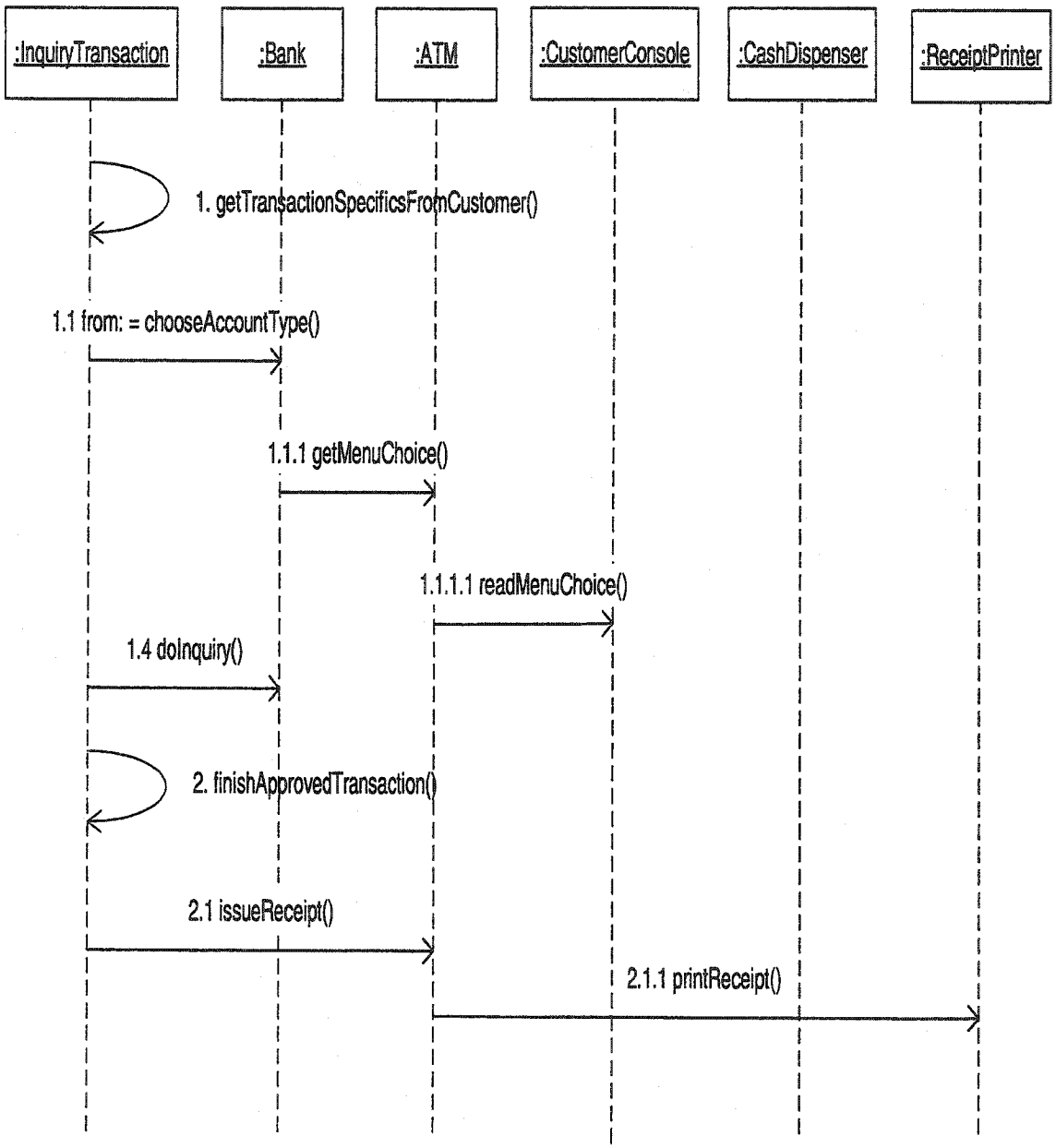


Figure 46 Sequence Diagram of Inquiry Use Case

K.5 Sequence Diagram of Unsuccessful Withdrawal Use Case

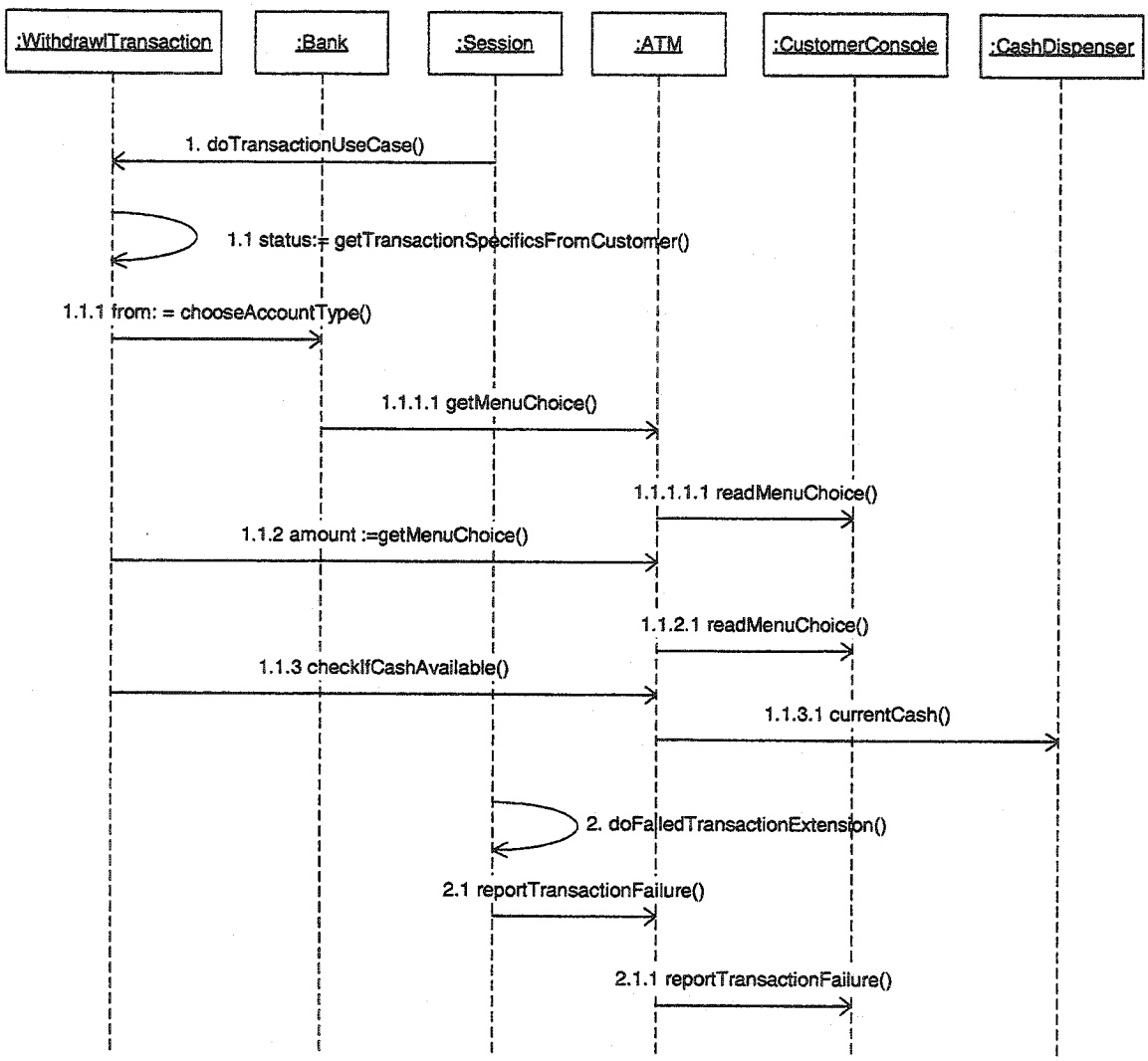


Figure 47 Sequence Diagram of Unsuccessful Withdrawal Use Case

Appendix L Retrieved Scenario Diagram of ATM System

L.1 Retrieved Scenario Diagram of Session Use Case

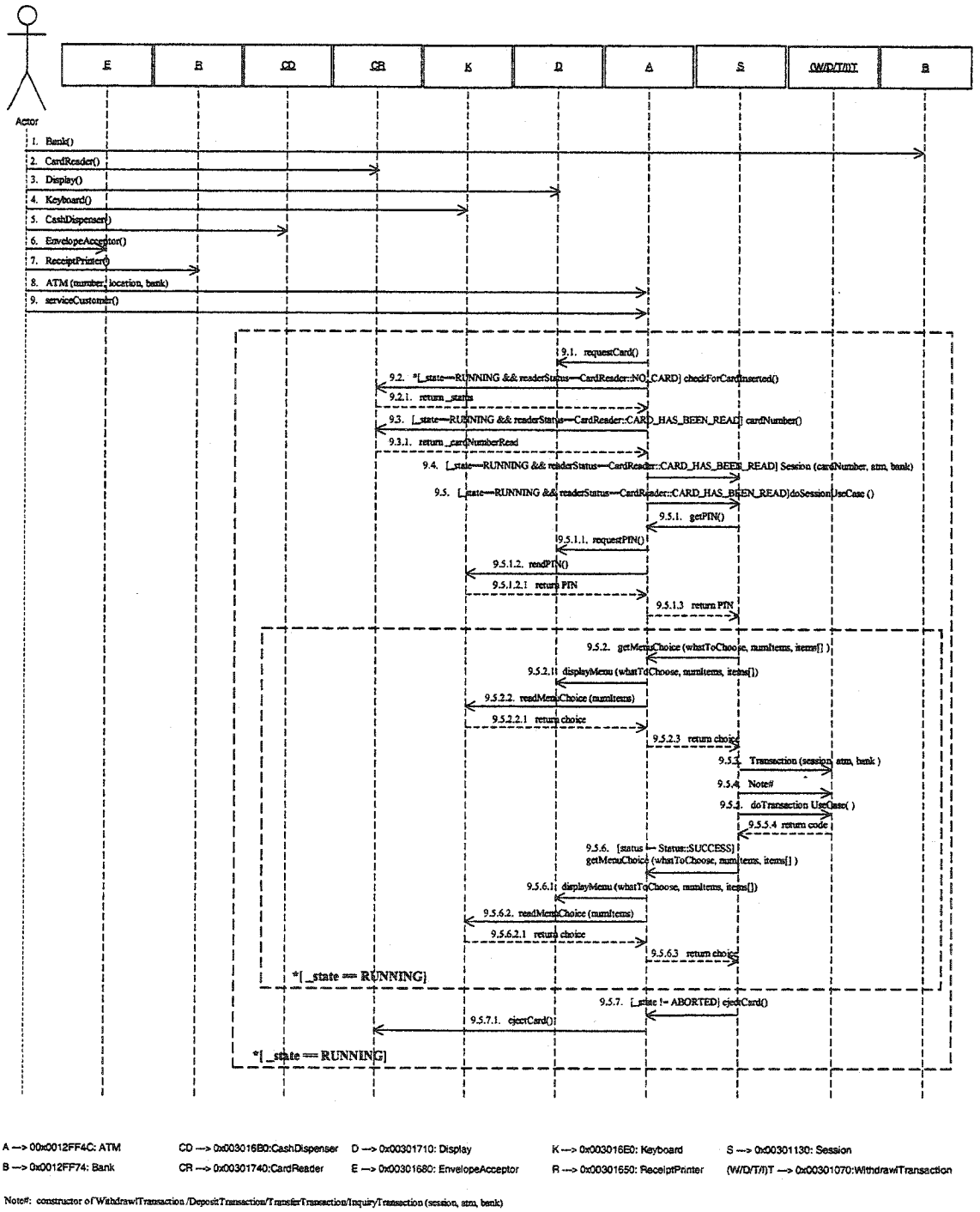


Figure 48 Retrieved Scenario Diagram of Session Use Case

L.2 Retrieved Scenario Diagram of Deposit Use Case

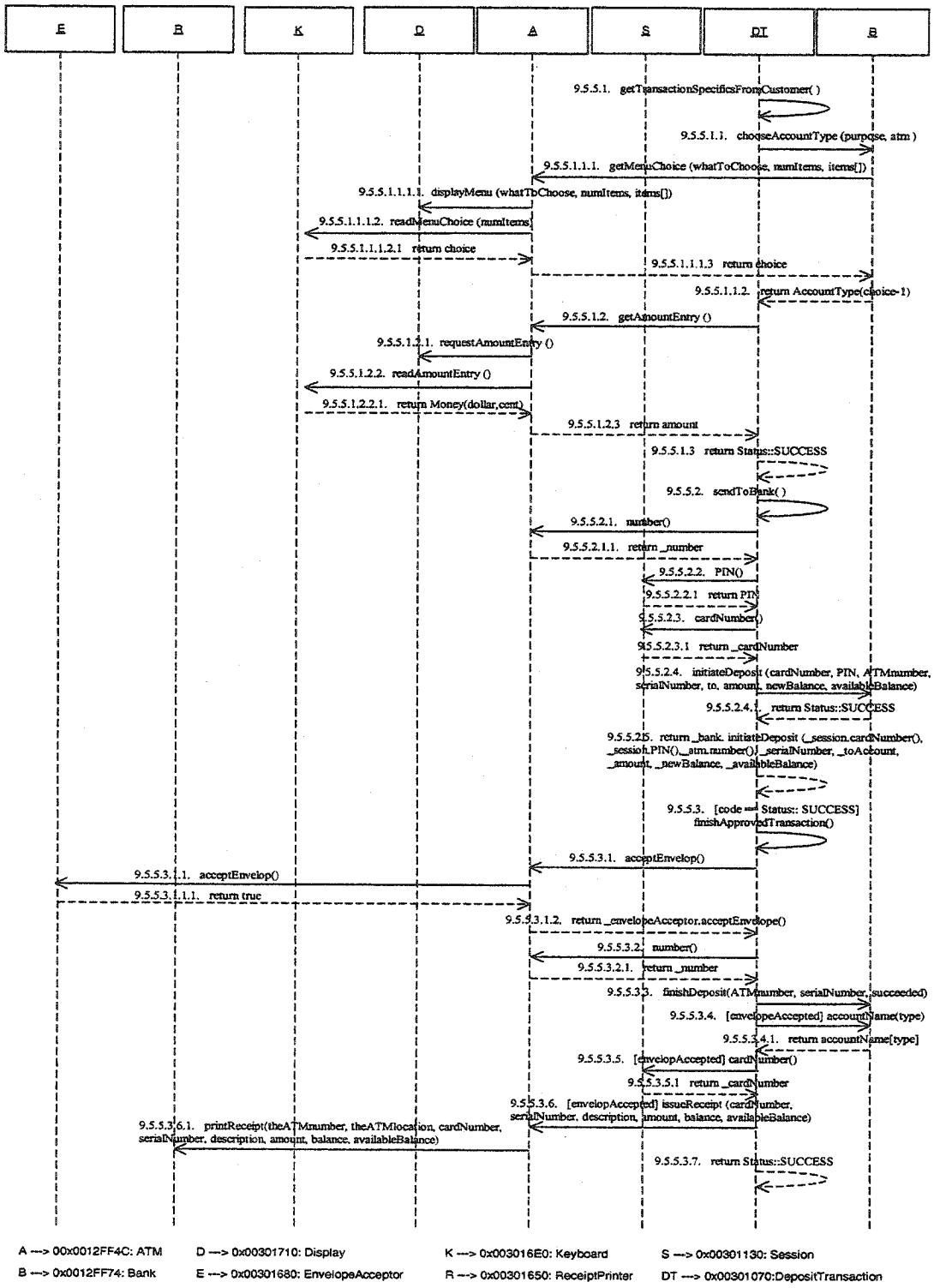


Figure 49 Retrieved Scenario Diagram of Deposit Use Case

L.3 Retrieved Scenario Diagram of Transfer Use Case

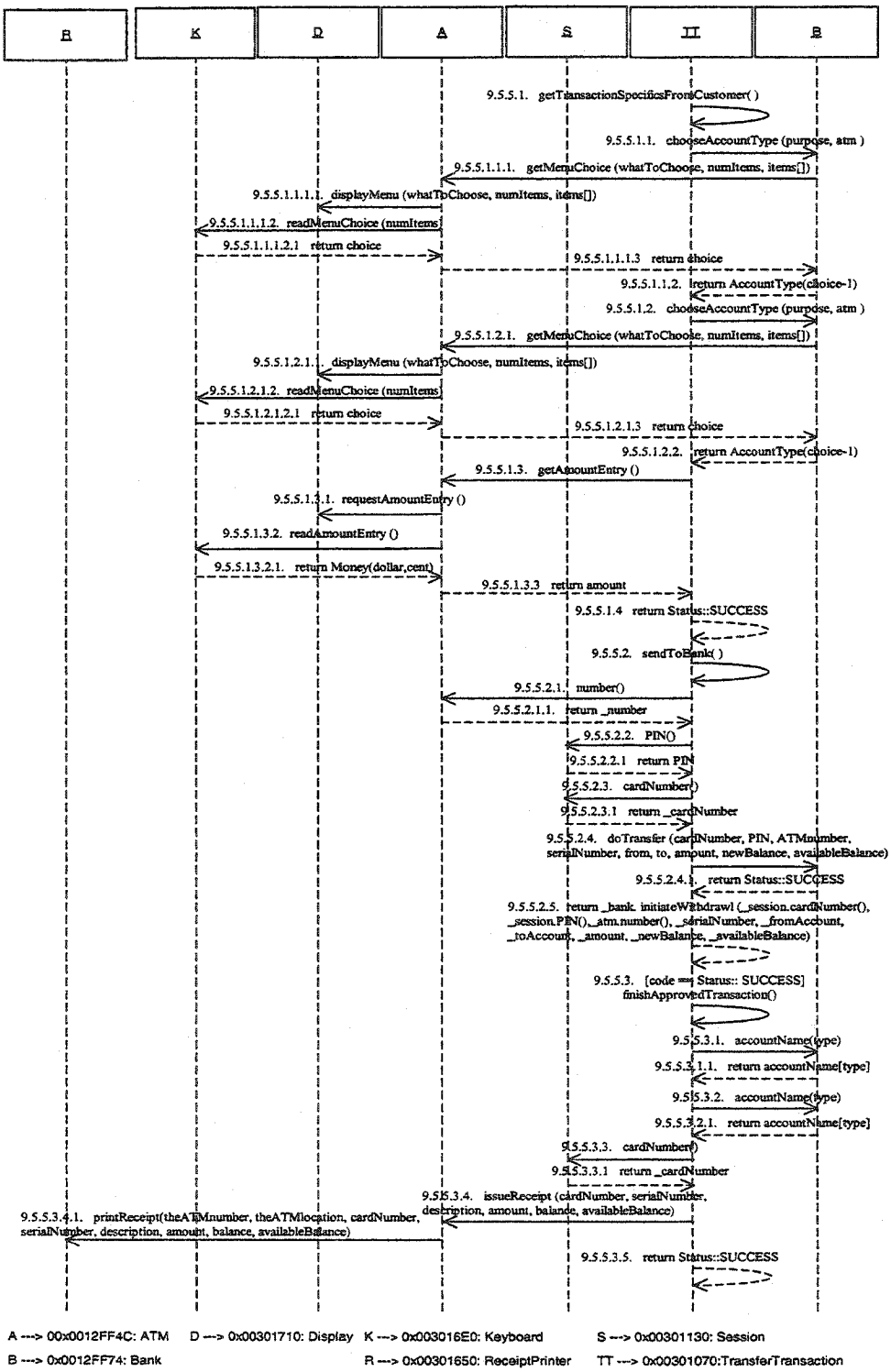


Figure 50 Retrieved Scenario Diagram of Transfer Use Case

L.4 Retrieved Scenario Diagram of Inquiry Use Case

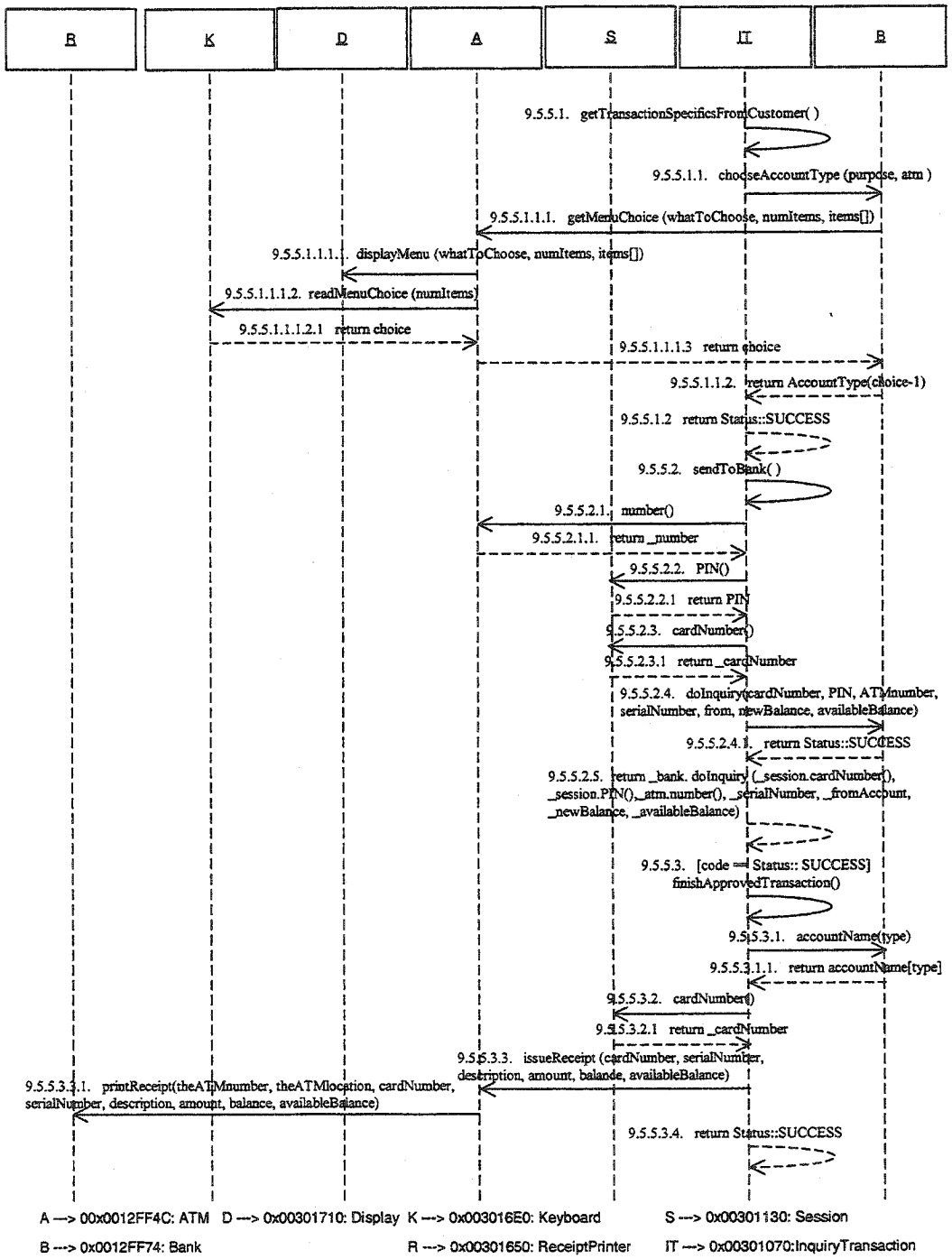


Figure 51 Retrieved Scenario Diagram of Inquiry Use Case