FINITE STATE TETING OF GRAPHICAL USER INTERFACES

Submitted in partial fulfillment of the
Award for the requirement for the Degree of

BACHELOR OF TECHNOLOGY
in
INFORMATION TECHNOLOGY

Under the Supervision of

DR.NITIN

BY

GEETIKA(111410)

TO

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
AND INFORMATION TECHNOLOGY,
JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY,
WAKNAGHAT,SOLAN-173215
HIMACHAL PRADESH,INDIA.

CERTIFICATE

This is to certify that the work entitled “**Finite state testing of graphical user interfaces”** submitted by **Geetika** in the partial fulfillment for the award of degree of Bachelor of Technology in Information Technology from Jaypee University of Information Technology, Waknaghat has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Supervisor:

Dr.Nitin

Assosciate Professor

Department of Computer Science Engg. And Information Technology,

Waknaghat,Solan-173215,Himachal Pradesh,

INDIA.

ACKNOWLEDGEMENT

I would like to express my gratitude to all those who gave me the possibility to complete this project. I want to thank the Department of CSE & IT in JUIT for giving us the permission to commence this project in the first instance, to do the necessary research work.

I am deeply indebted to my project guide Dr. Nitin, whose help, stimulating suggestions and encouragement helped me in all the time of research on this project. I feel motivated and encouraged every time I get his encouragement. For his coherent guidance throughout the tenure of the project, I feel fortunate to be taught by him, who gave me his unwavering support.

Geetika (111410)

Table of Contents

ABSTRACT

It is widely accepted that graphical user interfaces (GUIs) highly affect—positive or negative—the quality and reliability of human-machine systems. In spite of this fact, quantitative assessment of the reliability of GUIs is a relatively young research field. Existing software reliability assessment techniques attempt to statistically describe the software testing process and to determine and thus predict the reliability of the system under consideration (SUC). These techniques model the reliability of the SUC based on particular assumptions and preconditions on probability distribution of cumulative number of failures, failure data observed, and form of the failure intensity function, etc. We expect that the methods used for modeling a GUI and related frameworks used for testing it also affect the factors mentioned above, especially failure data to be observed and prerequisites to be met. Thus, the quality of the reliability assessment process, and ultimately also the reliability of the GUI, depends on the methods used for modeling and testing the system under consideration.

The most Human-Computer-Interfaces will be materialized by Graphical User Interfaces (GUI). With the growing complexity of the computer-based system, also their GUIs become more complex, accordingly making the test process more and more costly. This project introduces a holistic view of fault modeling that can be carried out as a complementary step to system modeling, enabling a precise scalability of the test process, revealing many rationalization potential while testing. Appropriate formal notions and tools enable to introduce efficient algorithms to generate test cases systematically. Based on a basic coverage metric, test case selection can be carried out efficiently. The elements of the approach will be narrated by realistic examples which will be used also to validate the approach.

Chapter 1

INTRODUCTON

Graphical user interfaces (GUIs) play a significant role in improving the usability of the software system, enabling easy interactions between user and system. Thus, a well-developed GUI is an important factor for software quality.

There are two distinct types of construction work while developing software:

- Design, implementation, and test of the programs.
- Design, implementation, and test of the user interface (UI).

We assume that UI might be constructed separately, as it requires different skills, and maybe different techniques than construction of common software. The design part of the development job requires a good understanding of user requirements; the implementation part requires familiarity with the technical equipment, i.e. programming platform, language, etc.Testing requires both: a good understanding of user requirements, and familiarity with the technical equipment. This paper is about UI testing, i.e. testing of the programs that materialize the UI, taking the design aspects into account. Graphical User Interfaces (GUIs) have become more and more popular and common UIs in computer-based systems. Testing GUIs is, on the other hand, a difficult and challenging task for many reasons: First, the input space possesses a great, potentially indefinite number of combinations of inputs and events that occur as system outputs; external events may interact with these inputs. Second, even simple GUIs possess an enormous number of states which are also due to interact with the inputs. Last but not least, many complex dependencies may hold between different states of the GUI system, and/or between its states and inputs. Nevertheless, nowadays it will be taken for granted that most Human-Computer-Interfaces (HCI) will be materialized via GUI. There exist a vast amount of research work for specification of HCI, there has been, however, little well known systematic study in this field, resulting an effective testing strategy which is not only easy to apply, but also scalable in sense of stepwise increasing the test complexity and accordingly the test coverage and completeness of the test process, thus also increasing the test costs in accordance with the test budget of the project. This paper presents a strategy to systematically test GUIs, being capable of test case enumeration for a precise test scalability. Test cases generally require the determination of meaningful test inputs and expected system outputs for these inputs. Accordingly, to generate test cases for a GUI, one has to identify the test objects and test objectives. The test objects are the instruments for the input, e.g. screens, windows, icons, menus, pointers, commands, function keys, alphanumerical keys, etc. The objective of a test is to generate the expected system behavior (desired event) as an output by means of well-defined test input, or inputs. In a broader sense, the test object is the software under test (SUT); the objective of the test is to gain confidence to the PUT. Robust systems possess also a good exception handling mechanism, i.e. they are responsive not in terms of behaving properly in case of correct, legal inputs, but also by behaving good-natured in case of illegal inputs, generating constructive warnings, or tentative correction trials, etc. that help to navigate the user to move in the right direction. In order to validate such robust behavior, one needs systematically generated erroneous inputs which would usually entail injection of undesired events into the SUT. Such events would usually transduce the software under test into an illegal state, e.g. system crash, if the program does not possess an appropriate exception handling mechanism. Test inputs of GUI represent usually sequences of GUI objects activities and/or selections that will operate interactively with the objects (Interaction Sequences – IS, [WHIT], see also [KORE], “Event Sequences”). Such an interactive sequence is complete (CIS), if it eventually invokes the desired system responsibility. From Knowledge Engineering point of the view, the testing of GUI represents a typical planning problem that can

be solved goal-driven [MEM2]: Given a set of operators, an initial state and a goal state, the planner is expected to produce a sequence of operators that will change the initial state to the goal state. For the GUI test problem described above, this means we have to construct the test sequences in dependency of both the desired, correct events and the undesired, faulty events. A major problem is the unique distinction between correct and faulty events (Oracle Problem, [MEM1]). Our approach will exploit the concept of CIS to elegantly solve the Oracle Problem. Another tough problem while testing is the decision when to stop testing (Test Termination
Problem and Testability [LITT, HAML, FRIE]). Exercising a set of test cases, the test results can be satisfactory, but this is limited to these special test cases. Thus, for the quality judgement of the program under test one needs further, rather quantitative arguments, usually materialized by well-defined coverage criteria. The most well known coverage criteria base either on special, structural issues of the program to be tested (implementation orientation/white-box testing), or its behavioral, functional description (specification orientation/black-box testing), or both, if both implementation and specification are available (hybrid/gray-box testing). The present paper will summarize our research work, depicting it by examples lent from real projects, e.g. electronic vending machines which accept electronic and hard money, “emptying” the machine by transfer of the cashed money to a bank account, etc. The favored methods for modeling concentrate on finite-state–based techniques, i.e. state transition diagrams and regular events. For the systematically, scalable generating and selection of test sequences, and accordingly, for the test termination, the notion Edge Coverage of the state transition diagram will be introduced. Thus, our approach is addressed to the black-box testing. It enables an incremental refinement of the specification which may be at the beginning rough and rudimentary, or even not existing. The approach can be, however, also deployed in white-box testing, in a refined format, e.g. using the implementation (source code as a concise description of the SUT) and its control flow diagram as a finite-state machine and as a state

transition diagram, respectively. It introduces the notion of finite-state modeling and regular expressions which will be used both for modeling the system and the faults through interaction sequences. Cost aspects will also be discussed.A basic test coverage metric will be introduced to justifiable generate test cases. An optimization model will be introduced to solve the test termination problem. Some potentials of test cost reduction will be discussed; It includes further rationalization aspects as automatically executing test scripts that have been specified through regular expressions. Further examples and discussion on the validation of the approach will be given. Putting the different components of the approach together, a holistic way of modeling of software development will be materialized, with the novelty that the complementary view of the desired system behavior enables to obtain the precise and complete description of undesired situations, leading to a systematic, scalable, and complete fault modeling.

1.1 Project Specifications

1.1.1.*Hardware Specifications:*

Operating System – Microsoft Windows XP professional 2002,SP2

Primary Memory – 512 MB RAM

Processor – IntelPentium 4 CPU 2.4 GHz

Secondary Memory – 80 HDD

1.1.2.*Software Specifications:*

Frond End – Java(jdk-6-windows-i586),XML,MetaEdit,JCreator

Back End – MS Access (using JDBC driver)

1.2. Java

The Internet helped catapult Java to the forefront of programming, and Java, in turn, has had a profound effect on the Internet. The reason for this is quite simple: Java expands the universe of objects that can move about freely in cyberspace. In a network, two very broad categories of objects are transmitted between the server and your personal computer: passive information and dynamic, active programs.

SIMPLE

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java. Also, some of the more confusing concepts from C++ are either left out of Java or implemented in a cleaner, more approachable manner. Beyond its similarities with C/C++, Java has another attribute that makes it easy to learn: it makes an effort not to have surprising features. In Java, there are a small number of clearly defined ways to accomplish a given task.

OBJECT ORIENTED

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while simple types, such as integers, are kept as high-performance non objects.

ROBUST

The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas, to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. In fact, many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.

MULTITHREADED

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

INTERPRETED AND HIGH-PERFORMANCE

Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be interpreted on any system that provides a Java Virtual Machine. Most previous attempts at crossplatform solutions have done so at the expense of performance. Other interpreted systems, such as BASIC, Tcl, and PERL, suffer from almost insurmountable performance deficits. Java, however, was designed to perform well on very low-power CPUs. As explained earlier, while it is true that Java was engineered for interpretation, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code. "High-performance cross-platform" is no longer an oxymoron.

DYNAMIC

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner.

1.3.HTML

HTML is a language for describing web pages.

-HTML stands for Hyper Text Markup Language

-HTML is not a programming language, it is a markup language

-A markup language is a set of markup tags

-HTML uses markup tags to describe web pages

HTML Tags

HTML markup tags are usually called HTML tags

• HTML tags are keywords surrounded by angle brackets like <html>

• HTML tags normally come in pairs like <b> and </b>

• The first tag in a pair is the start tag, the second tag is the end tag

• Start and end tags are also called opening tags and closing tags

HTML Documents

• HTML documents describe web pages

• HTML documents contain HTML tags and plain text

• HTML documents are also called web pages

The purpose of a web browser (like Internet Explorer or Firefox) is to read HTML documents and display them as web pages. The browser does not display the HTML tags, but uses the tags to interpret the content of the page.

Example Explained

• The text between <html> and </html> describes the web page

• The text between <body> and </body> is the visible page content

• The text between <h1> and </h1> is displayed as a heading

• The text between <p> and </p> is displayed as a paragraph.

1.4.XML

In a sentence, the eXtensible Markup Language (XML) is an open standard providing the means to sharedata and information between computers and computer programs as unambiguously as possible. Once transmitted, it is up to the receiving computer program to interpret the data for some useful purpose thus turning the data into information. Sometimes the data will be rendered as HTML. Other times it might be used to update and/or query a database. Originally intended as a means for Web publishing, the advantages of XML have proven useful for things never intended to be rendered as Web pages. Think of XML as if it represented tab-delimited text files on steroids. Tab-delimited text files are very human readable. They are easy to import into word processors, databases, and spreadsheet applications. Once imported, their simple structure make their content relative easy to manipulate. Tab-delimited text files are even cross-platform and operating system independent (as long as you can get around the carriage- return/linefeed differences between Windows, Macintosh, and Unix computers).

Chapter 2

MODELLING AND DESIGN OF GUI'S

While developing a system, the construction usually starts with creating a model of the system to be built, in order to better understand its "look and feel" [SHNE], including its overall external behavior, mainly in order to validate the user requirements. Thus, modeling of a system requires the ability of abstraction, extracting the relevant issues and information from the irrelevant ones, taking the present stage of the system development into account. While modeling a GUI, the focus is usually addressed rather to the correct behavior of the system as desired situations, triggered by legal inputs. Describing the system behavior in undesired, exceptional situations which will be triggered by illegal inputs and other undesired events are likely to be neglected, due to time and cost pressure of the project. The precise description of such undesired situations is, however, of decisive importance for a user-oriented fault handling, because the user has not only a clear understanding how his or her system functions properly, but also which situations are not in compliance with his or her expectations. In other words, we need a specification to describe the system behavior both in legal and illegal situations, in accordance with the expectations of the user. Once we have such a complete description, we can then also precisely specify our hypotheses to detect undesired situations, and determine the due steps to localize and correct the faults that cause these situations.

2.1.FINITE STATE MODELING

Deterministic finite state automata (FSA), also called finite state, sequential machines have been successfully used for many decades to model sequential systems, e.g. logic design of both combinatorial and sequential circuits, protocol conformance of open systems, compiler construction, but also for UI specification and testing. FSA are broadly accepted for the design and specification of sequential systems for good reasons. First, they have excellent recognition capabilities to effectively distinguish between correct and faulty events/situations. Moreover, efficient algorithms exist for converting FSA into equivalent regular expressions. A FSM can be represented by

- a set of inputs,

- a set of outputs,

- a set of states,

- an output function that maps pairs of inputs and states to outputs,

- a next-state function that maps pairs of inputs and states to next states.

For representing GUI, we will interpret the elements of FSA as follows

- Input set: Identifiable objects that can be perceived and controlled by input/output devices,i.e. elements of WIMPs (Windows, Icons, Menus, and Pointers).

- Output set has two distinct subsets

+ Desired events: Outcomes that the user wants to have, i.e. correct, legal responses,

+ Undesired events: Outcomes that the user does not want, i.e. a faulty result, or an unexpected result that surprises the user.

Our following assumptions that do not constrain the generality:

- We use FSA and its state transition diagram (STD) synonymously.

- STDs are directed graphs, having an entry node and an exit node, and there is at least one path from entry to exit (We will use the notions “node” and “vertex” synonymously).

- Outputs are neglected, in the sense of Moore Automata.

- We will merge the inputs and states, assigning them to the vertices of the STD of the FSA.

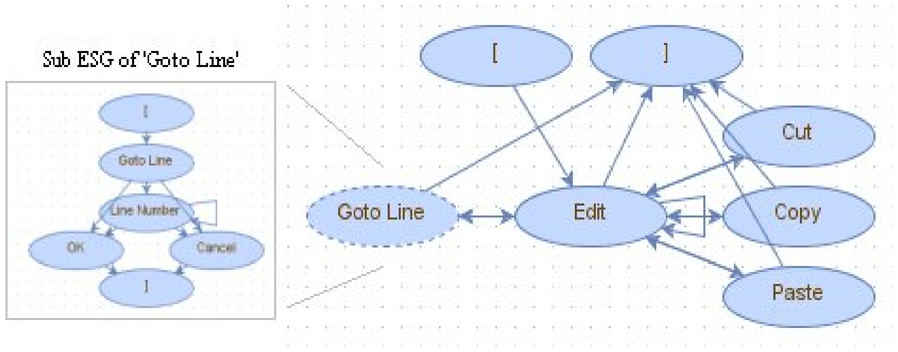
- Next-state function will be interpreted accordingly, i.e. inducing the next input that will be merged with the due state.

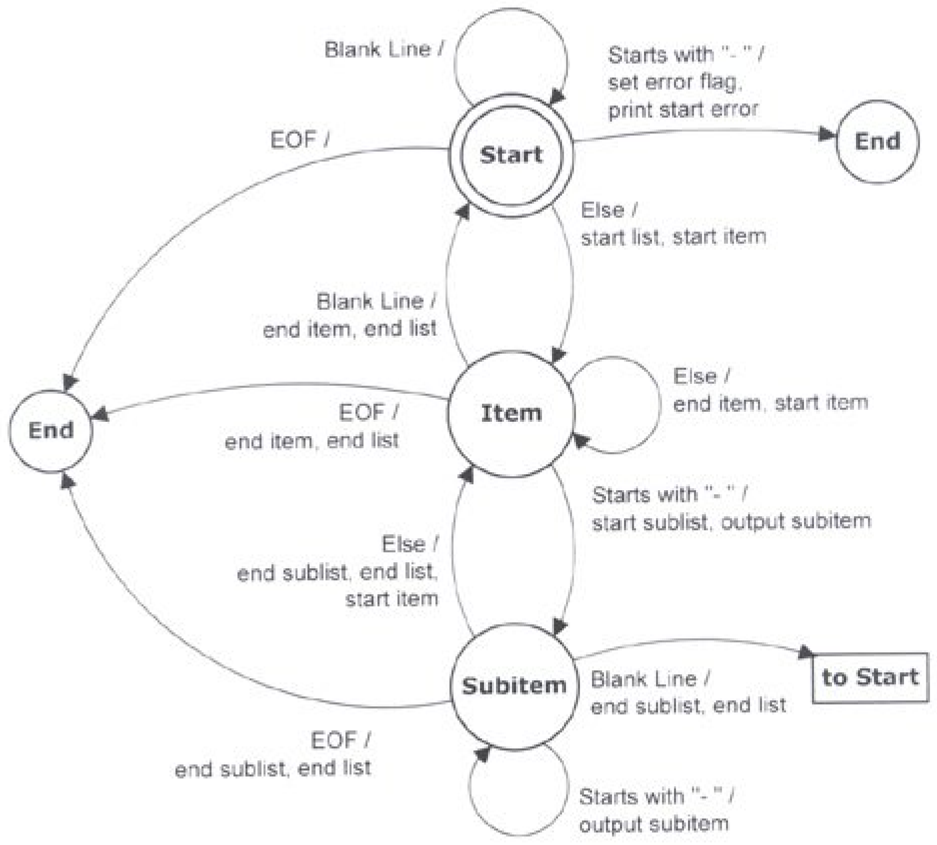
Thus, we use the notions “state” and “input” on the one side and “state”, “system response” and “output” on the other side synonymously, because the user is interested in external behavior of the system, and not its internal states and mechanisms. Thus, we are strongly focusing to the aspects and expectations of the user.Any chain of edges from one vertex to another one, materialized by sequences of user inputsstates-triggered outputs defines an interaction sequence (IS) traversing the FSA from one vertex to another.To introduce informally, we assume that a Regular Expression RegEx consists of symbols a,b, c,...of an alphabet which can be connected by operations

- Sequence (usually no explicit operation symbol, e.g. “ab” means “b follows a”),

- Selection (“+”, e.g. “a+b” means “a or b”),

- Iteration (“*”, Kleene’s Star Operation, e.g. “a*” means “a will be repeated arbitrarily”;“+”: at least one occurrence of “a”).




Blank Line /
Starts with "- " / set error flag, print start error
EOF /
Start
End
Else / start list, start item
Blank Line / end item, end list
Else / end item, start item
End
EOF / end item, end list
Item
Starts with "- " / start sublist, output subitem
Else / end sublist, end list, start item
Subitem
Blank Line / end sublist, end list
to Start
EOF / end sublist, end list
Starts with "- " / output subitem

Example of finite state machine

2.2 INTERACTION SEQUENCES

Once the FSA has been constructed, more information can be gained by means of its state transition graph. First, we can identify now all legal sequences of user-system interactions which may be complete or incomplete, depending on the fact whether they do or do not lead to a well-defined system response that the user expects the system to carry out (Please note that the incomplete interaction sequences are sub-sequences of the complete interaction sequences).Second, we can identify the entire set of the compatible, i.e. legal interaction pairs(IP) of inputs as the edges of the FSA. This is key issue of the present approach, as it will enable us to define the edge coverage notion as a test termination criterion.The generation of the CISs and IPs can be based either on the FSA, or more elegantly, on the corresponding RegEx [GLUS, SAL1], whatever is more convenient for the test engineer. Finite state-based techniques have already been widely used for many years in a rudimentary way in conformance testing of protocols by many authors [SARI, BOCH]. The systematic expansion of the RegEx, as we introduced is, however, relatively new to generate test cases in a scalable way.

2.3 FAULT MODELING THROUGH INTERACTION SEQUENCES

The causes of faults are mostly:

- The expected behavior of the system has been wrongly specified (Specification Errors), or

- the implementation is not in compliance with the specification (Implementation Errors).

In our approach, we will exclude the User Errors, suggesting that the user is always right, i.e.we suggest that there are no user errors. We require that the system must detect all inputs that cannot lead to a desired event, inform the user, and navigate him, or her properly in order to reach a desired situation. One consequence of this requirement is that we need a view that is complementary to the modeling of the system. This can be done by systematically and stepwise manipulation of the FSA that models the system. For this purpose, we introduce the notion Faulty/Incompatible Interaction Pairs (FIP) which consist of inputs that are not legal in sense of the specification. Generate FIP by threefold manipulations:

- Add edges in opposite direction wherever only one way edges exists.

- Add loops to vertices wherever no one exists in the specification.

- Add edges between vertices wherever no one exists.

Adding all manipulations to the FSA defines the Completed FSA.

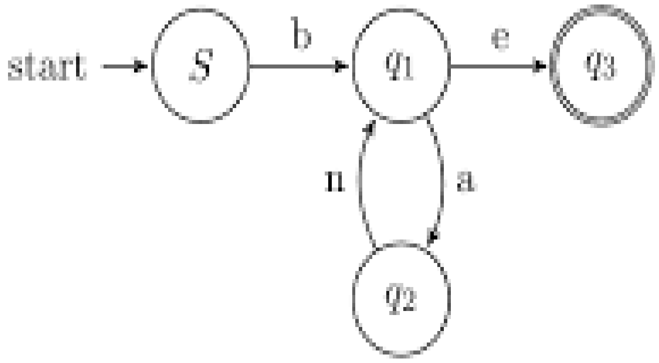


Figure 1: Finite State Automaton, accepting the pattern $b(an)+e$

Now we can construct all potential interaction faults systematically building all illegal combinations of symbols that are not in compliance with the specification. Once we have generated a FIP, we can extend it through an IS that starts with entry and end with the first symbol of this FIP; we have than a faulty/illegal complete interaction sequence (FCIS), bringing the system into a faulty situation. Please note that the attribute "complete" within the phrase FCIS may not imply that the exit node of the FSA must be necessarily reached; once the system has been conducted into a faulty state, it cannot accept further illegal inputs, in other words, an undesired situation cannot be even more undesired, or a fault cannot be faultier. Prior to further input, the system must recover, i.e. the illegal event must be undone and the system must be conducted into a legal, state through a backward or forward recovery mechanism [RAND].

The test process can be summarized now as follow:

1. Construct the complete set of test cases which includes all types of interaction sequences,i.e. all CISs and FCISs (Predictability of the tests, requiring oracles).

2. Input CISs and FCISs to transduce the system into a legal or illegal state, respectively(Controllability).

3. Observe the system output that enables a unique decision whether the output leads to a desired system response or an undesired, faulty event occurs which invokes an error message/warning, provided that a good exception handling mechanism [GOOD] has been materialized (Observability).

If the steps 1 to 3 can be carried out effectively, we have a monitoring capability of testing process that leads to a high grade of testability. Monitoring requires a special structure of software which must be designed carefully, considering the methods and principles of the modern Software Engineering ("Design for Testability").

2.4 HANDLING CONTEXT SENSITIVITIES

A problem we have to encounter with during system modeling stems from the convenience of using the same commands, or icons for similar operations in different hierarchical levels of the application, e.g. delete for deleting a symbol, but also a record, or even a file. Upon the context information, the system can usually carry out the proper action. Our approach eliminates, however, the hierarchy information while abstracting the real system into the model. While constructing the IPs and FIPs, and accordingly the CISs and FCISs, we have to differ between the state a that leads to b and the state a that can be reached by b and c. Often,an ambiguity that can be best resolved by indexing, i.e.a1 for noting the first appearance of the a, and a2 for the second one.

A good naming policy keeps the Hamming Distance of the identifiers of states with semantic and pragmatic similarity as small as possible, not only in order to enable a good proximity of the associated notions, but also the unambiguous distinction of the corresponding states of the FSA to avoid inconsistencies while producing the IPs, FIPs, CIS, and FCISs. As an example, we could name the operation “delete”

- delete_c, if we want to delete a character,

- delete_w, if we want to delete a word,

- delete_r, if we want to delete a record,

- delete_f, if we want to delete a file,etc., or assign different, but associative icons to such operations.

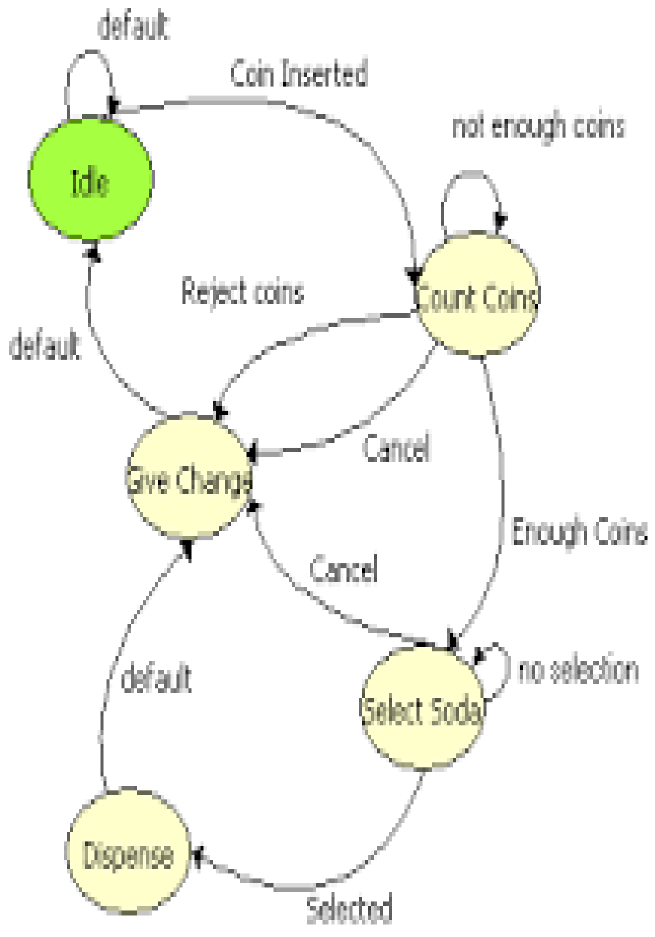
Chapter 3

VALIDATION OF APPROACH

The approach we described here has been used in different environments,
i.e. we could extend and deepen our theoretical view interactively along practical insight during several applications.Following, we will summarize our experiences with the approach; instead of a full documentation which would run out space available in a brief report and the patience of the reader, we will rather display some spots, instantaneously enlightening some relevant aspects,focusing on the fault detection capabilities of the introduced method. We chose examples from a broad variety of applications to emphasize the versatility of the approach.

3.1 Vending Machine

For the sake of clearance and ease of understanding, we reduce here the full capabilities of the studied modern vending machines considerably. Nowadays, such machines can accept money or credit cards for issuing train or flight tickets, carrying out transfers to and from a control system according to a communication protocol, considering the due security procedures, etc. We simplify different
components of a vending machine and provide a simple example.

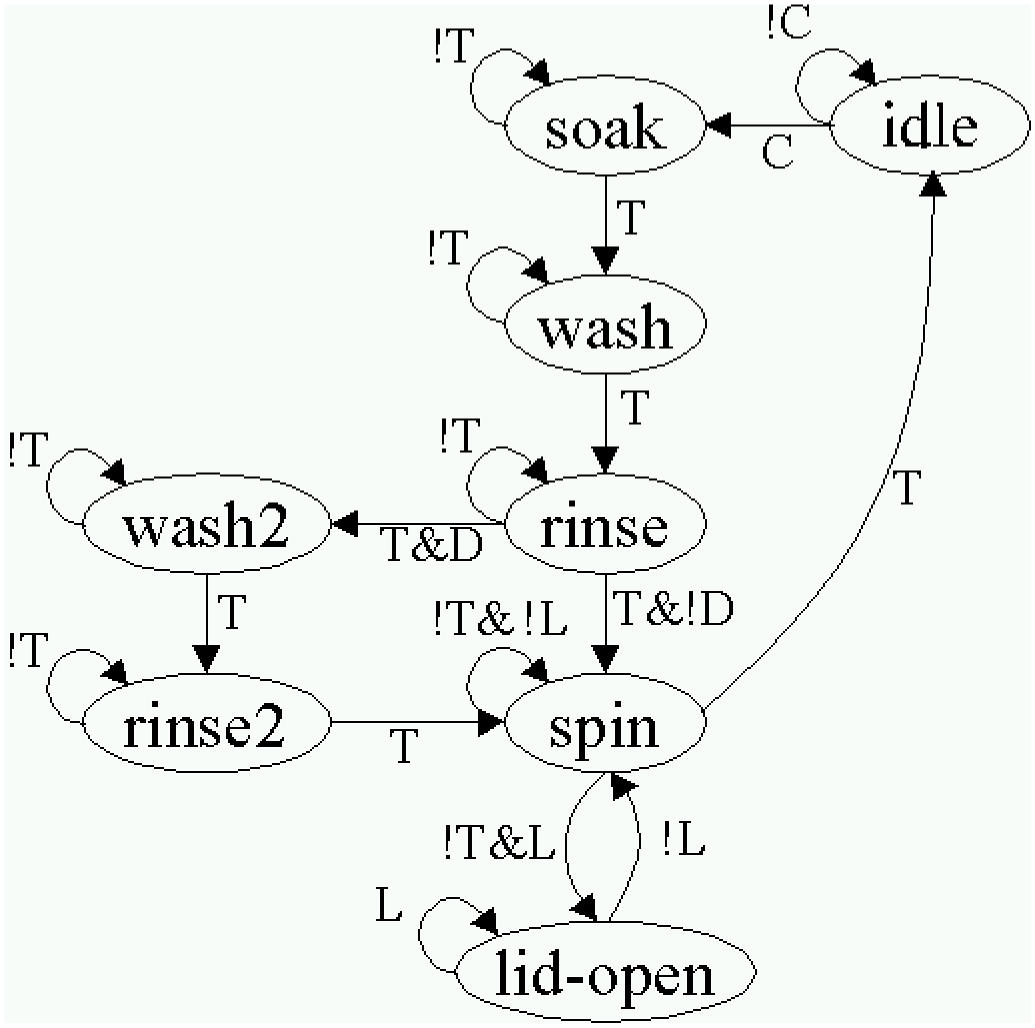


FSA of Vending Machine

3.2 Washing Machine

The following figure describes the FSA of vending machine describing all the

Legal and illegal connections between all the nodes of a washing machine.



FSA of Washing Machine

Skipping the complete analysis,we exploit for fault analysis describing only some of the intresting facts :

Fault 1: During the entire washing process, the door should be kept locked until the stop key has been touched.

Fault 2: Before the washing process starts, a sensor must ensure that some clothes are filled in.

Fault 3: The stand_by mode is reachable only from on or stop states. The transition from washing state to the stand_by must be excluded through an appropriate mechanism.

Fault 4: No key signal should be accepted if the machine is connected, but is down

Fault 5: If the machine is in one of the washing states, any transition into another washing state must be excluded, i.e. during the main wash, the selection pre wash should be locked until the main wash has finished.

3.3 Threats to Validity

In practice, using several different types of GUI-based applications, testing frameworks, and performing larger number of experiments would achieve a greater confidence in the obtained results. Also, EFG models are derived from ESG models. Thus,preparing EFG models independently may change the results

3.4.RESULT ANALYSIS OF FAULT MODELING

While some of the results of the fault analyses are in compliance with our expectations, some other results are surprising. Instead of listing long columns of statistical data, we summarize following directly the results of the analysis of these data.

- Incomplete Exception Handling: The initial concept for handling the undesired events, i.e.exceptions was in most cases strongly incomplete. The number of the exceptions could be increased in average about 70%. This result was expected: Our approach was originally founded to help the routinization of the exception handling.
- Conceptual flaws: Not as often as the forgotten undesired events, we found that also some major elements of the modeled system were missing, because the developer simply forgot them. In other words, the FSA was not lack of the edges, but vertices (Remember: vertices present inputs and states that merge). Thus, our initial concept was seriously defect, having forgotten, or corrupted some vital components. The number of the vertices could be increased in average about 20%. This result was not expected: The approach helped to accelerate the conceptual maturation process considerably, supporting the creative mental activities.
- Another unexpected result was the willingness of the user to participate at the design process. Even the user without any knowledge in Automata Theory and Formal Languages could understand the approach very fast, especially the Transition Diagrams (They called them “Bubble Diagrams” which they could operate skillfully with). The participation of the user helped to complete the exception handling (they contributed to find about half of the forgotten exceptions), but also to detect the

conceptual flaws (about 30% of them).We recommend to use the approach incrementally, i.e. start very early, even with a rudimentary model of the system which should then be completed, adding the illegal connections to determine the faulty interaction pairs (FIP). The discussion of these FIPs is
very often the most fruitful part of the modeling, leading to detect conceptual defects,and systematically completing the diagram not only by edges, but also by vertices. During this process, the test cases will be also systematically and scalable collected.

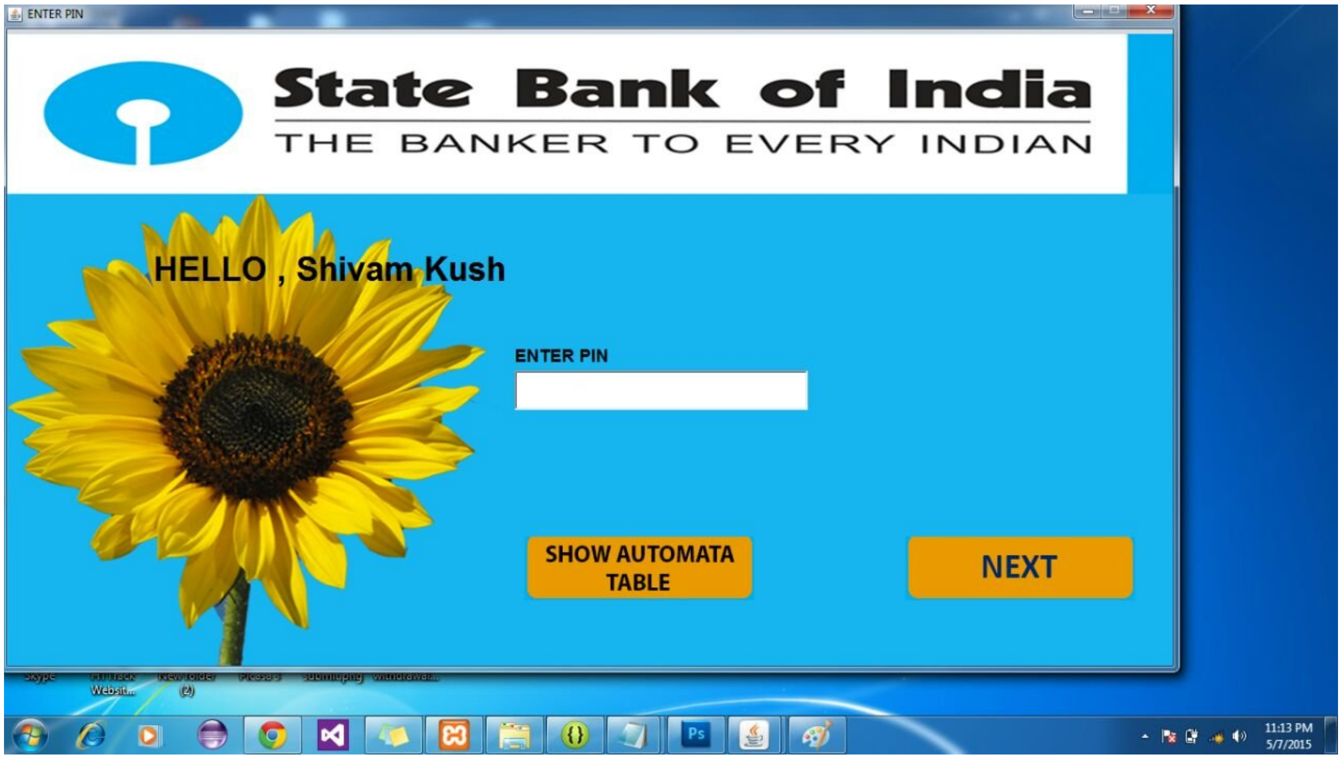

ENTER PIN
State Bank of India
THE BANKER TO EVERY INDIAN
HELLO , Shivam Kush
ENTER PIN
SHOW AUTOMATA TABLE
NEXT

Fig.1

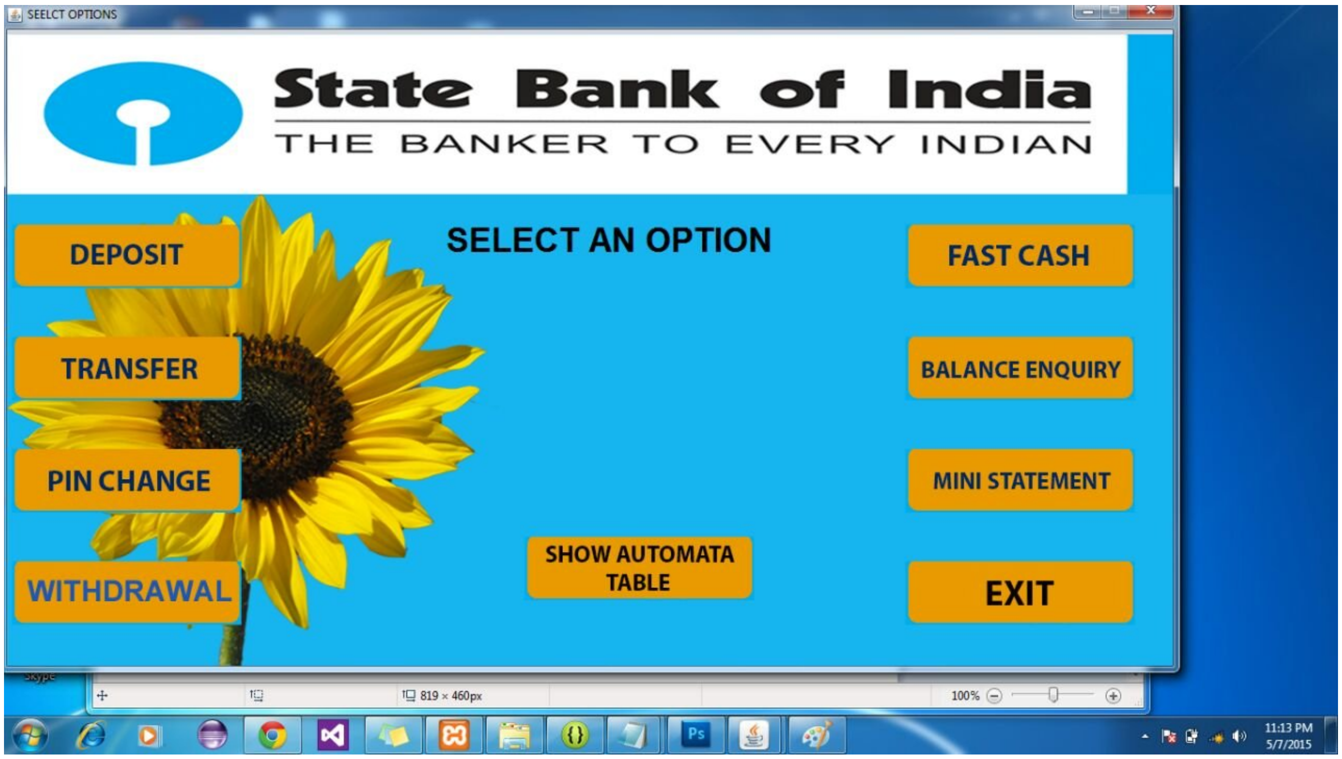
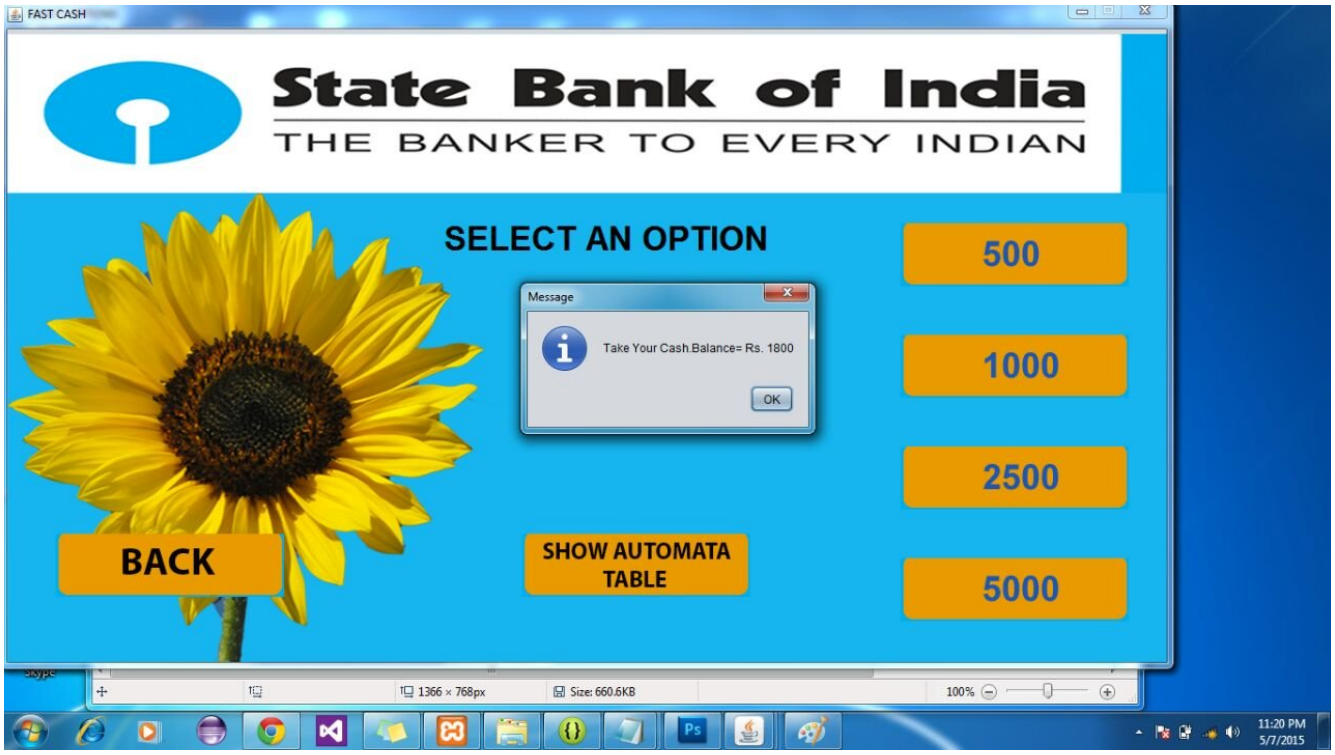
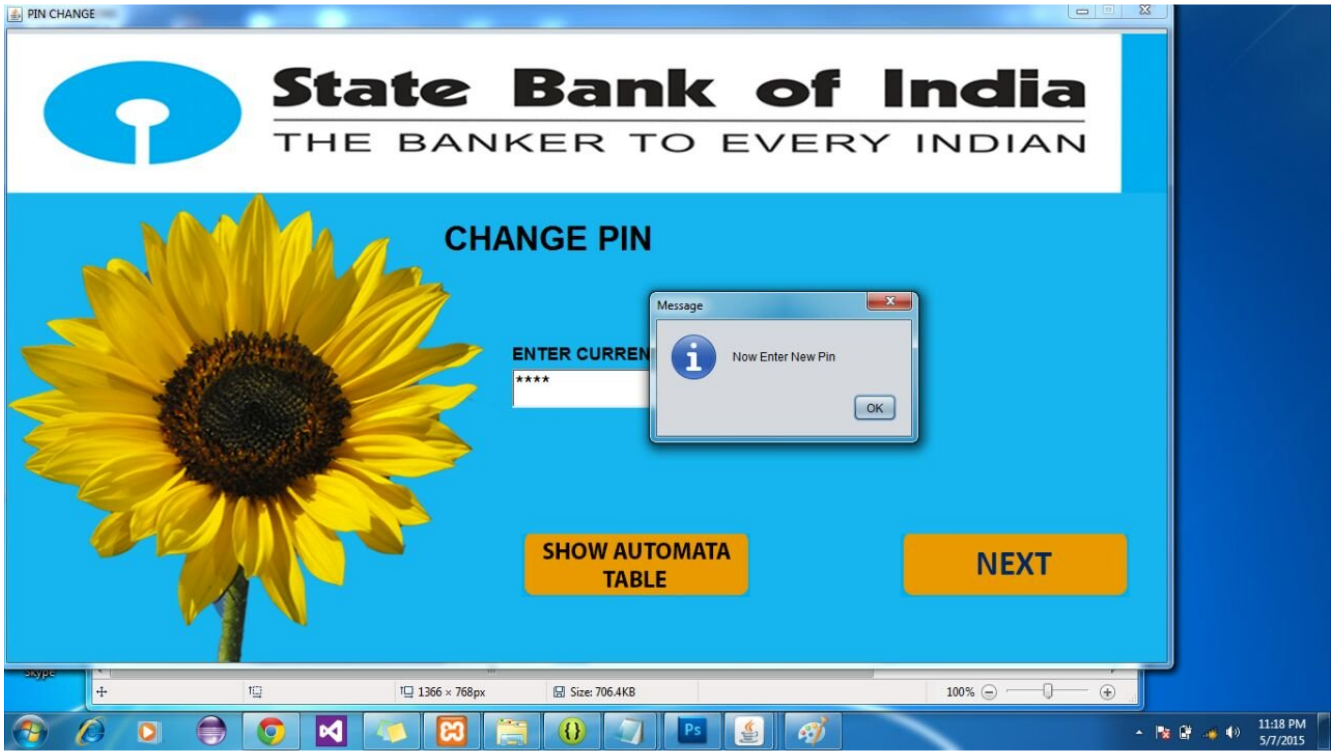

SEELCT OPTIONS
State Bank of India
THE BANKER TO EVERY INDIAN
SELECT AN OPTION
DEPOSIT
TRANSFER
PIN CHANGE
WITHDRAWAL
FAST CASH
BALANCE ENQUIRY
MINI STATEMENT
SHOW AUTOMATA TABLE
EXIT

Fig.2



Take Your Cash.Balance= Rs. 1800

Fig.3



Now Enter New Pin

Fig.4

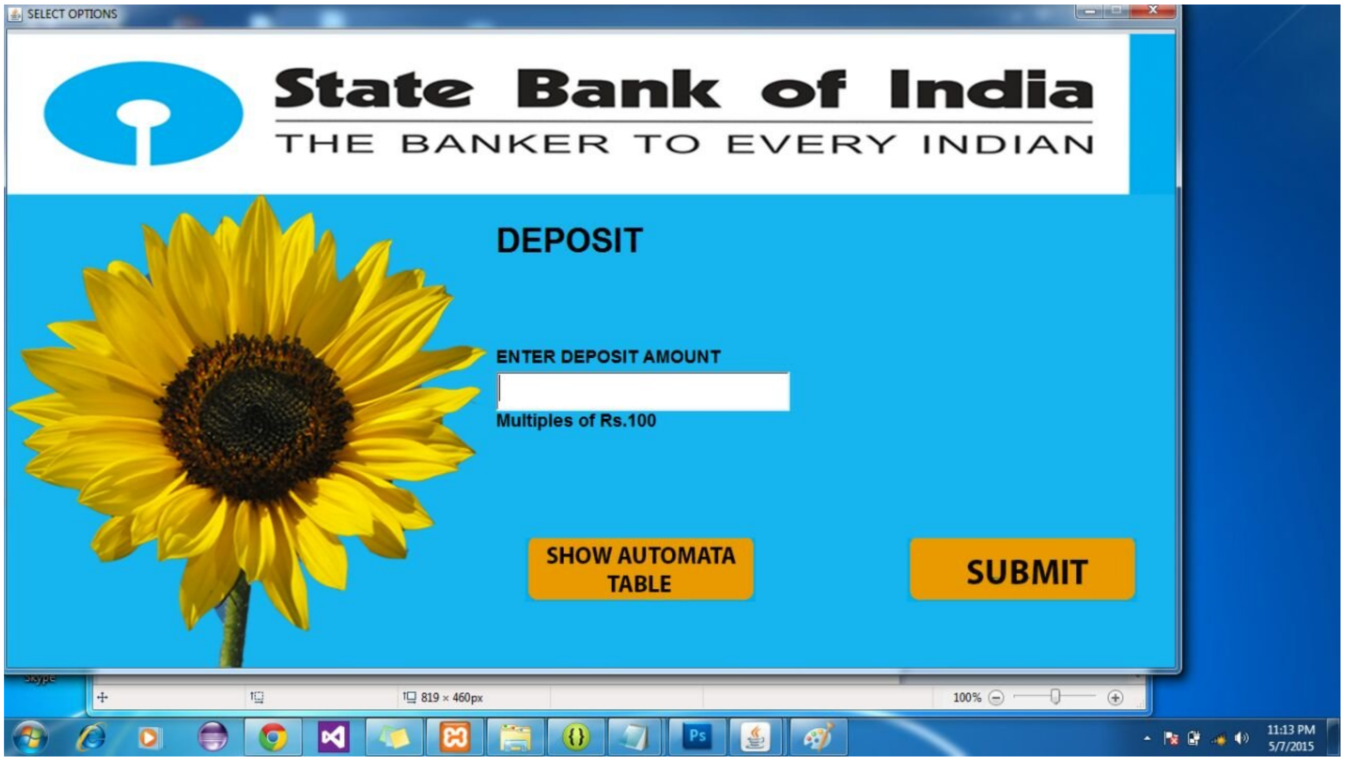
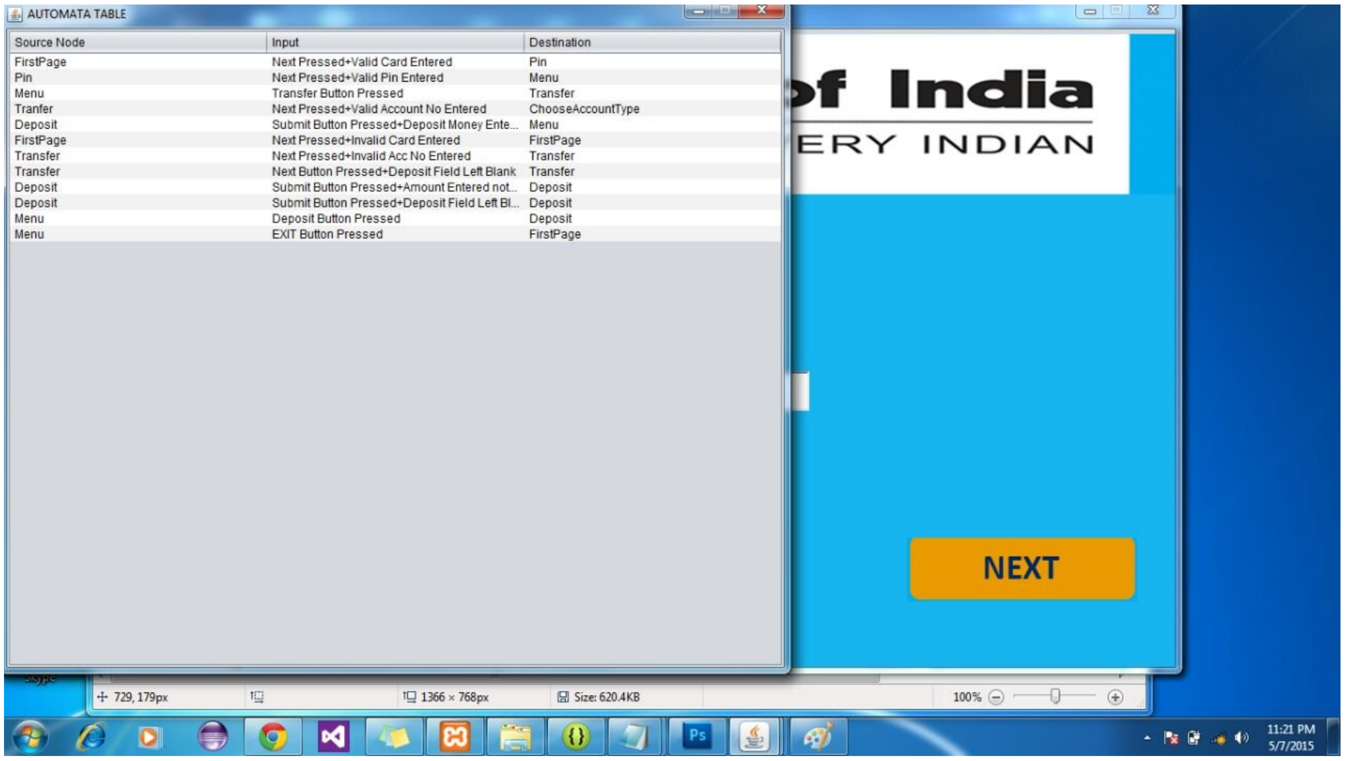
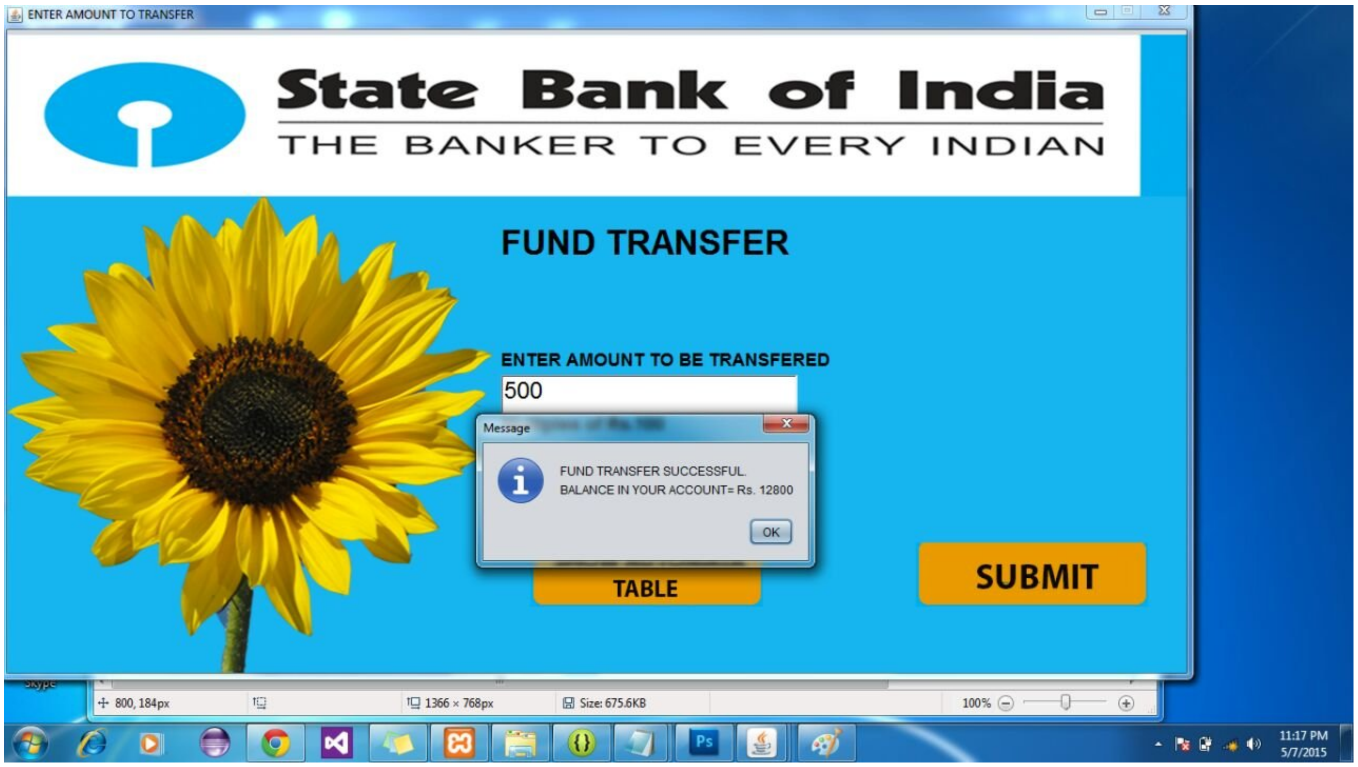


Fig.5



Source Node	Input	Destination
FirstPage	Next Pressed+Valid Card Entered	Pin
Pin	Next Pressed+Valid Pin Entered	Menu
Menu	Transfer Button Pressed	Transfer
Tranfer	Next Pressed+Valid Account No Entered	ChooseAccountType
Deposit	Submit Button Pressed+Deposit Money Ente...	Menu
FirstPage	Next Pressed+Invalid Card Entered	FirstPage
Transfer	Next Pressed+Invalid Acc No Entered	Transfer
Transfer	Next Button Pressed+Deposit Field Left Blank	Transfer
Deposit	Submit Button Pressed+Amount Entered not...	Deposit
Deposit	Submit Button Pressed+Deposit Field Left Bl...	Deposit
Menu	Deposit Button Pressed	Deposit
Menu	EXIT Button Pressed	FirstPage

Fig.6



FUND TRANSFER SUCCESSFUL.
BALANCE IN YOUR ACCOUNT= Rs. 12800

Fig.7

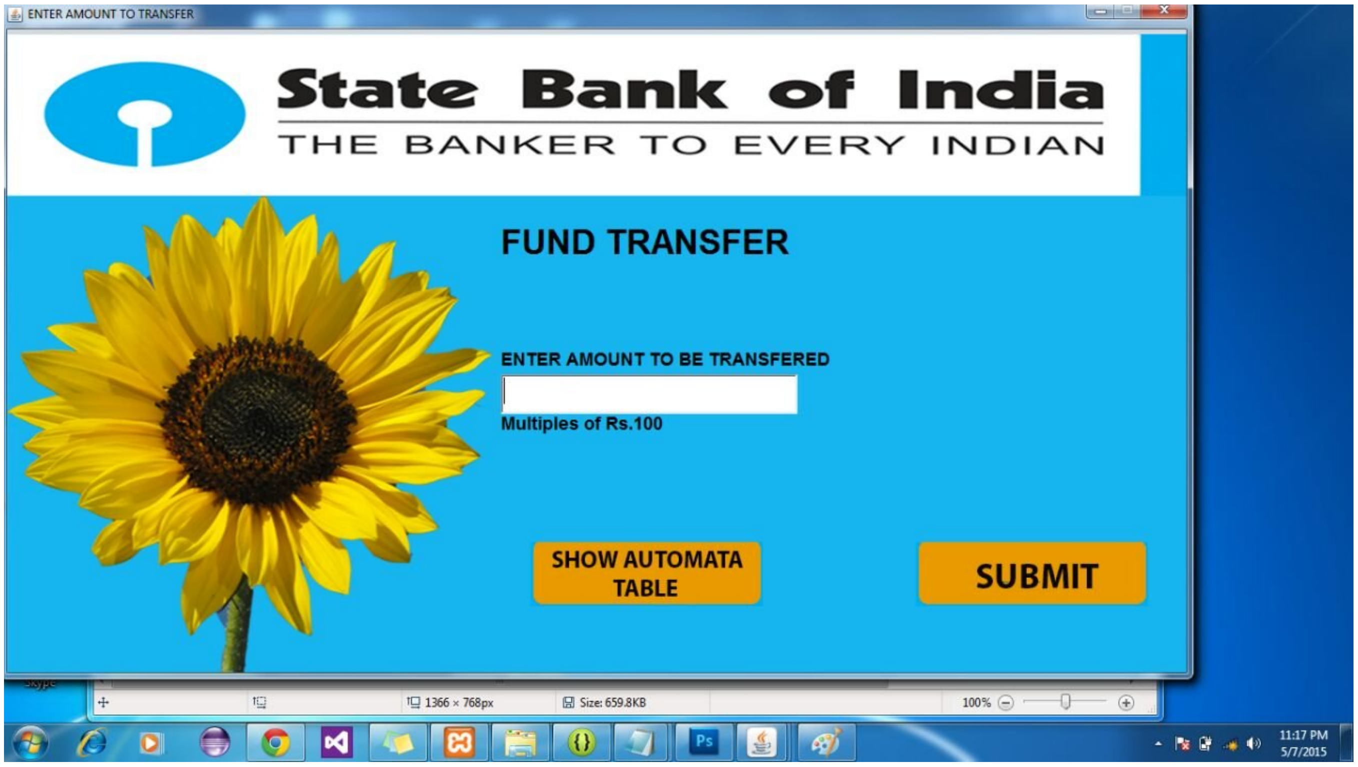


Fig.8

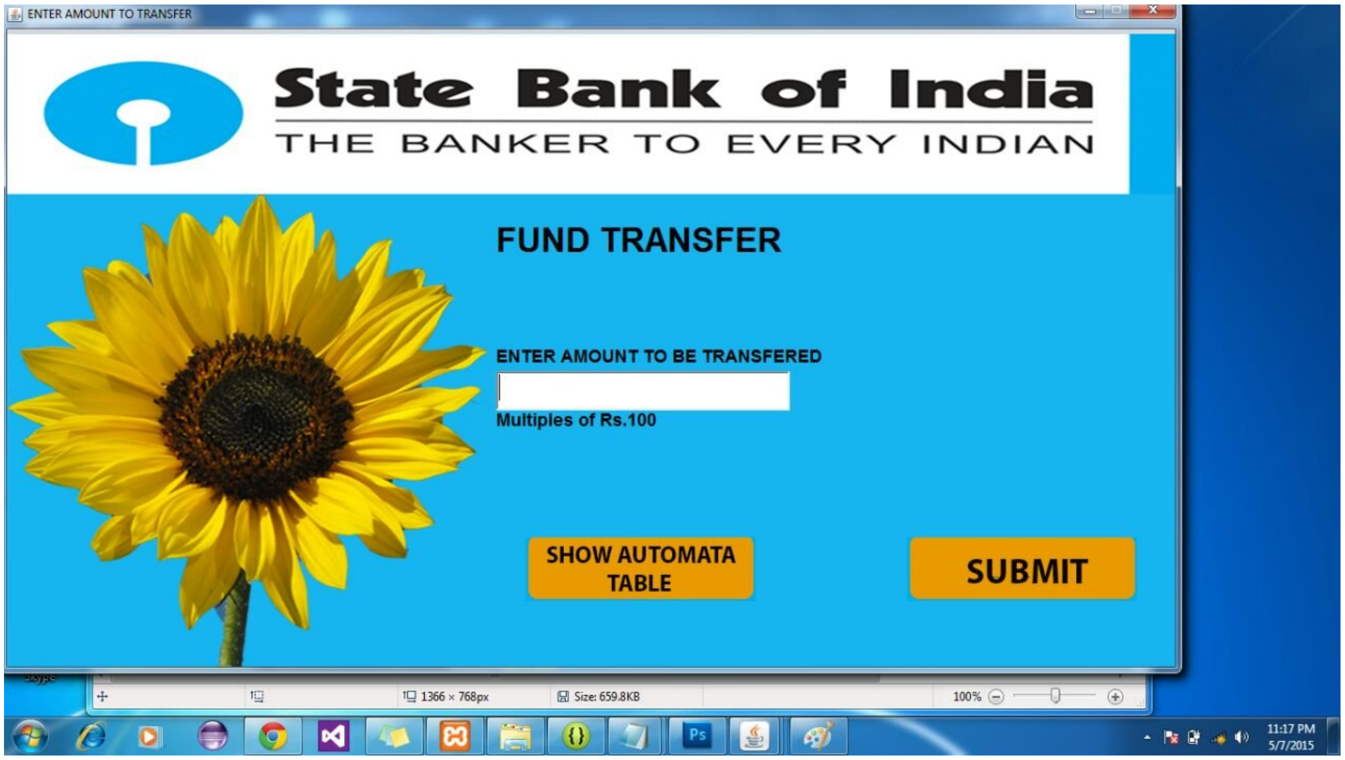

ENTER AMOUNT TO TRANSFER
State Bank of India
THE BANKER TO EVERY INDIAN
FUND TRANSFER
ENTER AMOUNT TO BE TRANSFERED
Multiples of Rs.100
SHOW AUTOMATA TABLE
SUBMIT

Fig.9

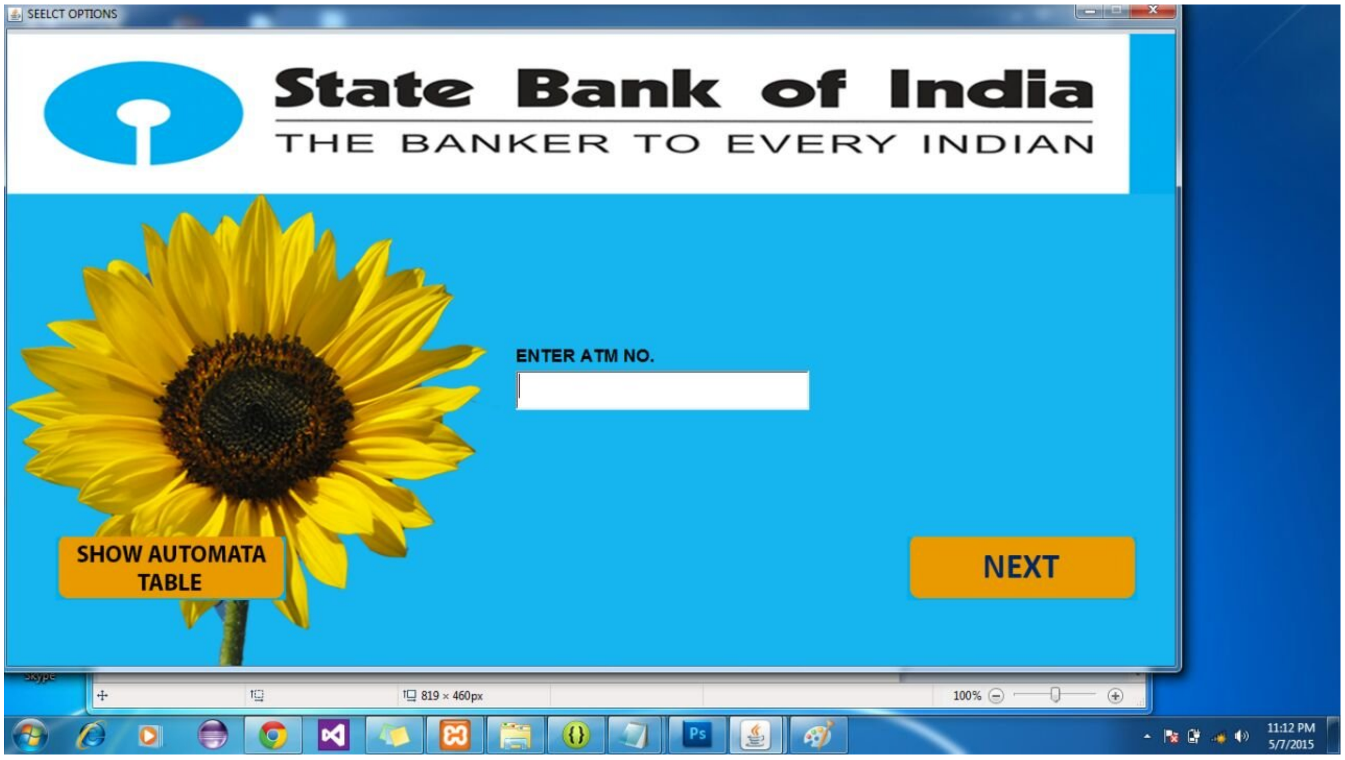
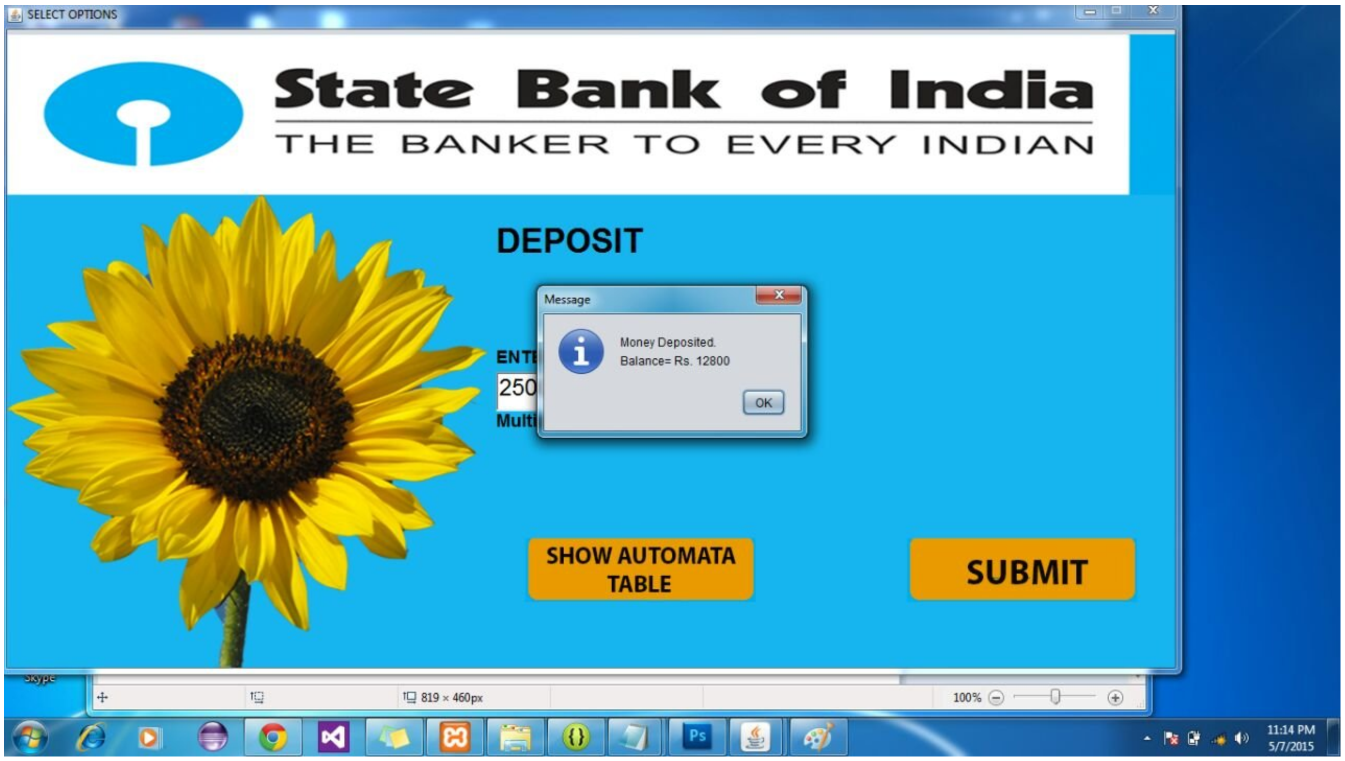

SEELCT OPTIONS
State Bank of India
THE BANKER TO EVERY INDIAN
ENTER ATM NO.
SHOW AUTOMATA TABLE
NEXT

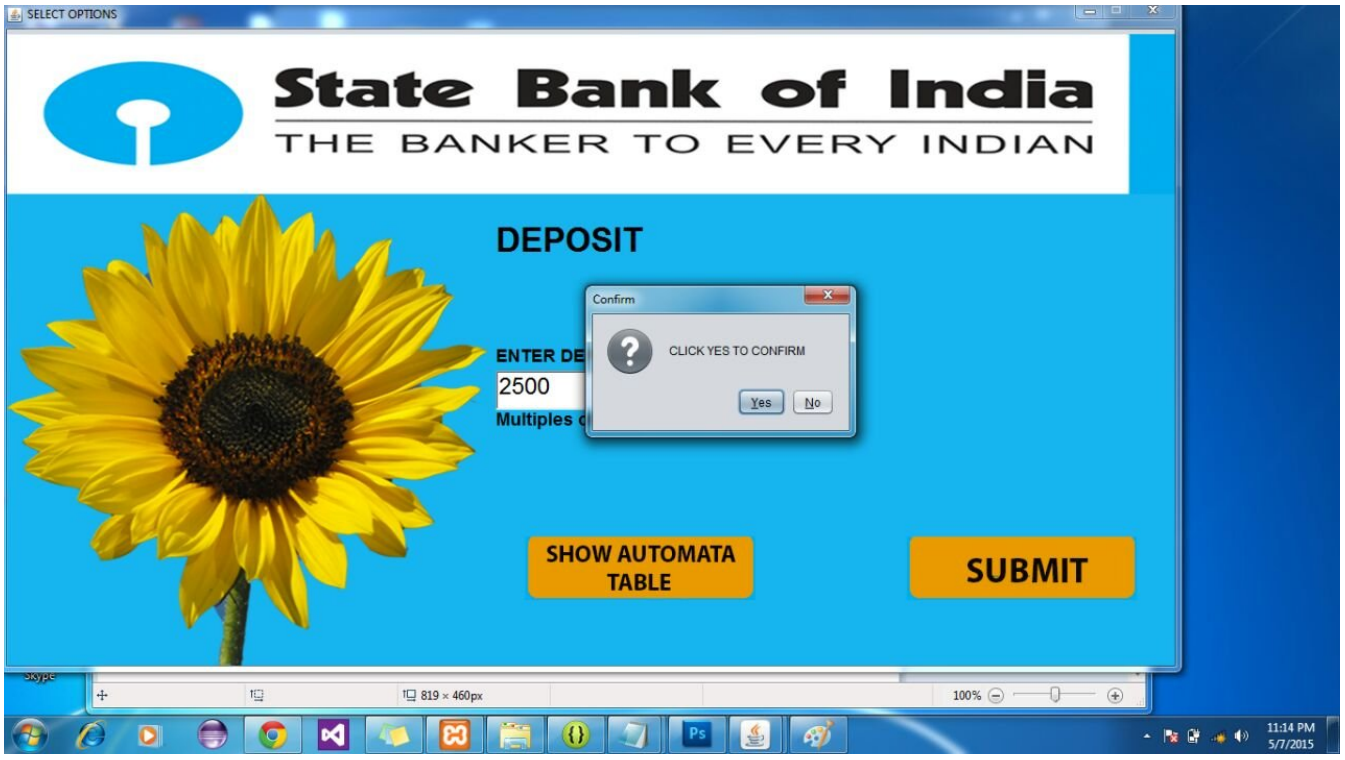
Fig.10



Message

Money Deposited.
Balance= Rs. 12800

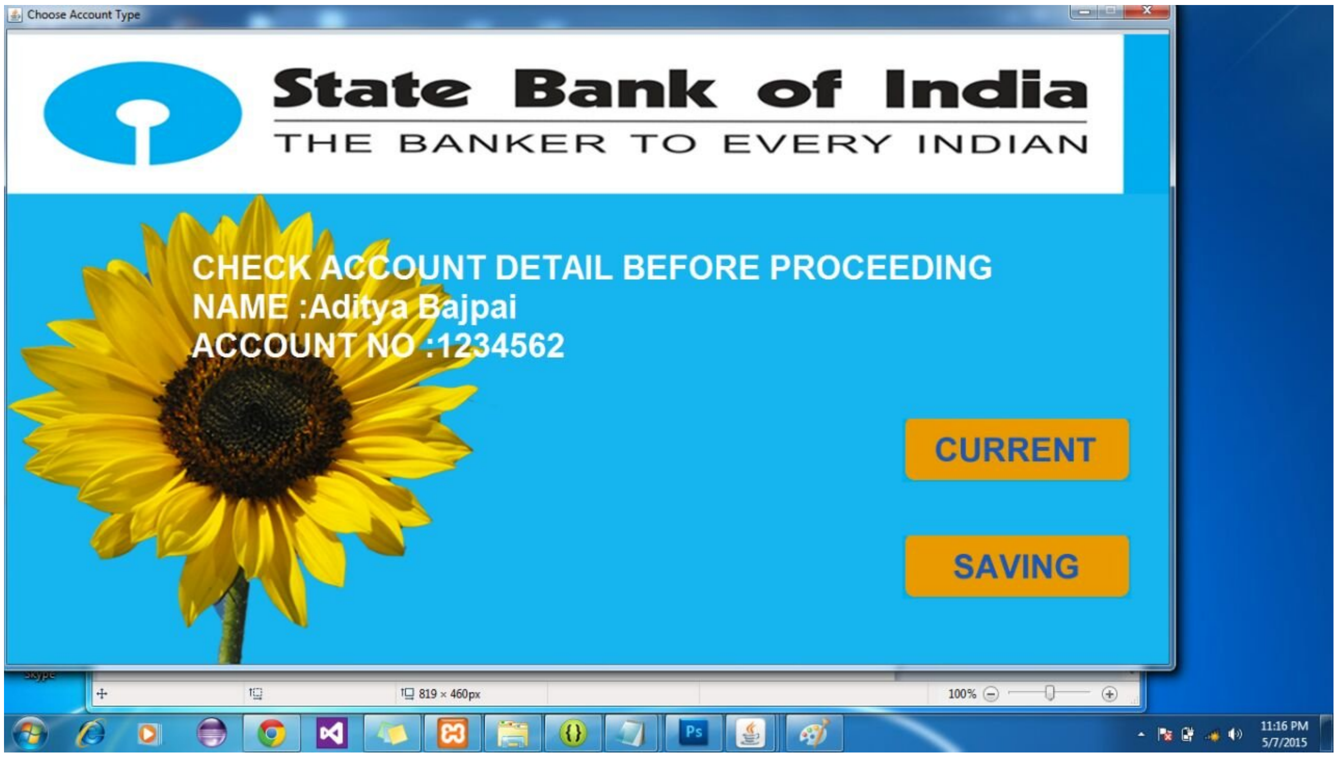
Fig.11



Confirm

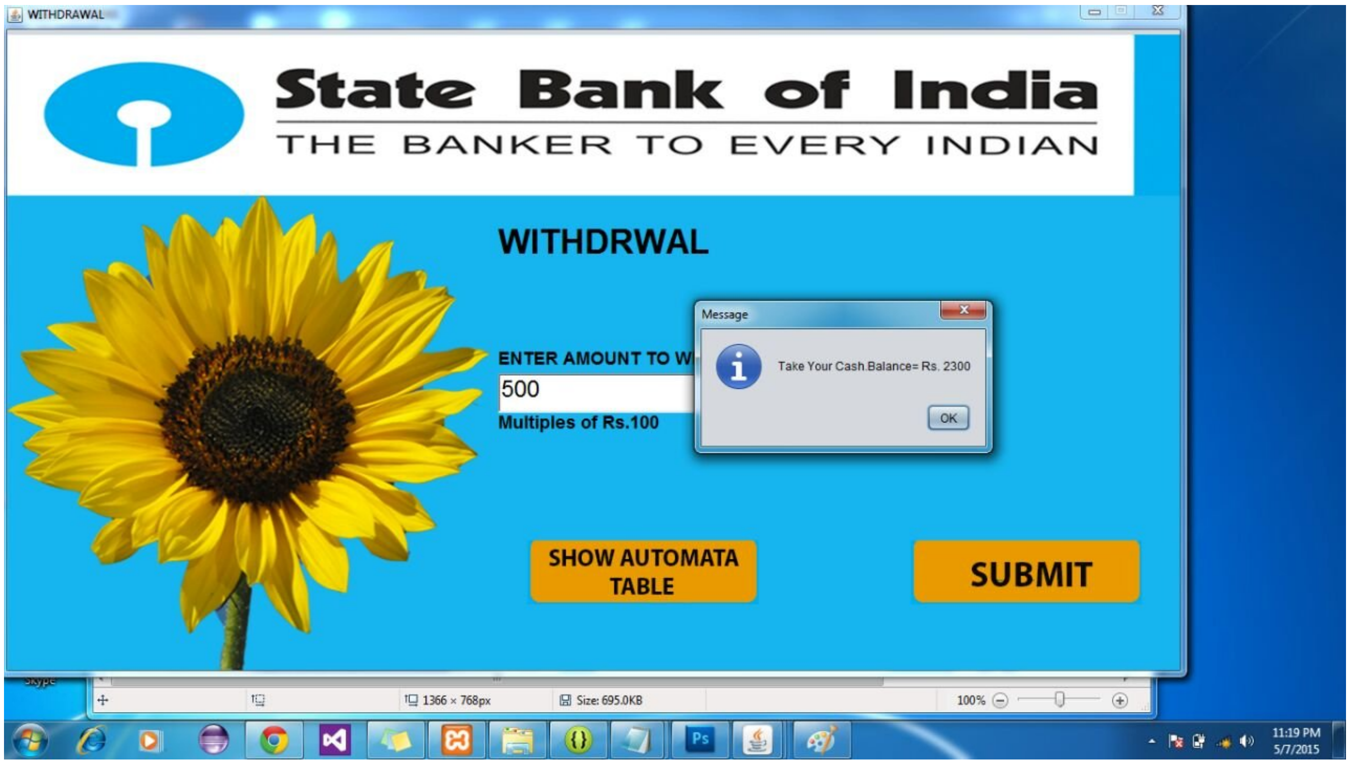
CLICK YES TO CONFIRM

Fig.12



CHECK ACCOUNT DETAIL BEFORE PROCEEDING
NAME :Aditya Bajpai
ACCOUNT NO :1234562

Fig.13



ENTER AMOUNT TO W
500
Multiples of Rs.100

Take Your Cash.Balance= Rs. 2300

Fig.14

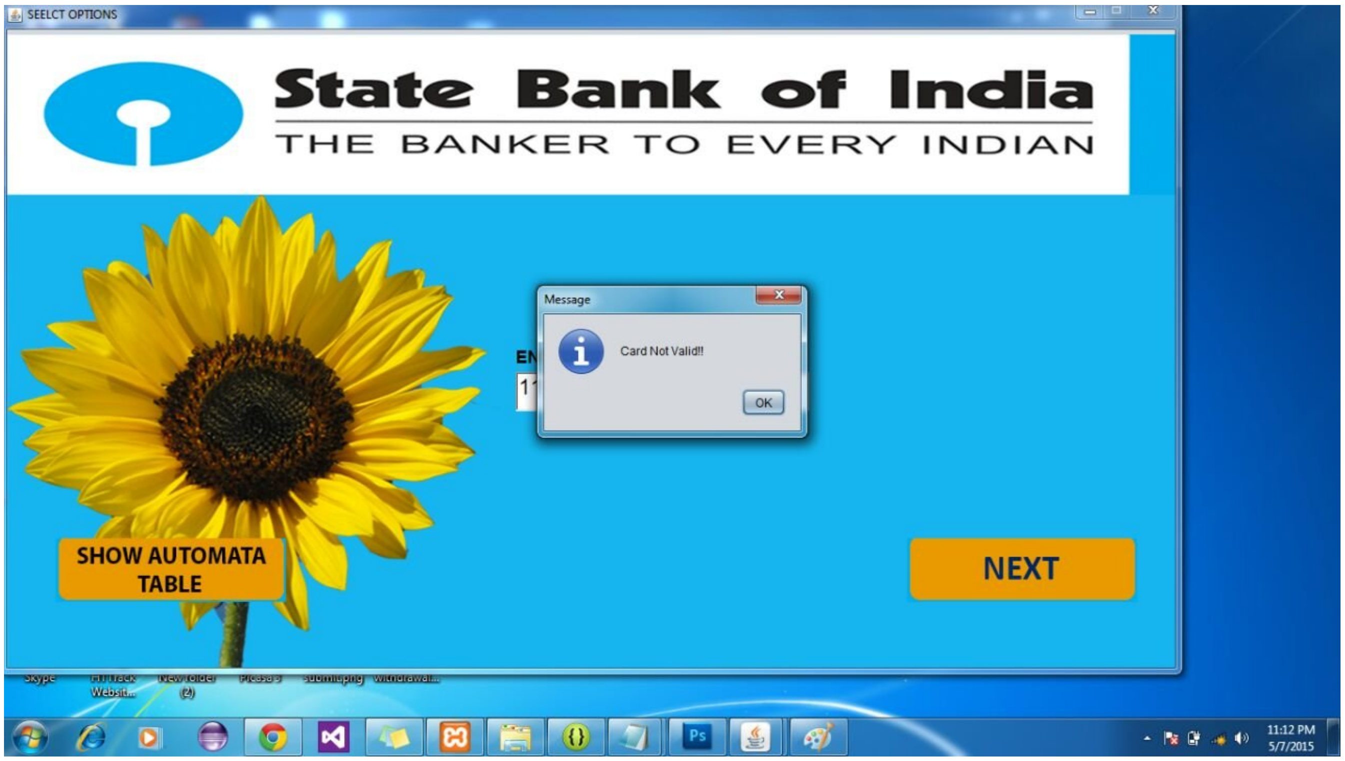

SEELCT OPTIONS
State Bank of India
THE BANKER TO EVERY INDIAN
Message
Card Not Valid!!
OK
SHOW AUTOMATA TABLE
NEXT

Fig.15

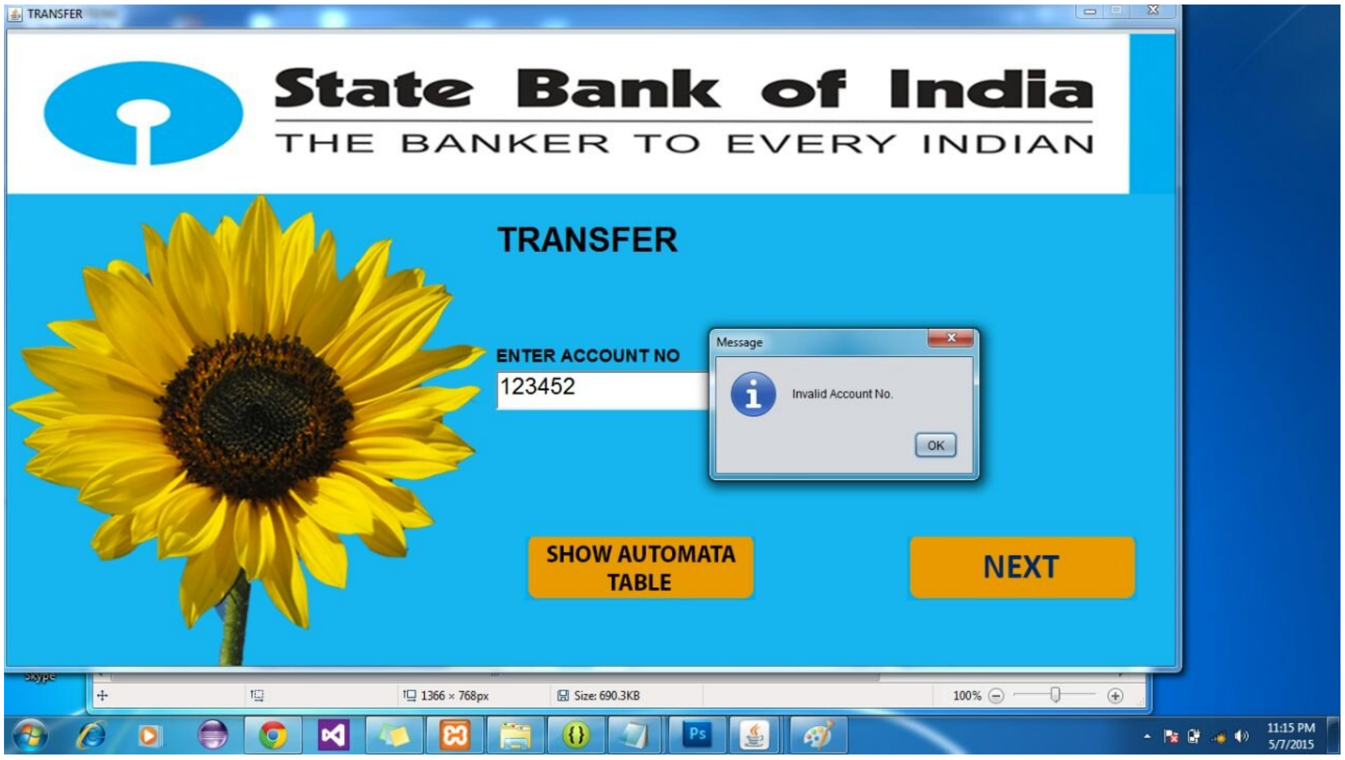
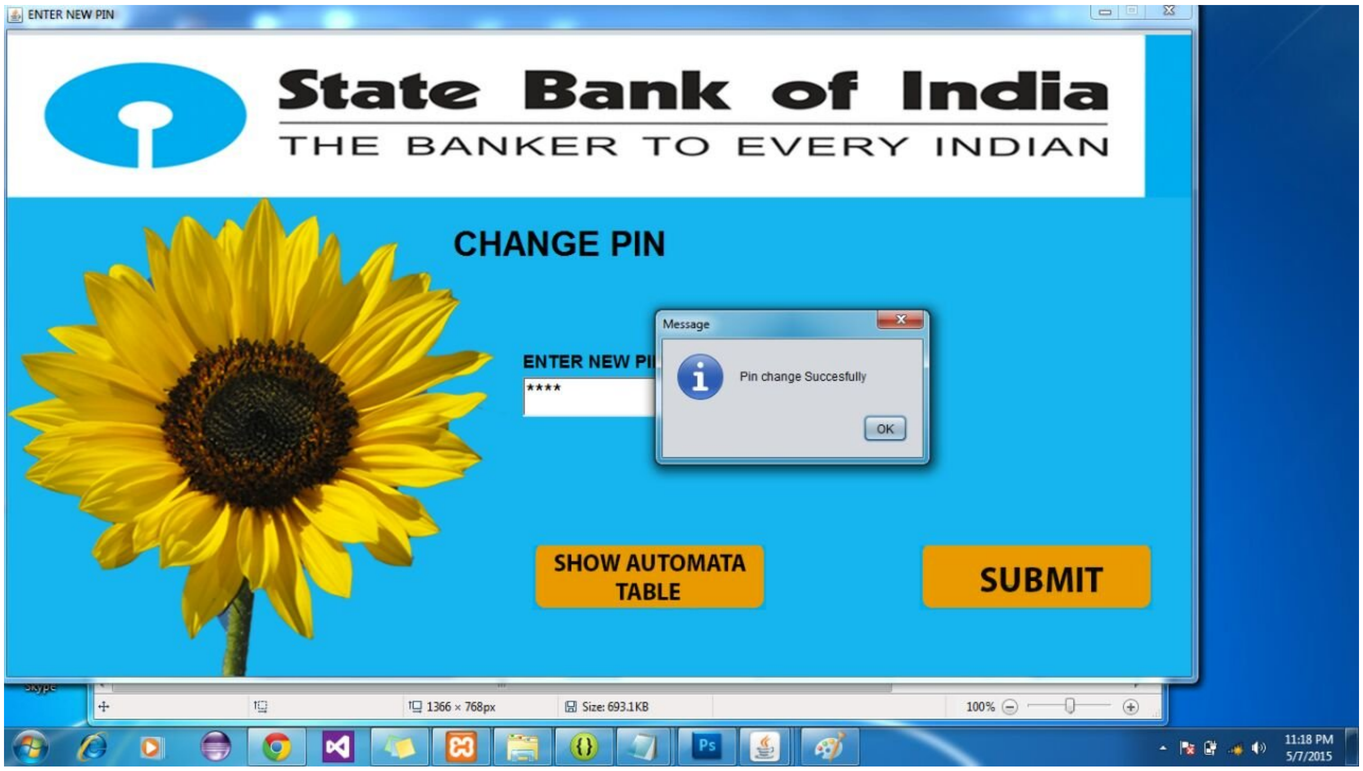
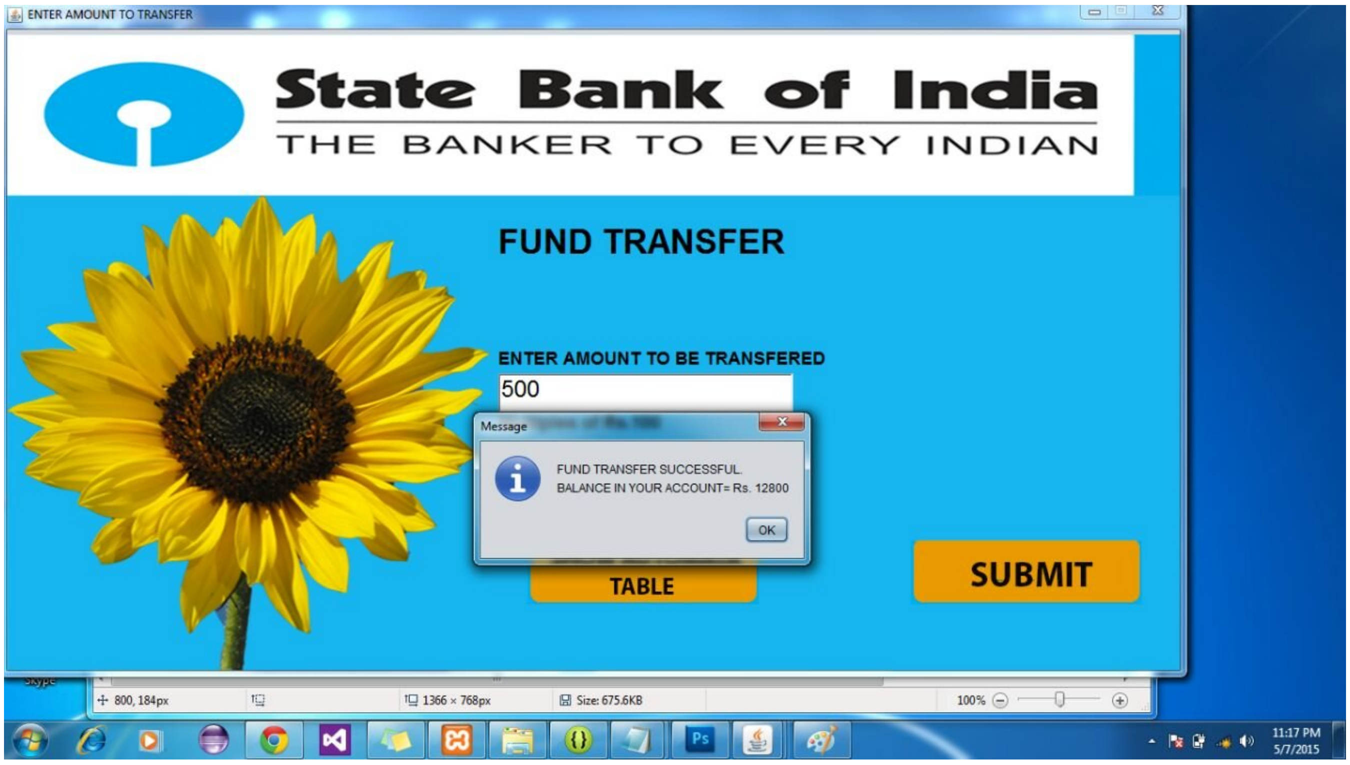

TRANSFER
State Bank of India
THE BANKER TO EVERY INDIAN
TRANSFER
ENTER ACCOUNT NO
123452
Message
Invalid Account No.
OK
SHOW AUTOMATA TABLE
NEXT

Fig.16



Pin change Succesfully

Fig.17



FUND TRANSFER SUCCESSFUL.
BALANCE IN YOUR ACCOUNT= Rs. 12800

Fig.18

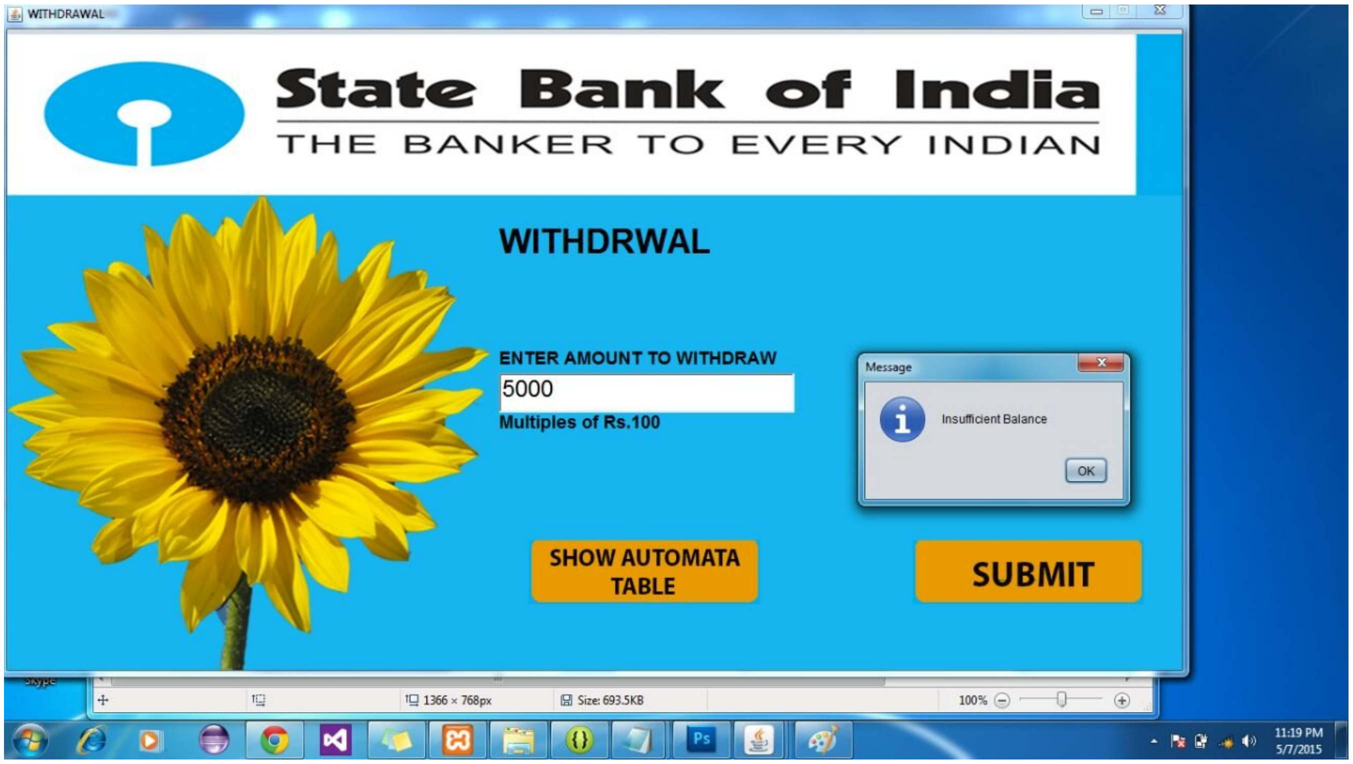

WITHDRAWAL
State Bank of India
THE BANKER TO EVERY INDIAN
WITHDRWAL
ENTER AMOUNT TO WITHDRAW
5000
Multiples of Rs.100
Message
Insufficient Balance
OK
SHOW AUTOMATA TABLE
SUBMIT

Fig.19

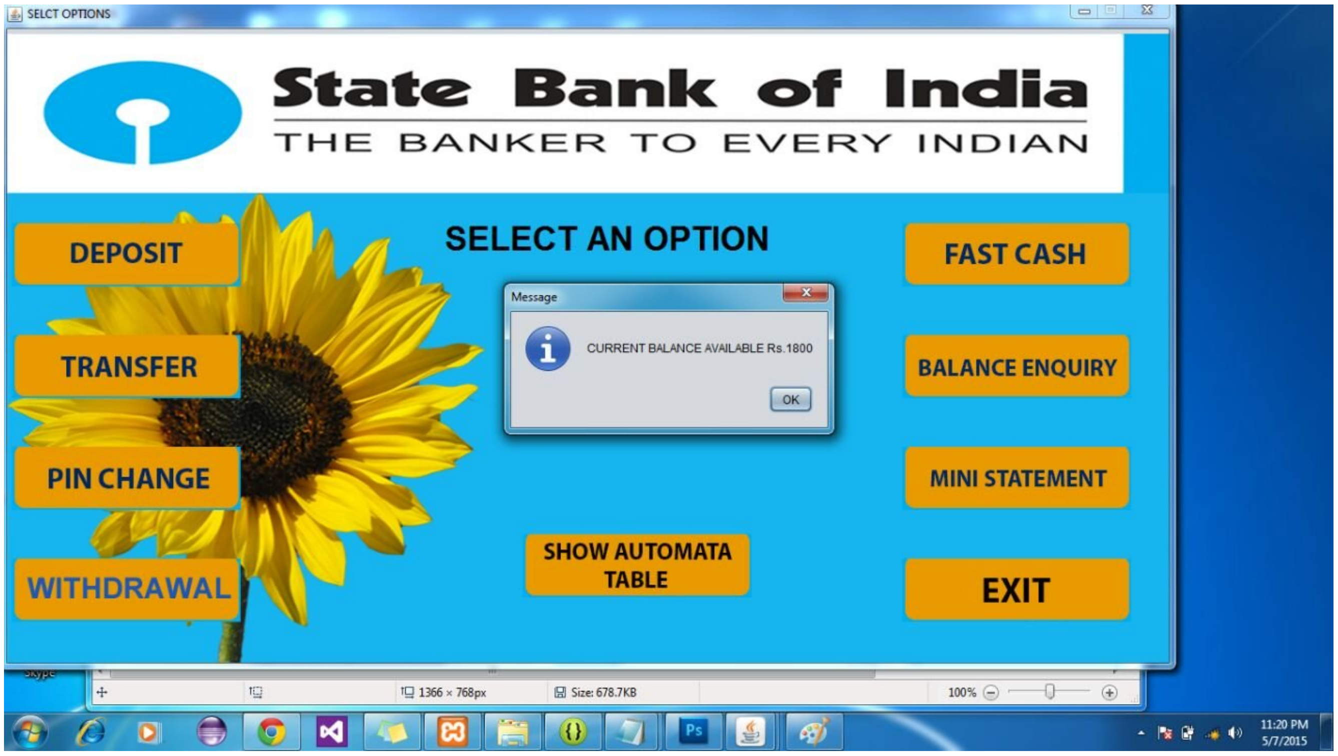

SELCT OPTIONS
State Bank of India
THE BANKER TO EVERY INDIAN
SELECT AN OPTION
DEPOSIT
TRANSFER
PIN CHANGE
WITHDRAWAL
FAST CASH
BALANCE ENQUIRY
MINI STATEMENT
EXIT
Message
CURRENT BALANCE AVAILABLE Rs.1800
OK
SHOW AUTOMATA TABLE

Fig.20

3.5 RELATED WORK

Since E.W. Dijkstra's critics "Program testing can be used to show the presence of bugs, but never to show their absence." [DIJ1], we have almost a religious belief in some part of the academic community that testing should never be used as a verification method [DIJ2,BOYE]. Nevertheless, testing is the most accepted and widely used method in the industrial software development, not only for verification of the software correctness, but also its validation,especially concerning the user requirements (See [BOEH] for the precise definitions of the terms "Verification" and "Validation"). Taking this fact into account, many researchers in Software Engineering started very early to form a mathematical sound fundament for a systematic testing. One of the pioneer works that made Software Testing become a solid discipline is the "Fundamental Test Theorem" of S. Gerhart and J.B. Goodenough which was published 1975: "We prove ... that properly structured tests are capable of demonstrating the absence of errors in a program" [GeGo]. By means of test their selection criteria, based on predicate logic, Gerhart and Good enough found numerous capital errors in a program of P.Naur which was published in a text book and repeatedly quoted by other renowned authors, also as an example for correctness proof. Since the Fundamental Test Theorem, a vast amount of further research work enabled systematic testing to become more and more recognized and also accepted in the academia, e.g. through the work of E.W. Howden: "It is possible to use testing to formally prove the correctness of programs" [HOWD]. Worthwhile to mention is also the work of L. Bouge, who made valuable contributions to the Software Test Theory [BOUG].FSA-based methods and RegEx have been used since almost four decades for specification and testing of software and system behavior, e.g. for Conformance Testing [BOCH, CHOW,MARC, SARI]. Recently, L. White introduced an FSA-based method for GUI testing, including a convincing empirical study to validate his approach [WHIT]. Our work is intended to extend L. White's approach by taking not only desired behavior of the software into account, but also undesired situations. This could be seen as the most important contribution of our present work, i.e. testing GUIs not only through exercising them by means of test cases which show that GUI is working properly under regular circumstances, but exercising also all potentially illegal events to verify that the GUI behaves satisfactory also in exceptional situations.Thus, we have now a holistic view concerning the complete behavior of the system we want to test. Moreover, having an exact terminology and formal methods, we can now precisely scale the test process, justifying the cumulating costs that must be in compliance with the test budget.Beside L. White's pioneer work, another state-oriented approach, based on the traditional method SCR (Software Cost Reduction) is described by C. Heitmeyer et al. in [GARG]. This approach uses model checking to generate test cases, using well known coverage metrics for test case selection. For expressing conditioned events in temporal-logic formulae, the authors propose to use modal-logic abbreviations which requires some skill with this kind of formalism.A different approach for GUI testing has been recently published by A. Memon et al.[MEM1, MEM2], as already mentioned in Section 1. The authors deploy methods of Knowledge Engineering, to generate test cases, test oracles, etc. to handle also the Test Termination Problem. Both approaches, i.e. of A. Memon et al., and C. Heitmeyer et al., use some heuristic

methods to cope with the state explosion problem. We also introduced in the present paper methods for test case selection; moreover we handled test coverage aspects for termination of GUI testing, based on theoretical knowledge that is well-known in Conformance Testing and validated in the practice of protocol validation for decades. We showed that the approach of Dahbura, Aho et al to
handle the Chinese Postman Problem [AHO1, SHEN] in its original version might not be appropriate to handle GUI testing problems, because the complexity of our optimization problem is considerable lower, as summarised.Thus, the results of our work enables efficient algorithms to generate and select test cases in sense of a meaningful criterion, i.e. edge coverage.Converting the FSA into a RegEx enables us to work out the GUI testing problem more comfortable, applying algebraic methods instead of graphical operations. A similar approach was introduced 1979 by R. David and P. Thevenod-Fosse for generating test patterns for sequential circuits using regular expressions [THEV]. Regular expressions have been also proposed for software design and specification [SHAW] which we strongly favor in our approach. The introduced holistic approach, unifying the modeling of both the desired and undesired features of the system to be developed enables the adoption of the concept "Design for Testability" in software design; this concept was initially introduced in the seventies [WILL] for hardware. We hope that further research will enable the adoption of our approach in more recent modeling tools as to State Charts etc. There are,however, some severe theoretical barriers, necessitating further research to make the due extension of the algorithms we developed in the FSA/RegEx environment, mostly caused by the explosion of states when taking concurrency into account.

CONCLUSION AND FUTURE WORK

The need to incorporate GUI testing into testing processes throughout the software life-cycle is becoming apparent.Over the past several years, advances in model-based GUI testing have made this more cost-effective by providing test-adequacy criteria and by automating test-case generation, test execution, test oracles, and regression testing. In the future, GUI testing may commonly be woven into the testing process, with different levels of testing designed to meet goals at different time scales. Lessons learned from GUI testing will likely be applied to the testing of other kinds of event-driven software. Our results have some limitations; that is, they are not universally valid for GUI testing practice, but they provide an experimental insight.Therefore, to obtain more general results in the future,we plan to include other (including non-event-based) GUI testing frameworks, use different test generation algorithms,and increase the number of experiments by also considering different types of GUI-based applications.

References, IEEE Format

Book

[1] Java 2,Complete Reference, *fifth edition*

Research Paper

[2] Fevzi Belli,Mutlu Beyazit,”Event Based GUI testing and reliability assessment techniques”.

[3]Fevzi Belli,University of Paderborn,”Finite State Testing of GUI’s”.

Web References

[4] http://w3schools.org/

[5] http://google.com/

[6] http://en.wikipedia.org/

[7] http://www.slideshare.com/

CODE

```
package atm;


import javax.swing.*;


import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.sql.*;

public class FirstPage extends JFrame implements ActionListener {


    String acc_no = "";
    JLabel l1, l2;
    JButton jb2, jb1;
    TextField tf;
```

```
        Connection con;

        Statement st;

        ResultSet rs;


        FirstPage() {

                try {

                        Class.forName("com.mysql.jdbc.Driver");

                        con = DriverManager

        .getConnection("jdbc:mysql://localhost/?user=root&password=");

                        st = con.createStatement();

                        st.executeUpdate("CREATE DATABASE IF NOT EXISTS
    db");

                        con =
    DriverManager.getConnection("jdbc:mysql://localhost/db",

                                        "root", "");


                        // creating table

                        String table = "create table IF NOT EXISTS atm_detail ("

                                        + "acc_no varchar(50) PRIMARY KEY," +
    "name varchar(50),"
```

```
                        + "atm_no varchar(50)," + "pin varchar(6), " +
"acc_type varchar(7))";
                st = con.createStatement();

                /*
                 * String table= "CREATE TABLE if not exists
REGISTRATION " +
                 * "(id INTEGER not NULL, " + " first VARCHAR(255), " +
                 * " last VARCHAR(255), " + " age INTEGER, " +
                 * " PRIMARY KEY ( id ))";
                 */
                st.executeUpdate(table);
                st = con.createStatement();


                table = "create table IF NOT EXISTS db.balance ("
                        + "acc_no varchar(50) REFERENCES
db.atm_detail(card_no),"
                        + "balance INT( 10 ))";
                st.executeUpdate(table);
```

```
                    st = con.createStatement();
                    table = "create table IF NOT EXISTS db.automata_table ("
                                        + "current_state varchar(60) ,"
                                        + "input varchar(60)," + "next_state
varchar(60) " + ")";
                    st.executeUpdate(table);


          } catch (ClassNotFoundException e) {
                    e.printStackTrace();
          } catch (SQLException e) {
                    e.printStackTrace();
          }
          Container c = this.getContentPane();
          this.setLayout(null);
          JPanel jp = new JPanel();
          jp.setBounds(0, 0, 1200, 680);


          l1 = new JLabel("ENTER ATM NO.");
          l1.setBounds(520, 320, 300, 30);
          Font f = new Font("Arial", Font.BOLD, 18);
          l1.setFont(f);
```

```
c.add(l1);


tf = new TextField(16);

tf.setBounds(520, 350, 300, 40);

Font f3 = new Font("Arial", Font.PLAIN, 22);

tf.setFont(f3);

c.add(tf);


l2 = new JLabel("ATM Card Not Valid");

l2.setBounds(690, 360, 300, 100);

Font f2 = new Font("Arial", Font.BOLD, 14);

l2.setForeground(Color.red);

l2.setFont(f2);

l2.setVisible(false);


c.add(l2);

// l2=new JLabel("");


ImageIcon i = new ImageIcon(

          Constants.path+"sbi1.png");

JLabel jl = new JLabel(i);
```

```
        jl.setBounds(0, 0, 1280, 680);
        jp.add(jl);

        ImageIcon i2 = new ImageIcon(
                Constants.path+"next.png");
        //ImageIcon i2 = new ImageIcon(
        //
    "E:\\adi_new_workspace\\ATM\\src\\atm\\next.png");

        jb1 = new JButton(i2);
        jb1.setBackground(new Color(0, 0, 0, 0));
        jb1.setBounds(920, 520, 235, 65);
        jb1.addActionListener(this);
        c.add(jb1);

        ImageIcon ii = new ImageIcon(
                Constants.path+"automata_table.png");
        jb2 = new JButton(ii);
        jb2.setBackground(new Color(0, 0, 0, 0));
        jb2.setBounds(50, 520, 235, 65);
        jb2.addActionListener(this);
```

```
            c.add(jb2);


            c.add(jp);


        }


        public static void main(String[] args) {
            FirstPage k = new FirstPage();
            k.setTitle("Welcome to SBI Bank ATM");
            k.setSize(1200, 680);
            k.setVisible(true);
            k.setResizable(false);
            try {
                UIManager

        .setLookAndFeel("com.sun.java.swing.plaf.nimbus.NimbusLookAndF
    eel");
            } catch (ClassNotFoundException ex) {

        Logger.getLogger(FirstPage.class.getName()).log(Level.SEVERE, null,
                        ex);
        } catch (InstantiationException ex) {

Logger.getLogger(FirstPage.class.getName()).log(Level.SEVERE, null,
                                ex);
        } catch (IllegalAccessException ex) {

Logger.getLogger(FirstPage.class.getName()).log(Level.SEVERE, null,
                                ex);
        } catch (UnsupportedLookAndFeelException ex) {

Logger.getLogger(FirstPage.class.getName()).log(Level.SEVERE, null,
                                ex);
        }
        k.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}


@Override
public void actionPerformed(ActionEvent e) {
        // TODO Auto-generated method stub
        if(e.getSource()==jb1)
        {
        String atm_no = tf.getText();


        if (atm_no.equals("")) {
                JOptionPane.showMessageDialog(null,
                                "You Cannot leave Field Blank!!\nPlease Enter
Card No");
                insertInTable("FirstPage","Next Pressed+Field
Blank","FirstPage");
                return;
        }


        String query = "Select * from db.atm_detail where atm_no='"
+ atm_no
                        + "'";
        try {
                st = con.createStatement();


                rs = st.executeQuery(query);


                if (!rs.next()) {
                        /*
```

```
                    * l2.setText("Card Not Valid!!"); l2.setVisible(true);
                    */
                    JOptionPane.showMessageDialog(null, "Card Not
Valid!!");
                    insertInTable("FirstPage","Next Pressed+Invalid
Card Entered","FirstPage");

                    tf.setText("");
               } else {
                    acc_no = rs.getString(1);
                         insertInTable("FirstPage","Next Pressed+Valid
Card Entered","Pin");
                         Pin obj = new Pin(acc_no);
                         obj.setVisible(true);
                         obj.setTitle("ENTER PIN");
                         obj.setSize(1200, 680);
                         obj.setResizable(false);

     obj.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                              this.setVisible(false);

               }
```

```
        } catch (SQLException e1) {
                // TODO Auto-generated catch block
                e1.printStackTrace();
        }
        }
        else if(e.getSource()==jb2)
        {
                AutomataTable obj = new AutomataTable();
                obj.setVisible(true);
                obj.setTitle("AUTOMATA TABLE");
                obj.setSize(800, 680);
                obj.setResizable(false);


obj.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
                //this.setVisible(false);


        }
        }
        //insertInTable("FirstPage","Next Pressed+Valid Card
Entered","Pin");
 void insertInTable(String cur,String inp,String next)
 {
        try {
                    st = con.createStatement();
                //first checking if its already present?
                    String sql = "SELECT * FROM automata_table WHERE
current_state='"+cur+"'AND input='"+inp+"' AND next_state='"+next+"'";
                    rs = st.executeQuery(sql);
                    int flag=0;
                    while(rs.next())
                    {
                        flag=1;
                      break;
                    }

    if(flag==0)
    {
     st = con.createStatement();
```

```
            sql = "INSERT INTO automata_table " +
                  "VALUES ('"+cur+"', '"+inp+"', '"+next+"')";
              st.executeUpdate(sql);
               }
              }catch (SQLException e1) {
                  // TODO Auto-generated catch block
                  e1.printStackTrace();
              }
 }
}
package atm;


import java.awt.Color;
import java.awt.Container;
import java.awt.Font;
import java.awt.TextField;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;

public class AutomataTable extends JFrame{

        JButton jb1;
        int balance;
        Connection con = null;
```

```
Statement st;

ResultSet rs;

Object[][] data;



public AutomataTable() {

        //this.setLayout(null);


        //ImageIcon i10 = new ImageIcon(

        //
"E:\\adi_new_workspace\\ATM\\src\\atm\\back.png");

        //jb1 = new JButton(i10);

        //jb1.setBackground(new Color(0, 0, 0, 0));

        //jb1.setBounds(920, 520, 235, 65);

        //jb1.addActionListener(this);

        //add(jb1);


        try {


                Class.forName("com.mysql.jdbc.Driver");
```

```
                    con =
DriverManager.getConnection("jdbc:mysql://localhost/db",

                                "root", "");

                    st = con.createStatement();

      rs = st.executeQuery("Select * from db.automata_table");

                    String s1,s2,s3;

                    int count=0,total=0;

                    while(rs.next())

                    {

                          total++;

                    }

                    st = con.createStatement();

      rs = st.executeQuery("Select * from db.automata_table");

            data = new Object[total][3];


      while(rs.next())

                    {

                          s1=rs.getString(1);

                          s2=rs.getString(2);

                          s3=rs.getString(3);

                          data[count][0]=s1;
```

```
                        data[count][1]=s2;

                        data[count][2]=s3;

                        count++;

                    }

                } catch (ClassNotFoundException e) {

                    e.printStackTrace();

                }


                catch (SQLException e) {

                    e.printStackTrace();

                }

                String[] columns = new String[] {"Source Node", "Input",
"Destination"};


                JTable table = new JTable(data, columns);


              //add the table to the frame

            this.add(new JScrollPane(table));

            this.setTitle("Automata Table");

        //this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //this.pack();
        //this.setVisible(true);


         }
}
package atm;


import java.awt.Color;
import java.awt.Container;
import java.awt.Font;
import java.awt.TextField;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.logging.Level;
import java.util.logging.Logger;


import javax.swing.ImageIcon;
import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JOptionPane;

import javax.swing.JPanel;

import javax.swing.UIManager;

import javax.swing.UnsupportedLookAndFeelException;


public class Menu extends JFrame implements ActionListener{

        String acc_no;

        JButton jb1,jb2,jb3,jb4,jb5,jb6,jb7,jb8,jb9,jb22;

        Connection con = null;

        Statement st;

        ResultSet rs;




        public Menu(String acNo) //throws ClassNotFoundException,
SQLException {

        {

                try

        {
```

```
        Class.forName("com.mysql.jdbc.Driver");
            con =
    DriverManager.getConnection("jdbc:mysql://localhost/db",
                    "root", "");
            st = con.createStatement();
        }
        catch(Exception ee)
        {
        }
            acc_no = acNo;
            Container c = this.getContentPane();
            this.setLayout(null);
            JPanel jp = new JPanel();
            jp.setBounds(0, 0, 1200, 680);



            JLabel lWc = new JLabel("SELECT AN OPTION");
            lWc.setBounds(450, 200, 1000, 30);
            Font f4 = new Font("Arial", Font.BOLD, 35);
            lWc.setFont(f4);
            c.add(lWc);
```

```
ImageIcon i6 = new ImageIcon(
        Constants.path+"deposit.png");
jb5 = new JButton(i6);
jb5.setBackground(new Color(0, 0, 0, 0));
jb5.setBounds(5, 200, 235, 65);
jb5.addActionListener(this);
c.add(jb5);
ImageIcon ii = new ImageIcon(
        Constants.path+"automata_table.png");
jb22 = new JButton(ii);
jb22.setBackground(new Color(0, 0, 0, 0));
jb22.setBounds(530, 520, 235, 65);
jb22.addActionListener(this);
c.add(jb22);



ImageIcon i7 = new ImageIcon(
        Constants.path+"transfer.png");
jb6 = new JButton(i7);
jb6.setBackground(new Color(0, 0, 0, 0));
jb6.setBounds(5, 315, 235, 65);
```

```
jb6.addActionListener(this);

c.add(jb6);



ImageIcon i8 = new ImageIcon(

        Constants.path+"pin_change.png");

jb7 = new JButton(i8);

jb7.setBackground(new Color(0, 0, 0, 0));

jb7.setBounds(5, 430, 235, 65);

jb7.addActionListener(this);

c.add(jb7);



ImageIcon i9 = new ImageIcon(

        Constants.path+"withdrawal.png");

jb8 = new JButton(i9);

jb8.setBackground(new Color(0, 0, 0, 0));

jb8.setBounds(5, 545, 235, 65);

jb8.addActionListener(this);

c.add(jb8);
```

```
ImageIcon i10 = new ImageIcon(
            Constants.path+"fast_cash.png");
jb9 = new JButton(i10);
jb9.setBackground(new Color(0, 0, 0, 0));
jb9.setBounds(920, 200, 235, 65);
jb9.addActionListener(this);
c.add(jb9);


ImageIcon i4 = new ImageIcon(
            Constants.path+"balance_enquiry.png");
jb3 = new JButton(i4);
jb3.setBackground(new Color(0, 0, 0, 0));
jb3.setBounds(920, 315, 235, 65);
jb3.addActionListener(this);
c.add(jb3);


ImageIcon i5 = new ImageIcon(
            Constants.path+"mini_statment.png");
```

```
jb4 = new JButton(i5);
jb4.setBackground(new Color(0, 0, 0, 0));
jb4.setBounds(920, 430, 235, 65);
//jb1.addActionListener(this);
c.add(jb4);



ImageIcon i2 = new ImageIcon(
          Constants.path+"exit.png");
jb1 = new JButton(i2);
jb1.setBackground(new Color(0, 0, 0, 0));
jb1.setBounds(920, 545, 235, 65);
jb1.addActionListener(this);
c.add(jb1);



ImageIcon i = new ImageIcon(Constants.path+"sbi1.png");
JLabel jl = new JLabel(i);
jl.setBounds(0, 0, 1280, 680);
jp.add(jl);
c.add(jp);
```

```
        }
 void insertInTable(String cur,String inp,String next)
         {
                 try {
                              st = con.createStatement();
                          //first checking if its already present?
                              String sql = "SELECT * FROM automata_table
WHERE current_state='"+cur+"'AND input='"+inp+"' AND
next_state='"+next+"'";
                              rs = st.executeQuery(sql);
                              int flag=0;
                              while(rs.next())
                              {
                                      flag=1;
                                break;
                              }

               if(flag==0)
               {
                st = con.createStatement();
                     sql = "INSERT INTO automata_table " +
                        "VALUES ('"+cur+"', '"+inp+"', '"+next+"')";
                    st.executeUpdate(sql);
                     }
                    }catch (SQLException e1) {
                            // TODO Auto-generated catch block
                            e1.printStackTrace();
                    }
 }

@Override
public void actionPerformed(ActionEvent e) {
        // TODO Auto-generated method stub
        if(e.getSource()==jb22)
          {
                  AutomataTable obj = new AutomataTable();
                        obj.setVisible(true);
                        obj.setTitle("AUTOMATA TABLE");
                        obj.setSize(800, 680);
                        obj.setResizable(false);

obj.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
```

```
                    return;
              }
            else if(e.getSource()==jb5)
            {
                insertInTable("Menu","Deposit Button
Pressed","Deposit");
                Deposit obj = new Deposit(acc_no);
                obj.setVisible(true);
                obj.setTitle("SELECT OPTIONS");
                obj.setSize(1200, 680);
                obj.setResizable(false);

                obj.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setVisible(false);

            }
            else if(e.getSource()==jb7)
            {
                insertInTable("Menu","Deposit Button
Pressed","Deposit");
                PinChange obj = new PinChange(acc_no);
```

```
                    obj.setVisible(true);
                    obj.setTitle("PIN CHANGE");
                    obj.setSize(1200, 680);
                    obj.setResizable(false);

                    obj.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    this.setVisible(false);

                }
                else if(e.getSource()==jb1)
                {
                    insertInTable("Menu","EXIT Button 
Pressed","FirstPage");
                    FirstPage obj = new FirstPage();
                    obj.setVisible(true);
                    obj.setTitle("SEELCT OPTIONS");
                    obj.setSize(1200, 680);
                    obj.setResizable(false);

                    obj.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    this.setVisible(false);
```

```
        }
        else if(e.getSource()==jb8)
        {
                insertInTable("Menu","EXIT Button
Pressed","FirstPage");
                Withdrawal obj = new Withdrawal(acc_no);
                obj.setVisible(true);
                obj.setTitle("WITHDRAWAL");
                obj.setSize(1200, 680);
                obj.setResizable(false);
                obj.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setVisible(false);


        }


        else if(e.getSource()==jb6)
        {
                insertInTable("Menu","Transfer Button
Pressed","Transfer");
                Transfer obj = new Transfer(acc_no);
                obj.setVisible(true);
                obj.setTitle("TRANSFER");
                obj.setSize(1200, 680);
                obj.setResizable(false);
                obj.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setVisible(false);
            }
            else if(e.getSource()==jb9)
            {
                insertInTable("Menu","Transfer Button
Pressed","Transfer");
                FastCash obj = new FastCash(acc_no);
                obj.setVisible(true);
                obj.setTitle("FAST CASH");
                obj.setSize(1200, 680);
                obj.setResizable(false);
                obj.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setVisible(false);
            }
            else if(e.getSource()==jb3)
            {
```

```
                        try{
                                Class.forName("com.mysql.jdbc.Driver");
                                con = DriverManager.getConnection(
                                                "jdbc:mysql://localhost/db", "root", "");
                                st = con.createStatement();
                                rs = st.executeQuery("Select * from db.balance
where acc_no='"+ acc_no + "'");
                                rs.next();
                                int balance = rs.getInt(2);
                                        JOptionPane.showMessageDialog(null,
                                                        "CURRENT BALANCE AVAILABLE
Rs."+balance);
                        }
                        catch (Exception te) {
                                // TODO: handle exception
                        }


                }
        }
}
```