Designing of Testing Framework through Finite State Automata for Object-Oriented Systems using UML

Sadhana Verma
Student
Computer Science and Engineering
PSIT, Kanpur, India

Ajay Pratap, PhD
Associate Professor
Computer Science and Engineering
PSIT, Kanpur, India

ABSTRACT
Many researches to testing object-oriented systems (OOSs) have been proposed for the past decade. After all, almost all large OO software specifications still contain incompleteness, inconsistency, and ambiguity. The framework can be defined using any state-based specification notation and used to derive test cases from state-based specifications in this paper, it is demonstrated using the RSML notation. A state transition diagram (STD) derived from RSML specification provides a complete behavior of a given OOS. System testing is concerned with testing an entire system based on its specifications. In the context of object-oriented, UML development, this means that system test requirements are derived from UML analysis artifacts such as use cases, their corresponding sequence and collaboration diagrams, class diagrams. The goal is here to support the derivation of functional system test requirements, which will be transformed into test cases once we have detailed design information. In this paper, we describe a methodology in a practical way and illustrate it with an example. In this paper a framework that formally defines test data sets and their relation to the operations in a specification and to other test data sets, providing structure to the testing process.

Keywords
Object-Oriented Program; software testing; software complexity; Finite automaton; UM

1. INTRODUCTION
Testing plays a vital role in software development. Testing is a practical means of detecting program errors that can be highly effective if performed rigorously. Despite the major limitation of testing that it can only show the presence of errors and never their absence, it will always be a necessary verification technique.

An object-oriented system is composed objects. The behavior of the system results from the collaboration of those objects. The objects are the real world entities that exist around us and the basic concepts like abstraction, encapsulation, inheritance, polymorphism all can be represented using UML.

So UML is powerful enough to represent all the concepts exists in object oriented analysis and design. UML diagrams are representation of object oriented concepts only. So before learning UML, it becomes important to understand OO concepts in details.

Another important aspect is automation. Large systems are inherently complex to test and require, regardless of the test strategy, large numbers of test cases. If a system testing method requires the tester to perform frequent, complex manual tasks, then such a method is not likely to be usable in a context where time to market is tight and qualified personnel is scarce. Therefore, the potential for automation of a test methodology is an important criterion to consider (Section VI).

A software requirements specification should be a comprehensive statement of a software system's intended behavior. Unfortunately, requirements specifications are often incomplete, inconsistent, and ambiguous. To provide analysis procedures to find errors in specifications, it is first necessary to determine the desirable properties of a Specification. Previously, defined formal criteria for requirements completeness, consistency and safety.

This paper defines the formal semantics of RSML and describes an automated approach to analyzing an RSML specification for two qualities: 1) completeness with respect to a set of test cases (a response is specified for every possible input and input sequence) and 2) consistency (the specification is free from conflicting requirements and undesired nondeterminism).

2. LITERATURE SURVEY
J. E. Hopcroft and R. M. Karp [1] described an algorithm for determining if two finite automata with start states are equivalent, the asymptotic running time of the algorithm is bounded by a constant times the product of the number of states of the larger automata with the size of the input alphabet.

A. J. Offutt and A. Abdurazik [2**]** presented a novel technique that adapts pre-defined state-based specification test data generation criteria to generate test cases from UML statecharts.

Vipin Saxena and Ajay Pratap [3**]** given a real case study of Indian Postal Services is discussed to solve the problem of learning and object oriented data classification. Naïve Bayesian classifier is a classification algorithm based on Bayes theorem which is guided by genetic algorithm.

D. Harel [4] presented a broad extension of the conventional formalism of state machines and state diagrams that is relevant to the specification and design of complex discrete-event systems. Statecharts can be used either as a stand-alone behavioural description or as part of a more general design methodology that deals also with the system's other aspects, such as functional decomposition and data-flow specification.

A. Andrews et al [5] given a technique for testing executable forms of UML (Unified Modelling Language) models is described and test adequacy criteria based on UML model elements are proposed.

H. Y. Chen et al [6] proposed in this article a methodology to integrate the black- and white-box techniques. The black-box technique is used to select test cases.

Chi-Ming Chung et al [7] proposed a technique and shows that inheritance has a close relation to object-oriented software complexity and reveals that overuse of repeated

(multiple) inheritance will increase software complexity and be prone to implicit software errors.

E. Weyuker et al [8] presented a family of strategies for automatically generating test data for any implementation intended to satisfy a given specification that is a Boolean formula. The fault detection effectiveness of these strategies is investigated both analytically and empirically, and the costs, assessed in terms of test set size, are compared.

3. PROBLEM IDENTIFICATION AND ANALYSIS

In recent years object-oriented paradigm is gaining acceptance for developing large and complex software. OO paradigm has moved into mainstream software development industry. Basically, object-oriented analysis (OOA) and object-oriented design (OOD) methodologies examine problem in the real world and facilitate in decomposing the problem in terms of classes, and some relationships between classes. However, almost all large OO software specifications still contains incompleteness, inconsistentency, and ambiguity [2]. So far two related approaches to verifying the consistency and completeness of procedure-oriented programs include methods based on formal proof systems and static analysis technique such as model checking.

These two approaches have drawbacks and are not suitable for OO software specification.

3.1 Formal Proof Systems

Formal proof systems can be powerful tools in the verification of critical properties of algorithm [13]. Unfortunately, the languages used in the theorem proving approach, such as process algebras and higher order logics, are not understandable by the non-software professionals. Also, formal proofs are notoriously difficult to derive, and these approaches may not be practical for complex systems.

3.2 Model Checking

Model checking is conceptually simple and is applicable in a wide variety of languages and application areas [14]. Consequently, the approach suffered from state-space explosion problems.

This approach is to build a testing framework based on finite automata to test the OO software specification. This approach differs from formal proofs and model checking in that it performs the testing directly on an executable finite automata without manually deriving the formal proofs or generating a great deal of state spaces.

4. OBJECT-ORIENTED REPRESENTATION OF SYSTEM USING UML: A CASE STUDY OF ATM SYSTEM

4.1 UML Use Case Diagram

The use case is a piece of functionality that the system can provide by interacting with the actors. A use case diagram consists of actors and the use cases. The actors are the direct external user of the system.

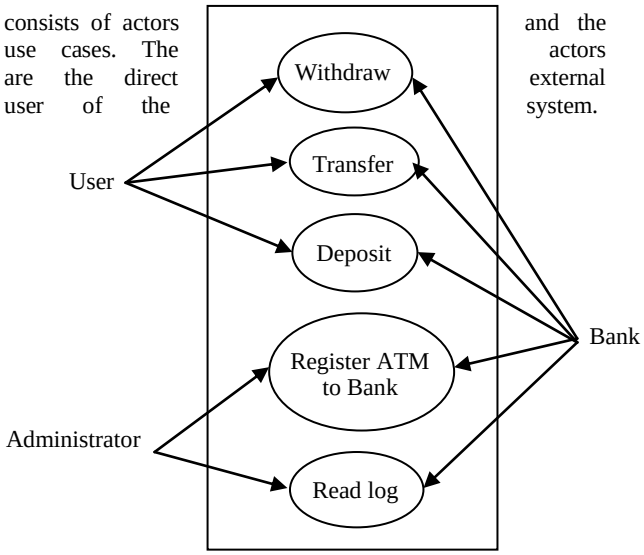


Fig 1: UML Use Case diagram of ATM system

Fig. 1 shows the use case diagram of the ATM system. The interaction between the two actors named as Customer, Administrator and Bank are shown with the five use cases performed by them. Customer sends the request to the ATM system to open account and interacts with the actor Bank and Administrator.

4.2 UML Class Diagram

The UML class diagram shows the static structural behavior of the system, in which attributes and operations are designed for the complete system. The classes can be related to each other in number of ways, like they can be associated, dependent, specialized or packaged.

Fig. 2 depicts the class diagram of the ATM system. The class diagram has five persistent classes, which are Bank, Customer, ATM system, Account, and Transaction. These classes are connected to each other by various relationships with their multiplicities as shown in the diagram. Customer, ATM system, Account, and Transaction are subclasses and they are inherited from non-abstract class Bank.

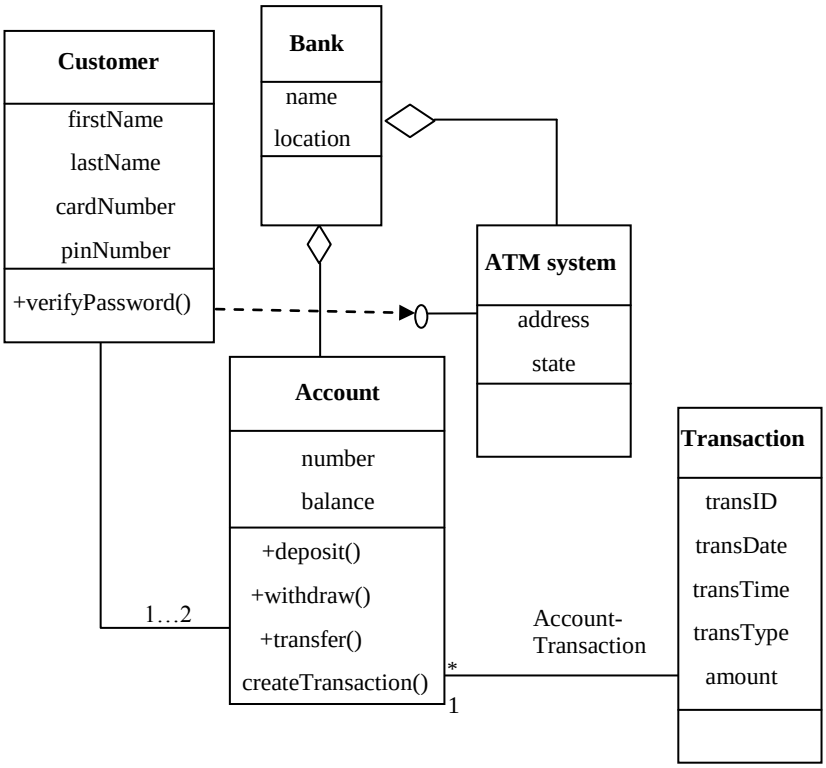


Fig 2: UML Class Diagram of ATM system

4.3 UML Sequence Diagram

Sequence diagram tells how states interact with each other i.e. how transitions are being send and receive between states. This diagram has two vertices: Each system states are represented by the vertical line. The horizontal axis shows the input symbols, which are being sent from the start state to the next state.

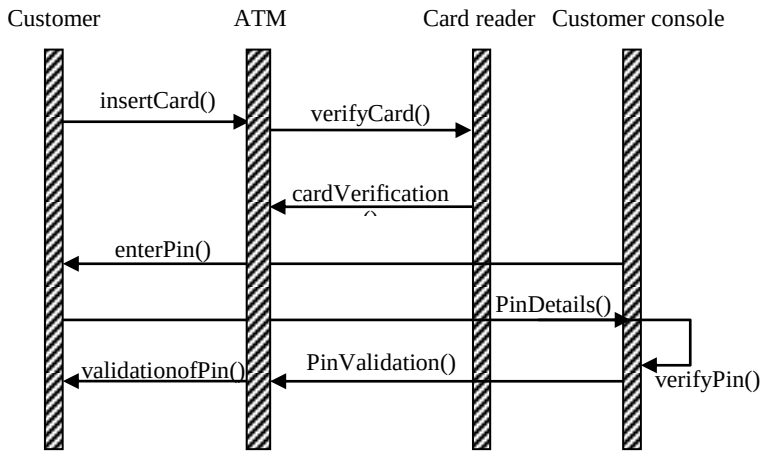


Fig 3: UML Sequence Diagram of ATM system

The sequence diagram for the ATM system is shown in Fig. 3. The sequence diagram presented in Fig. 3 shows that the object of the start state sends request for going to the next state according to their attributes. If the request is valid then the object goes to the next state.

4.4 UML State Diagram

UML state diagram is a graph whose nodes are states and directed arcs are the transitions between them. They are used to describe the behaviour of a system. It describes all of the possible states of an object as events occur.

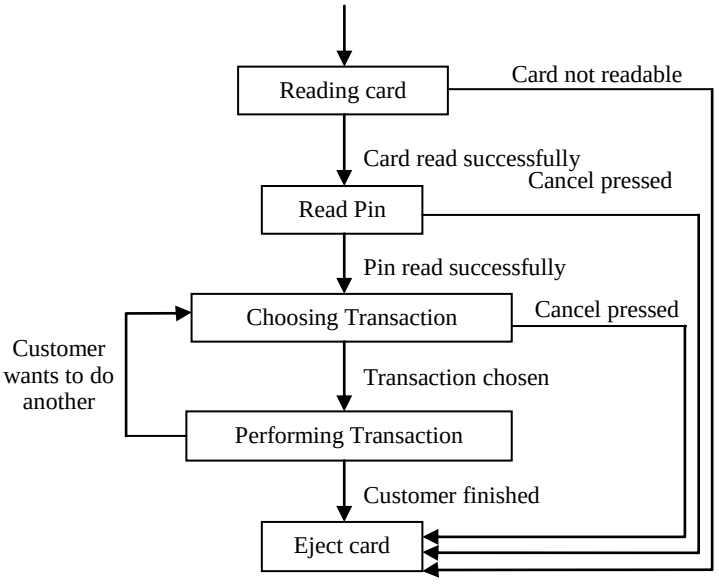


Fig 4: UML State Diagram of ATM system

UML – State diagrams can be used in multiple situations of software development for example they can be used:

- To describe the behavior of a system

- To model the dynamic nature of a software.

5. DESIGNING OF TESTING FRAMEWORK

5.1 Need of Testing

Testing is the process of executing a program with the intention of finding errors." – Myers

"Testing can show the presence of bugs but never their absence." - Dijkstra

The need of testing is to find out the errors in the application. Testing is a merry-go-round process which includes a good amount of time along with cost, for all. But the reality is quite opposite, without testing it is not possible to deliver projects successfully, as during software development, developers make many mistakes throughout the different phase of development and testing helps in correcting those mistakes.

For example, in the requirement engineering stage, the SRS (System Requirement Specification) document is written and tested to check whether it captures all the user requirements or not. The same is applicable for object oriented testing as object-oriented programming increases software reusability, extensibility, interoperability, and reliability and at the same time it is necessary to realize these benefits by uncovering as many programming errors as possible.

5.1 Testing Framework

The object-oriented testing (OOT) proposed four components:(1) describe the specification with a state-based requirement specification language, (2) describe the dynamic behavior of the OO specification with an extended state transition diagram, (3) generate test data in the form of regular expression (4) design a testing algorithm based on finite automata to test the OO specification. The RSML specification provides a means for the tester to retrieve implementation information without going to details by Jaffe et al [10]. On the other hand, the test data derived from RSML specifications can be defined as a sequence of operations (method invocations) along with expected effects; it can be represented by a sequence of events or states. A test data generation criteria called method invocation sequence scenario (MISS) is proposed to documents the correct causal order in which the methods of a class specification can be invoked.

Fig. 5 shows the full picture of this object-oriented testing framework.

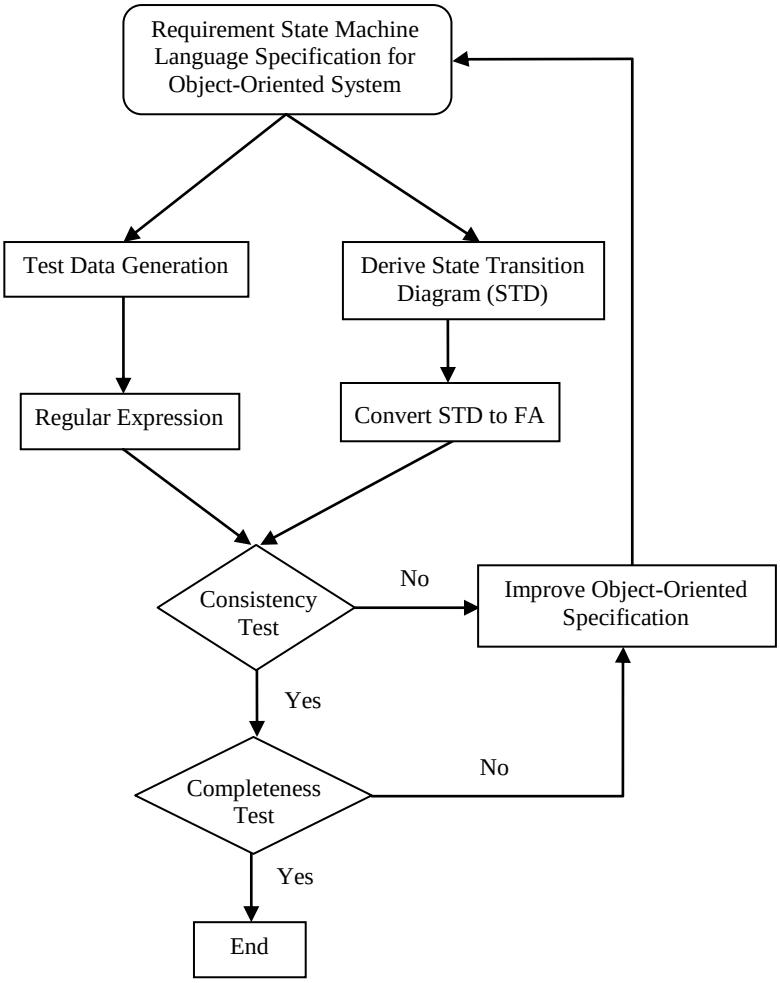


Fig 5: Testing Framework

5.2 RSML (Requirement State Machine Language)

RSML is a state-based requirements specification language suitable to describe complex software specifications. The RSML was developed by the Irvine Safety Research Group using a real aircraft collision-avoidance system called TCAS II (Traffic alert and Collision Avoidance System II) as a testbed by Leveson et al [9]. In addition, RSML has some unique syntactic and semantic features that were developed to enhance readability, reviewability, and analyzability and our ability to handle complex systems.

RSML includes several features developed by Hare1 for Statecharts: superstates, AND decomposition, broadcast communication, and conditional connectives.

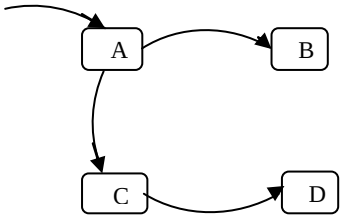


Fig 6: A basic state machine

In RSML (and Statecharts), a state that contains nested states is called a uperstates, and its nested states are called substate. A state that cannot be decomposed further is called a leafstate. In Fig. 7, state R and state T may be grouped into superstate S.

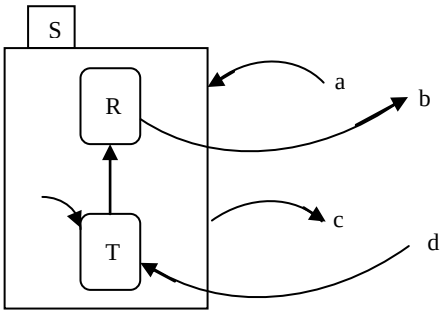


Fig 7: A super state example

6. TESTING METHODOLOGY BASED ON FINITE AUTOMATA

6.1 Testing Criteria and Testing Methodology

Object-oriented program is an unstructured diagram, its dynamic program behavior such as, message passing and state transition by method invocation, make the testing work more complicate than structured software. In this section, two testing criteria, completeness and consistency, are introduced to compare object-oriented software system against its specification.

6.1.1 Incompleteness

Incompleteness determines the extent to which a software system is solving the correct problem by Zualkernan and Tsai [12]. Completeness determines the extent to which a software system is solving the correct program. The completeness of a software system with respect to the problem specification is a measure of the portion of the specification implemented in the system. A system is complete if it covers all the problems indicated in the problem specification.

Definition 1: A software specification is complete if it covers all the problems indicated in the problem requirement.

In the software development lifecycle, testers usually generate test data based on its problem requirement to test the software specification. Therefore, if the test data, supposedly denoted as L_{test} is fully derived from the problem requirement, then the software specification L_{spec} is complete if $L_{spec} \subseteq L_{test}$ holds. Fig. 8 depict three testing relationships between L_{spec} and L_{test}. Fig. 8-a indicates that the software specification L_{spec} is complete and consistent with Ltest because $L_{spec} \subseteq L_{test}$ holds. Fig.8-b means that L_{spec} is partially complete and consistent with L_{test}. Fig.8-c shows that L_{spec} is not complete and consistent with L_{test} because the intersection of L_{spec} and L_{test}, denoted as $L_{spec} \cap L_{req}$, is a empty set ($L_{spec} \cap L_{test} = \phi$). Obviously $L_{spec} \cap L_{test}$ is the complete part of the software specification. In contrast, the incomplete part can be derived from the expression $L_{spec} - (L_{spec} \cap L_{req})$ which is equivalent to $L_{spec} \cap (L_{spec} \cap L_{test})$. However, the difficulties lie in how to exactly figure out the two expressions: $L_{spec} \cap L_{test}$ and $L_{spec} - (L_{spec} \cap L_{test})$.

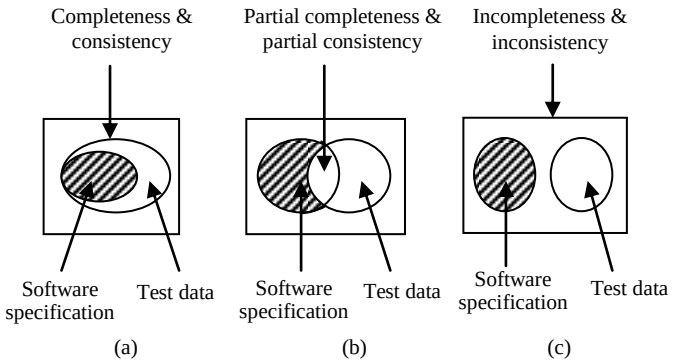


Fig 8: Three testing relationships between L_{spec} and L_{test}

6.1.2 Testing Methodology Modelled by Regular Expression

There are many operations on languages that preserve regular expressions, in the sense that the operation applied to regular expression result in regular expressions. For example, the union two regular expressions is a regular expression, since if r1 and r2 are regular expressions denoting regular sets L_1 and L_2, then $r_1 + r_2$ denotes $L_1 \cup L_2$, so $L_1 \cup L_2$ is also regular. Similarly, the concatenation of regular sets is a regular set and the kleene closure of a regular set is regular.

Property 1: The regular expressions are closed under union, concatenation, and Kleene closure.

Property 2: The class of regular expression are closed under complementation. That is, if L is a regular expression and $L \subseteq \Sigma^*$, then $\Sigma^* - L$ is a regular expression.

Property 3: The regular expressions are closed under intersection.

Property 4: The set of sentences accepted by a finite automaton M with n states is: empty if and only if the finite automaton does not accept a sentence of length less than *n*.

Property 5: For two DFA's M_1, and M_2, $\exists$ an algorithm to determine if $L(M_1) = L(M_2)$.

Theorem 1: For two DFA's M_1 and M_2, $\exists$ an algorithm to determine if $L(M_1) \subseteq L(M_2)$

Proof: (1) To prove if $L(M_1) \subseteq L(M_2)$ holds, if $L(M_1) = L(M_2)$ holds first. Let M_1 and M_2 be FA accepting L_1 and L_2, respectively. By property 1, property 2, property 3, $(L_1 \cap (L_2)^c) \cup ((L_1)^c \cap L_2)$ is accepted by some finite automata, M3. It is easy to see that M_3 accepts a word if and only if $L_1 \neq L_2$. In other words, M_3 does not accept a sentence iff $(L_1 \cap (L_2)^c) \cup ((L_1)^c \cap L_2) = \phi$, hence by property 4, there is an algorithm to determine if $L(M_1) = L(M_2)$.

(2) If $L_1 \subseteq L_2$ holds, then it implies that $L_1 = L_1 \cap L_2$. Let $L(M_3) = L_3 = L_1 \cap L_2$. Then, $L_1 = L_1 \cap L_2$ is equal to $L_1 = L_3$. By the result of (1), $L_1 = L_3$ iff $(L_1 \cap L_3) \cup (L_1 \cap L_3) = \phi$. Therefore, it implies that $L_1 \subseteq L_2$ iff $(L_1 \cap (L_1 \cap L_2)^c) \cup ((L_1)^c \cap (L_1 \cap L_2)) = \phi$.

This algorithm is easily designed by constructing a FA which accepts the language $(L_1 \cap (L_1 \cap L_2)^c) \cup ((L_1)^c \cap (L_1 \cap L_2)) = \phi$.

By Theorem 1, given an object-oriented programming L_{spec} (M), and a testing data L_{test} (M) generated from formal

specification, build a FA to verify L_{spec} (M) ⊆ L_{test} (M). Obviously, the testing work is to construct a DFA to accept

$(L_{spec} \cap (L_{spec} \cap L_{test})^c) \cup ((L_{spec})^c \cap (L_{spec} \cap L_{test})) = \phi$

Fortunately, there have been three primitive regular expression properties to help construct the FA by Ullman and Hopcroft [11].

6.1.3 Inconsistency

Consistency is the extent to which a software system, its problem specification, and its solution specification do not have internal contradictions.

Definition 2: A specification is consistent if its solution specification does not conflict with problem specification. Basically, there are two types of inconsistent problems which may occur in a state-transition based specification

Definition 3: Given an OO specification M = (Σ, Q, q_0, δ, F), A state q_m is said to be reachable from state q_n if there exists a path from q_n to q_m. q_m is reachable from state q_n iff ∃ s: δ (q_n . s) = q where s ∈ Σ*.

If a state is unreachable there are two possibilities. Either the state has no function and can be eliminated from the specification, or the state should be reachable and the requirements document must be modified accordingly.

6.1.3 Test Data Generation from RSML Specification

In the following, we propose a criterion called method invokation sequence scenario (MISS) as a guide to generate the test data from the RSML specification. The MISS of a state documents the correct casual order in which the methods of a state can be invoked. The MISS represents the method interactions, i.e. the dynamic interaction of classes. To describe test data generation precisely, make some definitions to describe the MISS.

Definition 5: Methods_of (S): Given a state S, Methods_of (S) is defined as a set of all methods defined in state S.

For example: Given a state Stack, the methods of Stack can be: Method_of (Stack) = {push, pop, check_empty, check_full}

To represent the method invokation sequence scenario of a state S, use regular expression over the alphabet (denoted as Σ) consisting of methods from Method_of (S). The regular definition is a sequence of definitions of form:

$$\bigcup_{i=1}^{n} [S_i] \rightarrow m_i . [S_i]$$

where each Si is a distinct label, $S_j \in \{ S_1, S_2, \ldots, S_n\}$, and method $m_i \in U^n_{i=1}$ Method _of (S_i) . This form means that after schema S_i invokes some method mi then state of state S_i will transfer to the state S_j. Note that m_i is a terminal symbol while $[S_i]$ is a nonterminal symbol which can derive new regular definitions until some terminal symbol is met.

Definition 6: Method Sequence: Given a state S with Method_of (S) = {m_0, m_1, …, m_n}, a method sequence S is defined as a finite lengthy sequence of methods over Method_of (S) which corresponds to a causal order in which the methods get involved.

For example: In the case of Account state a possible method sequence is "open • deposit • withdraw• close". In this method sequence, deposit method is invoked before the withdraw method is invoked.

Definition 7: Method Invokation Sequence Scenario, MISS(S): A MISS(S) is the whole invokation sequence set which define the relationships between all the methods in Method_of(S).

For example: In the case of Account state, the test data MISS (Account) is: open • deposit • (deposit | transfer | withdraw)* • close.

The MISS (Account) consists of four states with four types of events each corresponding to messages. The method open creates an Open Account state. The deposit method creates an actual Account Operation state. Once a deposit is chosen, deposit, transfer and withdraw methods can be invoked until the transaction is end.

7. IMPLEMENTATION

7.1 Case Study: An ATM System

This section uses an automated teller machine (ATM) as an example to illustrate how the testing methodology works out. The discussion of the ATM example will concentrate on the functionality of the software system.

7.1.1 The RSML Specification for ATM System

The specifications are established through communication with the customer. The RSML specification is derived based on the original description and an object-oriented analysis (OOA). OOA is used to define objects and their operations. During the formulation of the specification, ambiguities, incompleteness or errors may be found in the original requirements description or the analysis phase, and several iterations may be necessary.

The specification of the ATM system is given in Fig. 9. An ATM card has a unique card number, which is associated with to a personal identification number (PIN). The PIN is specified by the bank's customer when the card is issued. Both card number and PIN are stored in the bank computer system. The customer inserts the card into an ATM, which read the card number and prompts the customer for the PIN that is associated with this card. Meanwhile, the customer enters his/her PIN. If the two match, the customer is given a menu of transaction choices. The customer enter his/her request, which is verified by the ATM by communication with the bank computer. After the request is verified, the transaction will go on. This ATM system specification has six major states where there are two superstates: state verify_PIN and state Transaction.

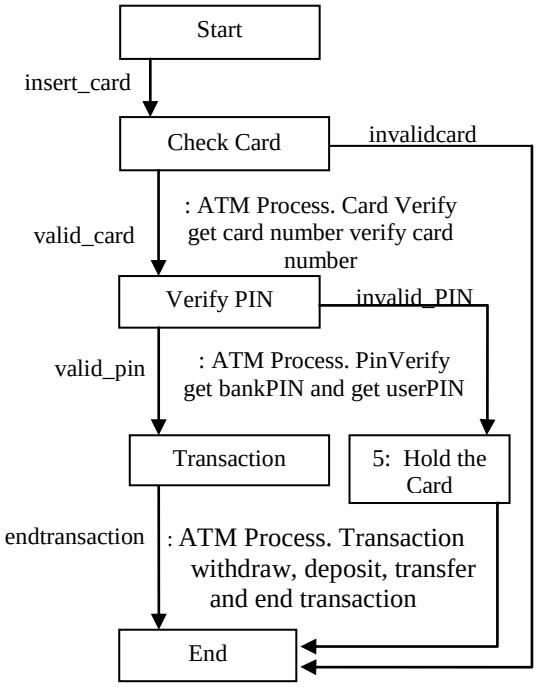


Fig 9: The RSML specification of ATM system

After the RSML specification is done, the requirements and OOA will conform to the specification. Implementation, testing and maintenance will all benefit from the effort put into the specification.

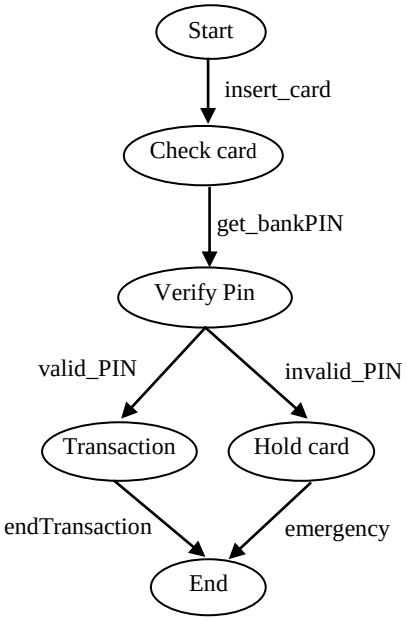


Fig 10: Convert the RSML specification to a corresponding finite automata

Here only present the test data of state 4 (Transaction state). The Transaction state is invoked once the PIN is verified. In the Transaction state, there are four leafstates: Open_Account, Account_Operation, Loan_Account and Eject_Card. Each state associates with a set of methods which causes the transitions from one state to another state.

7.1.2 Test Result Analysis

7.1.2.1 Construction of Testing Cases

Complement

Given an automata denoted L (M) = M = (Σ, Q, q_0, δ, F) the complement is $\Sigma^* - L$. To accept $\Sigma^* - L$, complement the final states of M. That is L (M) = M = (Σ, Q, q_0, δ, Q − F). Then M accepts a word w if and only if δ (q_0, w) is in Q-F, that is, w is in $\Sigma^* - L$.

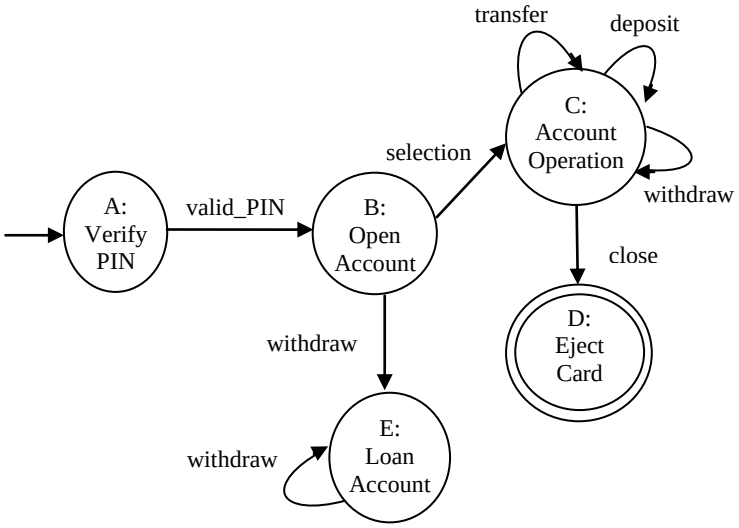
Union: Given two automata L (M_1) = (Σ_1, Q_1, q_1, δ_1, F_1) and L (M_2) = (Σ_2, Q_2, q_2, δ_2, F_2) the union of M_1 and M_2 denoted $M_1 \cup M_2$ is easily done by union the each element in 5-tuple. Then L (M_1) ∪ L (M_2) = $M_{1 \cup 2}$ ($Q_1 + Q_2$, $\Sigma_1 + \Sigma_2$, $\delta_1 + \delta_2$, $q_1 + q_2$, $F_1 + F_2$).

Intersection: The construction of intersection involves taking the Cartesian product of states, and we sketch the construction as follows, Let L (M_1) = (Σ_1, Q_1, q_1, δ_1, F_1) and L (M_2) = (Σ_2, Q_2, q_2, δ_2, F_2) be two deterministic finite automata. Let M = ($Q_1 \times Q_2$, Σ, δ, [q_1, q_2], $F_1 \times F_2$) where for all Specification p_1 in Q_1, p_2 in Q_2 and a in Σ. It is easily shown that L (M) = L (M_1) ∩ L (M_2).

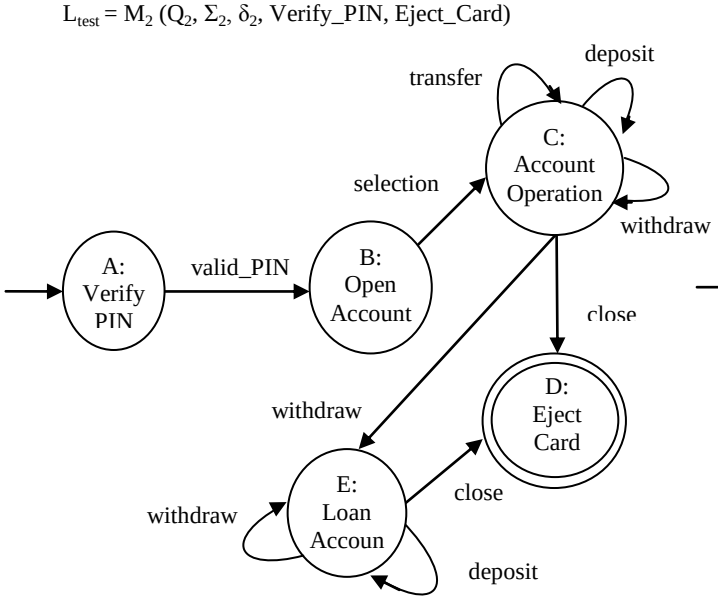
7.1.2.2 Completeness Analysis

Now use the Transaction state to illustrate how the testing methodology works out. The test data of Transaction state in the form of regular expression has been generated in above. Both of them can be represented by two corresponding finite automata in Fig. 11. The RSML specification is denoted as L_{spec}, and the test data is denoted as L_{test}.

$L_{spec} = M_1 (Q_1, \Sigma_1, \delta_1,$ Verify_PIN, Eject_Card)



(a) RSML specification of Transaction



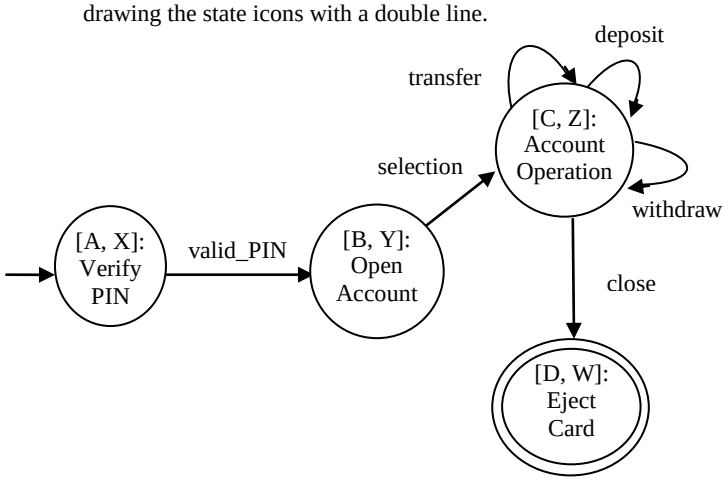
(b) Test Data MISS of Transaction

Fig 11: An illustration of the testing framework: using Transaction state

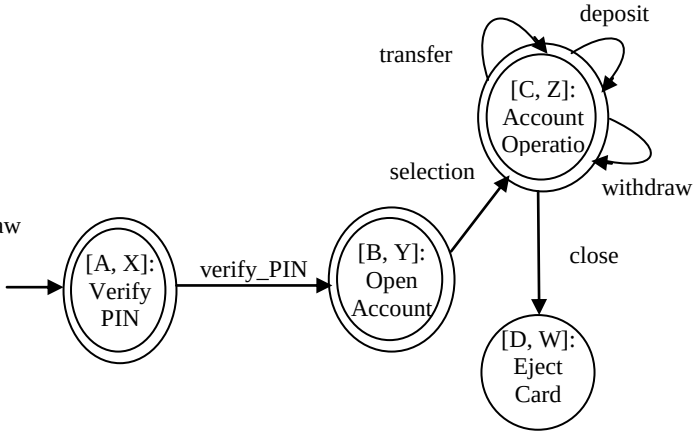
By the result of Theorem 1, L_{spec} is complete ($L_{spec} \subseteq L_{test}$) iff $(L_{spec} \cap (L_{spec} \cap L_{spec})^c) \cup ((L_{spec})^c \cap (L_{spec} \cap L_{test})) = \phi$. So we have to construct a finite automata (FA) to accept $(L_{spec} \cap (L_{spec} \cap L_{spec})^c) \cup ((L_{spec})^c \cap (L_{spec} \cap L_{test}))$. If this FA accepts an empty string (i.e., ϕ), then this specification is complete. To construct this FA, it involves three primitive rules: Intersection, Complement and Union proposed above.

Let $M_{1\cap2} = (M_1 \cap M_2) = (Q_1 \times Q_2, \Sigma, \delta, [A, X], [D, W])$ shown in Fig. 12-(a) which accept $L_{spec} \cap L_{test}$, where $Q_1 \times Q_2$ = {[A,X], [B,Y], [C,Z], [D,W]}, Thus transition δ in $M_{1\cap2}$ is δ ([*A*, X], a) = [δ (A, a), δ (X, *a*)] = [B,Y]; δ ([*B*,Y], b) = [δ (B, b),δ (Y, b)] =[C, Z]; δ ([C, Z],b) = [δ [C, b],δ [Z, b]) = [C,Z]; δ ([C, Z],c) = [δ [C, c],δ [Z, c]) = [C,Z] ; δ ([C, Z],e) = [δ[C, e], δ [Z, e]) = [D,W].

Let $M_{(1\cap2)}{}^c = \cap = (Q_1 \times Q_2, \Sigma_1 + \Sigma_2, \delta, [A, X], \{[A, X], [B,Y], [C, Z]\})$ shown in Fig. 12-(b) which accept $(L_{spec} \cap L_{test})^c$, where $Q_1 \times Q_2$ = {[A, X], [B,Y], [C, Z], [D,W]}, This transition δ in $M_{(1\cap2)}{}^c$ has the same set that $M_{1\cap2}$ has. However, the final state of $M_{(1\cap2)}{}^c$ is {[A, X], [B, Y], [C, Z]} which is the complement of the final state of $M_{1\cap2}$ (i.e., $Q_1 \times Q_2$ - [D,W]). Note that these final states are denoted by drawing the state icons with a double line.



(b) Intersection $M_1 \cap M_2$



(a) Intersection $(M_1 \cap M_2)^c$

Fig 12: Illustration for $L_{spec} \cap L_{test}$ and $(L_{spec} \cap L_{test})^c$ respectively

The final automata to accept $(L_{spec} \cap (L_{spec} \cap L_{spec})^c) \cup ((L_{spec})^c \cap (L_{spec} \cap L_{test}))$ is M_3 shown in Fig. 11 Because it is not an empty string, the Transaction state is incomplete. By the same way, check the other states in this ATM system specification.

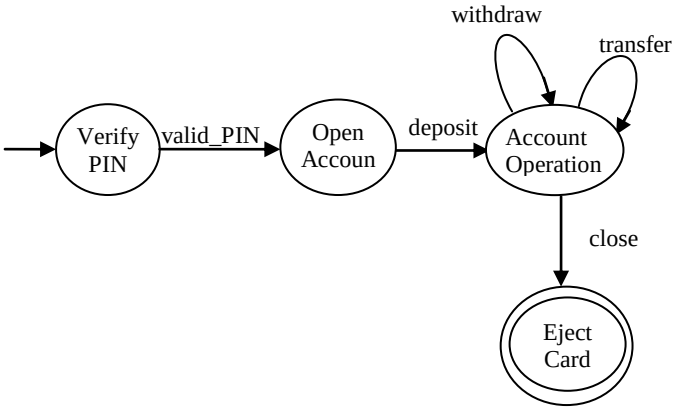


Fig 12: Illustration for $L_{spec} \cap (L_{spec} \cap L_{test})^c \cup (L_{spec})^c \cap (L_{spec} \cap L_{test})$

Every OO specification consists of a set of states (classes) which communicate with each other by sending message or invoking methods. Therefore, the analysis of completeness of an OO specification must take the whole states into consideration. Meanwhile, MISS test data satisfies distributive property and associative property, so check the completeness of OO specification state by state.

7.1.2.3 Consistency Analysis

Let's consider M_1 in Fig. 11 again. Given an invocation sequence "valid _ PIN • withdraw • deposit • withdraw • close", we know this is a wrong invokation sequence because any customer of a bank should deposite some cash into a bank before any withdraw. The transition δ (Verify _ PIN, valid _ PIN • withdraw • deposit • close) will stick to the state Loan Account and will not go to final state.

δ (start, invocation _ sequence) = δ (Verify _PIN, valid _ PIN • withdraw • deposit • close)

= δ (δ (Verify _ PIN, valid _PIN), withdraw • deposit • close)

= δ (δ (Open _ Account, withdraw), deposit • close)

= δ (Loan _ Account, deposit) • close)

= stick in Loan _ Account

7.1.2.4 Error Dection

Lemma 1: Given M = (Q, Σ, δ, q_0, F) be a DFA, A state q ∈ Q is stuck if there does not exist a word x ∈ Σ* such that δ (q, x) = f, for some f ∈ F.

Theorem 2: Given two DFAs denoted by $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$, and $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2)$ if $M_1 \nsubseteq M_2$, there must exists a set of states $A \in Q_1$, $B \in Q_2$ such that x is stuck at B and y is stuck at A, where $x \in L_1$, $y \in L_2$.

Proof: Given an invokation sequence, valid _ PIN • withdraw • deposit • close chosen from $L(M_1)$, but this is a wrong design because any customer of a bank should deposite some cash into a bank before any withdraw. When the word (operation sequences) is carried out against L_{test}, it will stick in some state of L_{test}. In contrast, there must exists some sequence in L_{test} if $\$ L_{test} \notin L(M)$, when input this invokation sequence to L(M), it must stick in L(M).

By Theorem 2, let $L_{spec} = (Q, \Sigma, \delta, q_0, F)$ respect to an object-oriented programming and L_{test} be a test data generated from formal specification, if $L_{spec} \notin L_{test}$ (i.e., inconsistency), there must exist stuck states in Q for some $x \in L_{test}$.

Definition 1: A test data is complete if it covers all the requirements indicated in the problem specification.

Definition 2: A test data is sound if it no invalid test cases are generated.

8. CONCLUSION

This paper has presented a testing framework based on finite automata. Test requirements derive from early artifacts produced at the end of the analysis development stage, namely use case diagram, use case description, interaction diagram (sequence or collaboration) associated with each use case, and class diagram (composed of application domain classes and their contracts). The fundamental principles of our methodology, which is based on regular expression, are emphasized here. The given methodology focuses on test automation, A bank's ATM system is used to illustrate the testing framework. One major issue currently under investigation is: how to strengthen the testing framework. But finite automata is only a subset of context free grammar, so it has some limitations in applications. Petri-net is a more powerful state-based machine which is very suitable for the description of object-oriented specification.

9. ACKNOWLEDGMENT

I am heartily thankful to Dr. Ajay Pratap, whose encouragement, guidance and support enabled me to develop an understanding of the subject.

10. REFERENCES

[1] J. E. Hopcroft and R. M. Karp. 1971. A linear algorithm for testing Equivalence of finite Automata. Cornell University, Tech. Rep. TR 71-114.

[2] A. J. Offutt and A. Abdurazik. 1999. Generating Tests from UML specifications. Proceedings Second International Conference on the Unified Modeling Language (UML'99), Fort Collins, CO. pp. 416-429.

[3] Vipin Saxena and Ajay Pratap. 2012. Modeling and Validation of Object-Oriented Database System. International Journal of Computer and Electrical Engineering, Vol. 4, No. 5.

[4] D. Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming. vol. 8, pp. 231-274.

[5] A. Andrews, R. France, S. Ghosh, and G. Craig. 2003. Test Adequacy Criteria for UML Design Models. Software Testing, Verification, and Reliability Journal. Vol. 13, No 2.

[6] H. Y. Chen, T.H. Tse, F.T. Chan, and T.Y. Chen. 1998. In Black and White: An Integrated Approach to Class-Level Testing of Object-Oriented Programs. ACM Transactions on Software Engineering and Methodology. Vol. 7, No. 3, pp. 250–295.

[7] Chi-Ming Chung, Timothy K. Shih, Chun-Chia Wang, and Ming-Chi Lee. 1997. Integrating Object-Oriented Software Testing and Metrics. International Journal of Software Engineering and Knowledge Engineering. U.S.A, Vol.7, No. 1, pp.125-144.

[8] E. Weyuker, T. Goradia, and A. Singh. 1994. Automatically generating test data from boolean specification. IEEE Transactions on Software Engineering. Vol. 20, no. 5.

[9] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J.D. Reese. 1994. Requirements Specification for Process-Control Systems. IEEE Transactions on Software Engineering. vol. 20, no. 9, pp. 684-707.

[10] M. S. Jaffe, N. G. Leveson, M.P.E. Heimdahl and B. Melhart. 1991. Software Requirements Analysis for Real-Time Process-Control Systems. IEEE Transaction on Software Engineering. vol. 17, no.3 pp.241-258.

[11] J. D. Ullman and J.E. Hopcroft. 1979. Introduction to Automata Theory, Languages and Computation. pp. 58-62, Addison Wesley.

[12] I. A. Zualkernan, W.T. Tsai. 1992. Object-Oriented Analysis and Design: A Case Study. International Journal of Software Engineering and Knowledge Engineering. vol. 2, no. 4, pp. 489-521.

[13] E. F. Miller, Introduction to Software Testing Technology. Tutorial: Software testing &Validation Techniques. Second Edition. IEEE Catalog No. EHO 180-0, pp. 4-16.

[14] E. M. Clarke, E. A. Emerson, and A. P. Sistla, 1986. Automatic verification of finite-state concurrent systems using temporal logic. Trans. Prog. Lung. and Syst., vol. 8, no. 2, pp. 244-263.