A Testing Framework Based on Finite Automata for Object-Oriented Software Specification

Ming-Chi Lee
Department of Information Technology
National PingTung Institute of Commerce
Ming-Sheng E. Road, Pingtung Taiwan, R.O.C.
E-mail: lmc@npic.edu.tw

Abstract

For the past decade, with the growing popularity of World Wide Web, object-oriented programming, such as Java and Visual .Net, have been widely applied to develop large software system on Internet. Moreover, object-oriented (OO) software techniques have gradually replaced procedure-oriented programming as the mainstream in software industry. However, the software errors still increases in proportion to the software system complexity. Although, there have been lots of researches on the object-oriented software testing (OOT) proposed to test OO software errors, most of them focused on the single class testing instead of high level testing for OO specification. This paper provides a testing framework based on finite automata to test the inconsistency and incompleteness of OO 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.

Keywords: Object-Oriented Program, software testing, software complexity.

1. Introduction

In recent years object-oriented paradigm is gaining acceptance for developing large and complex software. OO paradigm has moved into mainstream software development industry. This is due to a variety of claims about how it may improve the development of software, including such factors greater reusability, flexibility and increased extensibility. These outstanding advantages so far in OO software engineering are focused on problem analysis, class design, and implementation techniques, but do not include testing methodologies. Basically, object-oriented analysis (OOA) and object-oriented design (OOD) methodologies examine the problem in the real world and facilitate in decomposing the problem in terms of classes, and some relationships between classes. Therefore, the result of any OOA and OOD technique is a specification of the system under construction and should be complete and consistent with the problem requirements. However, almost all large OO software specifications still contains *incompleteness*, *inconsistentency,* and ambiguity [2]. This is because the formulation of OO software specifications require extensive human guidance and writing [12]. We know that many serious conceptual errors are introduced in this first stage of software development --- errors introduced during the specification stage have been shown to be more difficult and more expensive to correct than errors introduced later in the software lifecycle[17,18,19]. In view of this, it is important to provide methods and techniques to test specification-related errors in the early stage of software development[1]. Especially, the inconsistency and incompleteness of specification which lead to the design errors and implementation errors must be detected as early as possible [8,9,10,11]. Although different approaches have been proposed to test OOPs over the past decade, most of them focused on the static and manual analysis of program behavior in a single class instead of OO specification[8,11,12]. The problem of inconsistency and incompleteness of OO software specifications has been rarely discussed. 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. **Formal Proof Systems**. Formal proof systems can be powerful tools in the verification of critical properties of algorithm [20]. Attempts have been made to extend the use of formal proofs and apply them to requirements specifications, for example, the ProCoS (Provably Correct Systems) Project [21,22]. 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 involved in most requirements specification efforts and thus are not (in our opinion) suitable as high-level specification languages. Also, formal proofs are notoriously difficult to derive, and these approaches may not be practical for complex systems. **Model Checking**. Model checking is conceptually simple and is applicable in a wide variety of languages and application areas [23]. Early work in model checking also relied on a global reachability graph. Consequently, the approach suffered from state-space explosion problems. Newer approaches relying on a symbolic representation of the state space can significantly improve the performance of the model checking approach. Symbolic model checking has been applied to large models [24], but only for systems with simple, repetitive elements --- such as those commonly found in hardware applications. The time and space complexity of the symbolic approach is affected not only by the size of the specification but also by the regularity of specification. Software requirements specifications lack this necessary regular structure, and it is unclear how well the symbolic approach will perform on these specifications. Our 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. The contribution of this paper is to provide an innovative object-oriented testing framework modeled by finite automata and give a concrete example to show how to test the object-oriented software specification by using finite automata. The next section outlines the approach of this paper. Section 3 introduces a state-based requirement specification language called RSML to describe object-oriented specifications. In Section 4, an object-oriented testing methodology based on finite automata is proposed to test OO specifications for the aspects of incompleteness and inconsisitency. In addition, an algorithm based on automata is proposed to support the testing framework. In Section 5, we propose a criteria called *method invocation sequence scenario* (MISS) as a guide to generate test data from the RSML specification. This test data is described in the form of regular expression. Section 6 analyzes the test results. Section 7 concludes the paper.

2. Testing Framework

The approach to object-oriented testing(OOT) proposed here includes 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 requirement state machine language (RSML) is a state-based requirement specification language which provides a convenient way to describe the *operation sequences* of an object-oriented specification [4]. The RSML specification provides a means for the tester to retrieve implementation information without going to details [13]. It also provides the definition of whole class hierarchy and gives a precise description of the desired operation behavior. Although we prefer to use RSML to depict the object-oriented specification, it would be possible to describe object-oriented specification using other state-based specification notations such as Object Diagram, Event Trace Diagram, and Use-case diagram which are suggested by Booch[28]. It is worthy to note that not having to introduce and force users to learn a new notation is an important feature of this testing framework. Basically, the RSM model we use is a Mealy Machine [29], so we can convert the RSML specification to a finite automata directly. On the other hand, the test data derived from RSML specifications can be defined as a sequence of operations (method invokations) along with expected effects; it can be represented by a sequence of events or states. In this paper, a test data generation criteria called *method invokation sequence scenario* (MISS) is proposed to documents the correct causal order in which the methods of a class specification can be invoked. We will show how the test criteria can help generate adequate test data in the form of regular expression. Since the OO specification and test data are both represented in the form of regular expression, we can construct a testing algorithm based on finite automata to test the object-oriented specification. Fig.1 shows the full picture of this object-oriented testing framework. This approach for object-oriented software testing can be summarized as:

1 Construct the OO specification with RSML
2 Map the RSML specification to a finite automata
3 Derive adequate test data.
   - A method invocation sequence specification (called MISS) criteria is used to help generate test data.
   - Express the test data in the form of regular expression
4 Design a testing algorithm based on finite automata to test the OO specification by comparing the test data against the finite automata.
5 Evaluate test results. If there exists incompleteness and inconsistency, then go back the specification phase and refine the specification again else stop the testing process.

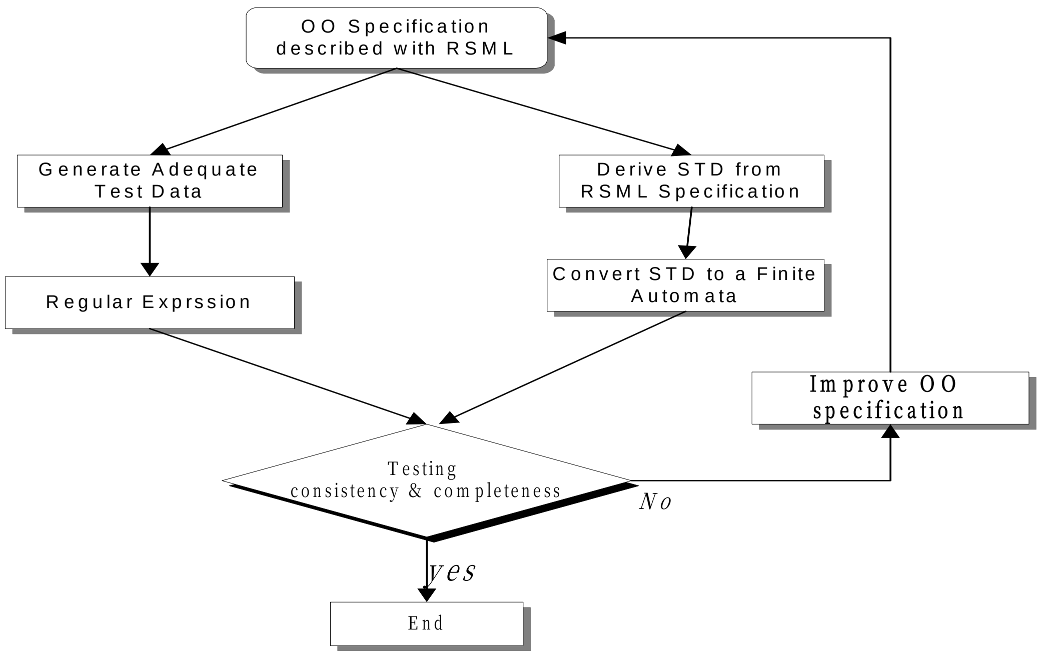

OO Specification described with RSML
Generate Adequate Test Data
Derive STD from RSML Specification
Convert STD to a Finite Automata
Regular Exprssion
Improve OO specification
Testing consistency & completeness
No
yes
End

Figure 1: Testing Framework

3. RSML Specification

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 [3]. It extends conventional state diagrams with state hierarchies and broadcast communications. Focusing on a subset of RSML, we model a system by a state hierarchy, events, and inputs; in particular, the input and output interfaces in RSML are ignored. In the following, we give a brief overview of RSML.

3.1 Overview of the RSML Notation

RSML includes several features developed by Harel for Statecharts [25,26]. A complete description of RSML is provided in [3]. This section contains only a description of the RSML features necessary to understand this paper. A simple finite-state machine is composed of *states* connected by *transition* (see Fig.2). *Default* or *start* states are signified by states whose connecting transition has no source. In Fig. 2 state *A* is the start state. Transitions define how to get from one state to another.

Transition Definitions. Transitions are labeled with logical expressions of the form *Input predicate*} || *Output predicate*, and a transition is taken if the *predicate* on that transition evaluates to *true*. If an output is to be produced, the constraint on that output are expressed in the *Output predicate* associated with the transition. For the sake of simplicity of notation, the output predicates are ignored in this paper.

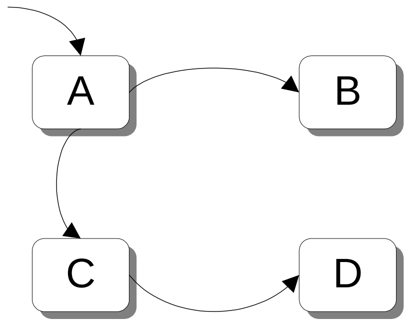


Figure 2: A basic state machine

Superstates. In RSML (and Statecharts), a state that contains nested states is called a *superstates*, and its nested states are called *substate*. A state that cannot be *decomposed* further is called a *leafstate*. In Fig.3, state R and state T may be grouped into *superstate* S. Such groupings reduce the number of transition by allowing transitions to and from the superstate rather than requiring explicit transitions to and from all of the grouped states (*substates*). Superstates can be entered in two ways. First, the transition to superstate may end at the superstate's border (transition *a* in Fig.3). In this example, state *S* is entered upon taking transition *a*. Alternatively, the transition may be made to a particular state inside the superstate (transition *b* in Fig.3). Analogous to transitions into the superstate, transitions out of the superstate may originate from the border or from an inner state. The same superstate may contain both types of exiting transitions.

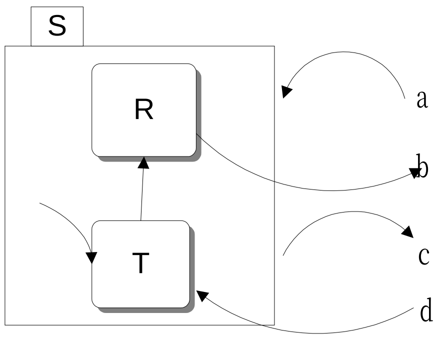


Figure3: A superstate example

3.2 State Transition Diagram for RSML Specification

The RSML specification itself is a state transition diagram (STD). These definitions in section 3.1 provide the abstraction capability needed for the derivation of state transition diagrams in a multi-layer style. This multi-layer representation provides the tester a direct mapping between design specification and usage, and allows the user to view the design in an understandable way. It is also feasible to use this approach to handle the state explosion problem of complex systems.

The advantages of deriving the state transition diagram in the way include:

1 Since a class or an object in RSML specification can be represented in a STD, it is natural to represent the whole system as a STD.

2 The behavior of a complex system can be represented in one diagram that allows the user to navigate through several levels. Each level can be fitted into a single

page for easier review and documentation.

3 The STD derived from the RSML specification is a design chart for programmers. It allows testers to check whether the programs executes according to the specification.

4 The behavior of a STD is equivalent to some finite automata. So we can easily convert the RSML specification to a finite automata which is easily programmable.

Each state or event in the STD is annotated. A '*' beside the state label is used to denote a *superstate*. An example of STD is shown in Fig.4(a). According to McCabe's complexity measure [27], a set of five basis paths cover all states and transitions in this STD. The basis set is not unique. In the following basis set example, v_n and e_m are used to indicate node(state) n, and edge (transition) *m*, respectively.

Path 1: $v_1v_2v_3v_5v_6v_7v_8$
Path 2: $v_1v_2v_4v_5v_8v_9$
Path 3: $v_1v_2v_4v_5v_6v_8v_9$
Path 4: $v_1v_2v_4v_5v_6v_7v_7v_8v_9$
Path 5: $v_1v_2v_3v_5v_6v_8v_9$ \$

A path can also be represented by a sequence of edges. For example, Path 1 can be $e_ae_ce_ee_fe_ge_je_l$ \$. In a substate, entry and exit nodes are denoted by double circles. Using this decomposition, v_5 is rewritten as $v_{5.1}(v_{5.2} \mid (v_{5.3})^+)v_{5.4}$ (see Fig.4-b). Path 1 is then decomposed as $v_1v_2v_3v_{5.1}(v_{5.2} \mid (v_{5.3})^+)v_{5.4}v_6v_7v_8v_9$. By stepwise refinement, a path can ultimately be represented by a sequence of transitions.

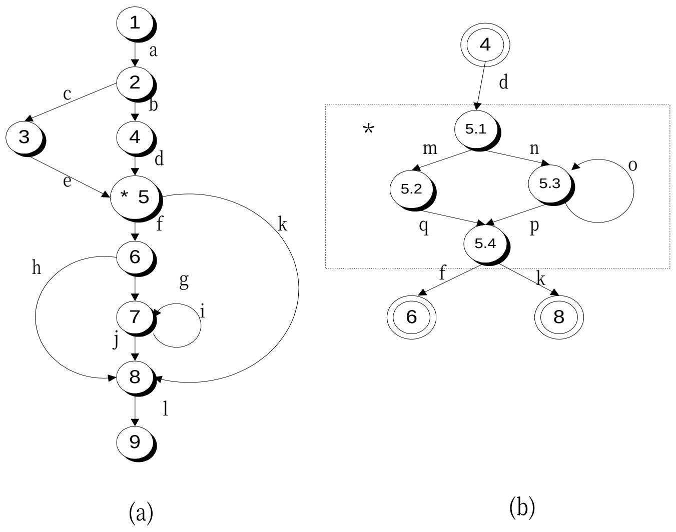


Figure 4: An example of STD

If RSML specification contains no loops, the set of all possible sequence is finite. However, this case is rather rare in real applications. The previous example contains a loop, and hence an infinite number of paths. Our approach is not to use McCabe's method to collect the basis paths, but to map the whole STD to a finite automata such that we can test the OO specification by comparing the finite automata with a set of test data which is derived in the form of regular expression. A suitable criteria for the test data generation of OO specification will be specified in next section.

3.3 Convert STD to Finite Automata

The state model we use is a Mealy machine[29] with outputs on the transitions between states. The STD is denoted as a eight-tuple $M = (\Sigma, Q, q_0, P_T, P_O, \delta, F, \lambda)$ where:

- Σ is the set of input events.
- Q is the states in M
- $q_0 \in Q$ is the initial state of M; the software is in the state before startup.
- P_T is the set of boolean functions over Σ representing predicates on the values of inputs
- P_O is the set of boolean functions over Σ representing predicates on the values of output.
- δ is the state transition function mapping $Q \times P_T \rightarrow Q$. That is, $\delta(q, p)$ where $q \in Q$ and $p \in P_T$ defines the next state when the software is in state q and takes the transition having p as the input predicate.

- λ is the trigger-to-output relationship mapping from $Q \times P_T$ to P_O.

Guidelines for mapping STD specification to corresponding finite state machine take three steps. First step, every predicate P_T in $M = (\Sigma, Q, q_0, P_T, P_O, \delta, F, \lambda)$ should be set to *true*. Second, every output P_O need be ignored. On the other words, $M = (\Sigma, Q, q_0, P_T, P_O, \delta, F, \lambda)$ is reduced to a 5-tuple $M = (\Sigma, Q, q_0, \delta, F)$. Third step, λ is replaced with a *pseudo final state*. It should be noted that many complex systems are not a simple input/output mapping. However, a test scenario cannot be designed to run forever, and a *pseudo final state* must be reached after the program has gone through a test scenario. A pseudo final state is inserted after finishing a transaction in the finite state machine.

4. Testing Methodology

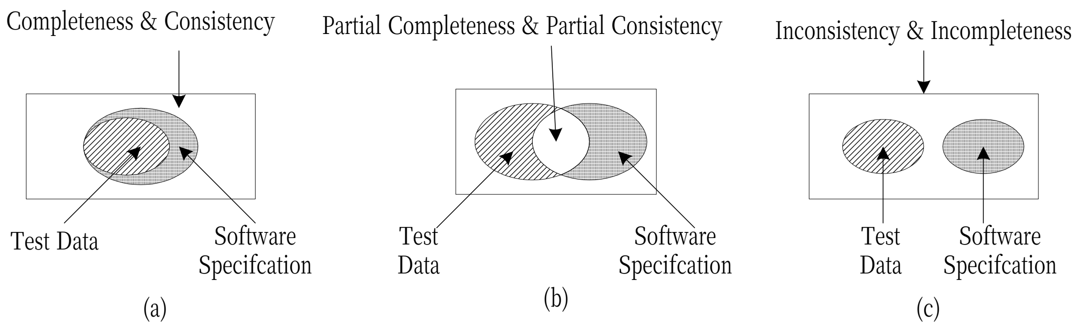
In this section, we present the object-oriented testing methodology modeled by finite automata. Instead of manually verifying the OO software specification, an innovative method is presented to test the incompleteness and inconsistency of OO software specification.

4.1 Incompleteness

Before tackling this issue, we need to clarify the real meaning of {\em completeness} of software specification. Generally speaking, {\em completeness} determines the extent to which a software system is solving the correct problem[12]. The completeness of a system with respect to the problem specification is a measure of the portion of the specification implemented in the system[4]. However, it is very difficult to verify to what extent can the completeness of specification be trusted because completeness of software specification is not easy to measure. In definition 4, we make a more precise description for the *completeness* of software 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 we can say that the software specification L_{spec} is *complete* if $L_{test} \subseteq L_{spec}$ holds. Fig.5 depict three testing relationships between L_{spec} and L_{test}. Fig.5-a indicates that the software specification L_{spec} is complete and consistent with L_{test} because $L_{test} \subseteq L_{spec}$ holds. Fig.5-b means that L_{spec} is partially complete and consistent with L_{test}. Fig.5-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 ($L_{test} \cap L_{spec} = \phi$). Obviously $L_{spec} \cap L_{req}$ 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 $L_{spec} \cap (\overline{L_{spec} \cap L_{test}})$. However, the difficulties lie in how to exactly figure out the two expressions: $L_{spec} \cap L_{test}$ and L_{spec} - ($L_{spec} \cap L_{test}$). We will utilize finite automata to resolve the problems.



In the following, we introduce a lemma in [29] which is helpful to construct our testing framework. It shows how to verify whether two regular expressions are equivalent or not. In this paper, because OO specification and test data are both represented in the form of regular expression, we will use this lemma to design a new method to test the incompleteness of OO specifications.

Lemma 1: Let M_1 and M_2 be two FAs accepting L_1 and L_2, respectively. There exists an automata M_3 which accepts $(L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2)$ to determine if two regular expression are equivalent (where $\overline{L}$ denotes the complement of L). *Proof:* see [29].

If $L_1 = L_2$, one can verify this, by constructing a new DFA, supposedly denoted as D_1, satisfying $L_1 \oplus L_2$ ($\oplus$ is the exclusive-or operator) and verifying whether D_1 accepts any non-empty string. If D_1 represent an empty regular set, then $L_1 = L_2$. In the following, we derive a theorem based on lemma 1 to detect the incompleteness of a given OO specification.

Theorem 1: Given an object-oriented specification denoted as L_{spe} (M), and a testing template L_{test} generated from its problem requirement, L_{spec} is *complete* ($L_{test} \subseteq L_{spec}$) *iff* $(L_{test} \cap \overline{L_{test} \cap L_{spec}}) \cup (\overline{L_{test}} \cap L_{test} \cap L_{spec}) = \phi$.

Proof: If $L_{test} \subseteq L_{spec}$ holds, then it implies that $L_{test} = L_{test} \cap L_{spec}$. Let $L(M_3)=L_3=(L_{test} \cap L_{spe})$. Then, $L_{test} = L_{test} \cap L_{spec}$ is equal to $L_{test} = L_3$. By the result of Lemma 1, $L_{test} = L_3$ iff $(L_{test} \cap \overline{L_3}) \cup (\overline{L_{test}} \cap L_3) = \phi$. Therefore, it implies that $L_{test} \subseteq L_{spec}$ iff $(L_{test} \cap \overline{L_{test} \cap L_{spec}}) \cup (\overline{L_{test}} \cap L_{test} \cap L_{spec}) = \phi$. This means that there exists a FA which accepts the language $(L_{test} \cap \overline{L_{test} \cap L_{spec}}) \cup (\overline{L_{test}} \cap L_{test} \cap L_{spec}) = \phi$.

Theorem 1 provides a formula to test if L_{spec} is complete. This formula shows that we are able to build a FA to verify whether $L_{test}(M) \subseteq L_{spec}$ holds. Obviously, the testing work is to construct a FA to accept the regular expression $(L_{test} \cap \overline{L_{test} \cap L_{spec}}) \cup (\overline{L_{test}} \cap L_{test} \cap L_{spec})$. However, the difficulties lies in how to construct this FA which accepts $(L_{test} \cap \overline{L_{test} \cap L_{spec}}) \cup (\overline{L_{test}} \cap L_{test} \cap L_{spec})$. Fortunately, there have been three primitive regular expression properties to help construct the FA [29]. They are shown as follows:

1. *Complement*: Given an automata denoted $L(M)= M = (\Sigma, Q, q_0, \delta, F)$, the complement is $\Sigma^* - L$. To accept $\Sigma^* - L$, complement the final states of M. That is $\overline{M} = M = (\Sigma, Q, q_0, \delta, Q - F)$. Then $\overline{M}$ accepts a word w if and only if $\delta(q_0, w)$ is in Q-F, that is, w is in $\Sigma^* - L$.
2. *Union*: Given two automatas $L(M_1) = (\Sigma_1, Q_1, q_1, \delta_1, F_1)$,and $L(M_2) = (\Sigma_2, Q_2, q_2, \delta_2, F_2)$, 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 $L(M_1) \cup L(M_2)$ $= M_{1\cup 2}(Q_1+Q_2, \Sigma_1 + \Sigma_2, \delta_1 + \delta_2, q_1 + q_2, F_1 + F_2)$
3. *Intersection*: The construction of *intersection* involves taking the Cartesian product of states, and we sketch the construction as follows, Let $L(M_1) = (\Sigma_1, Q_1, q_1, \delta_1, F_1)$ and $L(M_2) = (\Sigma_2, Q_2, q_2, \delta_2, F_2)$ be two deterministic finite automata. Let M=($Q_1 \times Q_2, \Sigma, \delta, [q_1, q_2], F_1 \times F_2$) where for all

p_1 in Q_1, p_2 in Q_2 and a in Σ. It is easily shown that $L(M)=L(M1) \cap L(M2)$.

In fact, the regular expression properties mentioned above are three primitive operations used to construct our proposed testing algorithm. We have designed three algorithms to implement the three regular expression properties respectively. Appendix A outlines the data structures and the algorithms which are described with C++ language. In the following, the testing algorithm which contains the three primitive operations is given in Fig.6.

Algorithm: Verify the Completeness of OO Specification
Input: An object-oriented Specification L_{spec} and a testing data L_{test}(M)
Output: A boolean result

1. Construct an automata M to accept $(L_{test} \cap \overline{L_{test} \cap L_{spec}}) \cup (\overline{L_{test}} \cap L_{test} \cap L_{spec})$ where M consists of *complement automata, union automata,* and *intersection automata* shown in Appendix A.
2. *if* $L(M) = \phi$ *then* L_{spec} is complete *else* there exists incompleteness.

Figure 6: An algorithm to verify OO specification's completeness

In the following we would like to show a method to extract the incomplete part of L_{spec} to enhance the Theorem 1.

Theorem 2: Given an object-oriented specification $M = (\Sigma, Q, q_0, \delta, F)$ denoted as $L_{spec}(M)$, and a testing template L_{test} in the form of regular expression, the *incomplete* part of specification is $L_{spec} \cap (\overline{L_{spec} \cap L_{test}})$.

Proof: The intersection of L_{spec} and L_{test} is the complete part between object-oriented specification and testing template, denoted as $L_{spec} \cap L_{test}$. Assume that the test data cover all the requirement problem. By definition 4. we know the incomplete part of this specification is L_{spec}- ($L_{spec} \cap L_{test}$) which is equal to $L_{spec} \cap (\overline{L_{spec} \cap L_{test}})$ via the complement property.

Algorithm: Extract the incomplete part from OO specification
Input: An object-oriented Specification L_{spec}(M) and a testing data L_{test} (M)
Output: Incomplete part of OO specification

1. if $(L_{test} \cap \overline{L_{test} \cap L_{spec}}) \cup (\overline{L_{test}} \cap L_{test} \cap L_{spec})$ is ϕ then go to end

2. Construct an automata *M* to accept $L_{spec} \cap (\overline{L_{spec} \cap L_{test}})$ where *M* consists of *complement automata*, *union automata*, and *intersection automata*.

Figure 7: An algorithm to extract the incomplete part from OO specification

4.2 Inconsistency

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. One is the *unreachable state* which reveal some kind of logical error in specifications. An *unreachable state* means that there does not exist any input sequence can reach it. The other is the *stuck* state which causes some conflictions between problem requirement and solution specification. A *stuck state* is that there does not exist any input sequence which can pass through it to reach the final state.

Definition 3: Given an OO specification $M = (\Sigma, Q, q_0, \delta, F)$, 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 $\exists$ s: $\delta(q_n \cdot s) = q_m$ where $s \in \Sigma^*$.

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. In the following, we show an algorithm to find out the unreachable states (see Fig.8).

```
Algorithm Search Unreachable States
Input: A directed Graph G=(V,E) and a node u in V
Output: The set UR contains all not reachable states from u
begin
    R:={u}; N:= {u}
    Repeat
        T:=empty
        for all v in N do
        T:= T $\cup$ {w | (v,w)$\in$E}
        N:=T-R;
        R:=R $\cup$ N
    Until N=empty
    UR:= V - R
end
```

Figure 8: Search Unreachable States

Definition 4: Given an OO specification $M = (\Sigma, Q, q_0, \delta, F)$, a state $q \in$ Q is stuck if there does not exist a word $x \in \Sigma^*$ such that $\delta(q, x0 = f$, for some $f \in$ F.

The construction of object-oriented specification is an interactive and manual process. During this period, lots of classes (states) could be newly inserted or deleted frequently. Furthermore, different kind of classes could be aggregated to simplify the class hierarchy design. Many classes which should be aggregated or deleted originally were left and lead to stuck states unexpectedly. Stuck states not only conflicts the requirement specifications but also lead to inconsistency of specification [11]. An algorithm to search the stuck states is proposed (see Fig.9).

Theorem 3: Given an OO specification L_{spec}(M), and a testing template L_{test} (M), if L_{spec}(M) contains *stuck states*, then L_{spec}(M) $\not\subset$ L_{test} (M).
Proof: Prove it by contradiction. Suppose any input sequence $\in$ L_{spec} can reach final states, then it means L_{spec} satisfies all the problem requirement specification. On the other word, L_{spec} is a subset of L_{test}. This leads to a contradiction with Theorem 1.

```
Algorithm Search Stuck States
Input: A directed Graph G=(V,E) and a node $u$ in V
Output: stuck state
begin

   for each node in $V$
      call reachability
      check whether final states are in reachable set $R$
            if it does not contain final states, then
                  it is a stuck state
end
```

Figure 9: Search Stuck states

5. Test Data Generation

The primary goal of test data generation is to derive adequate test data to identify faults in software development. Although, Weyuker develops a general axiomatic theory of test data adequacy and proposed a set of criteria to generate adequate test data, these criteria were not suitable for specification-based (black-box) testing, because it was concentrated on program-based (white-box) testing [5,6,7]. For a specification-based testing, it is desirable to generate test data according to the original requirement specification which indicates what the software system should do. In this paper, the OO specification is formulated with a state-based requirement specification language called RSML, so the test data will be derived from the RSML specification. Basically, a RSML specification consists of a set of states and transitions which are like the *classes* and *methods invokation* in OOPs. It is naturally that classes will be associated with states in RSML.

In the following, we propose a criteria 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, we 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, we 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:

$$\bigcup_{i=1}^{n}[S_i] \longrightarrow m_i \bullet [S_j] \qquad (4.1)$$

where each S_i is a distinct label, $S_j \in \{ S_1, S_{\_2}, \ldots, S_{\_n}\}$, and method $m_i \in \bigcup_{i=1}^{n} Method\_of\,(S_i)$. This form means that after schema S_i invokes some method m_i 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 ``$open \cdot deposit \cdot withdraw \cdot close$''. 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:

$$open \cdot deposit \cdot (deposit \mid transfer \mid withdraw)^{*} \cdot close$$

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 *operational_account* state. Once a *deposit* is chosen, *deposit*, *transfer* and *withdraw* methods can be invoked until the transaction is end.

6. 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. In this section, a brief description on the ATM requirements is given, then the RSML specification and its corresponding finite automata follows.

6.1 Specification

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. 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 is processed and the customer database is updated. The ATM returns to the idle state after finishing each transaction. The requirements for the ATM can be elaborated.

1. Check ATM card

- Return the card immediately if it is an invalid card or another party is using the card with the same number.
- Return the card if it is valid but has been reported used for fraud.
- Hold the card if it is valid but too many invalid PIN attempts have been tried.

2.Input Selection

3,Transaction functions:

- Withdraw cash from an account.
- Deposit cash into an account.
- Transfer funds from an account.

6.2 The RSML specification of ATM

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. 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. The quality of the software will be improved, and the probability of changing requirements specification in the later phases will be reduced. For reasons of space, we will only be able to present some of states here; the complete specification is given in [15].

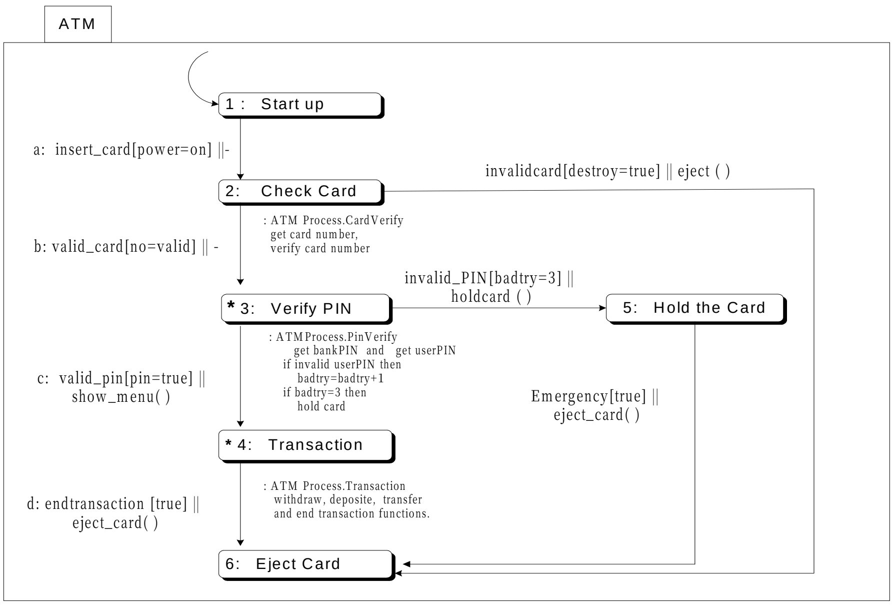


Figure 10. The RSML specification of ATM Transaction

The objects of the ATM can be translated to the RSML states in a straightforward way. For example, the states may include *Cardreader*, *CashDispenser*, *Account*, *Bank*, *ATMTransaction* and even *SecurityCamera*. Each object has its operations defined in the analysis phase. For example, *Cardreader* can perform *EjectCard*, *HoldCard*, *CleanHold* and *ReadCard* operations. We will start with the specification *ATMTransaction*, which is at the top level of the ATM system. The transitions between states are caused by either external events or internal operations. *ATMTransaction* is *idel* until a card is inserted (an external event). This causes a transition to a state that will check if the card is valid. If the card is valid, *ATMTransaction* reaches the *VerifyPIN* state, where it will request the user to input a PIN. If the PIN is valid, the ATM would receive account information from the bank (an internal operation) and cause the transition to the Transaction state. At this state, different transaction options can be selected. A transaction is processed based on its eligibility. Once a transaction is done, the account information is updated, and the ATM returns to the *idle* state. The above scenario describes normal usage of the ATM system. The specification of the *ATMTransaction* is given in Fig.10. This *ATMTransaction* specification has six major states where there are two *superstates*: state *verify_pin* and state *transaction* (see Fig.11 and Fig.12}). The *verify_pin superstate* check whether *bank_pin* is equal to *user_pin* which users input into the ATM. In our case, the ATM allows no more than three bad tries. The *transaction superstate*} include the basic services which an ATM can support. To facilitate the testing, we have to map the ATM RSML specification to a corresponding finite automata. Guidelines for converting RSML specification to a corresponding finite automata has been specified in section 3.1. Fig.13 shows the result of conversion from RSML specifications (Fig.10-12) to a finite automata.

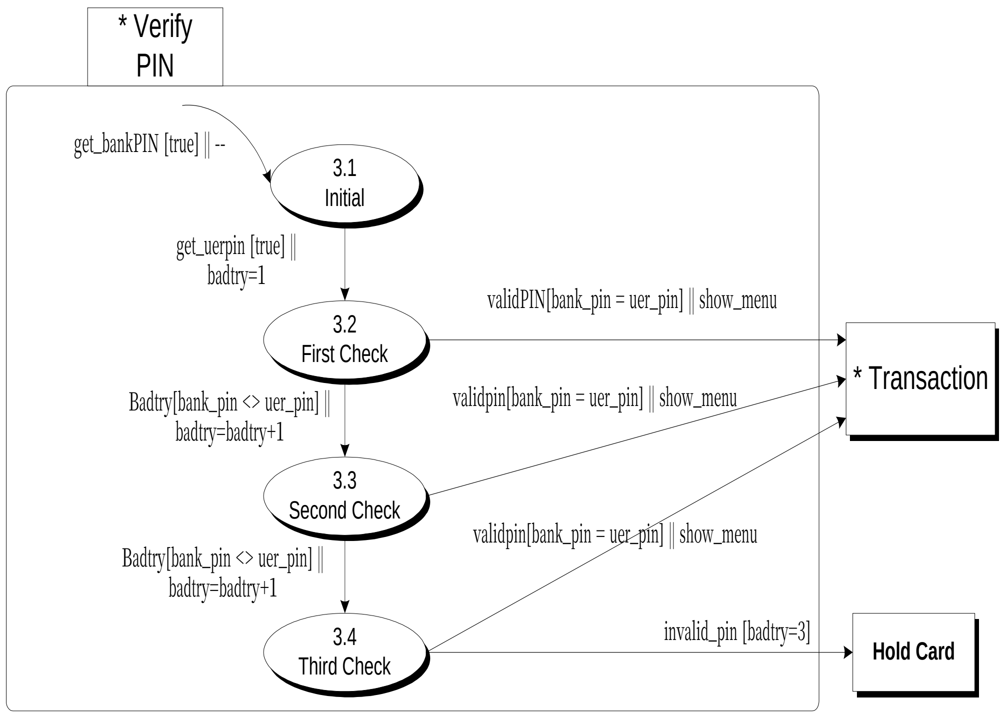

* Verify PIN
get_bankPIN [true] || --
3.1 Initial
get_uerpin [true] || badtry=1
3.2 First Check
validPIN[bank_pin = uer_pin] || show_menu
* Transaction
Badtry[bank_pin <> uer_pin] || badtry=badtry+1
validpin[bank_pin = uer_pin] || show_menu
3.3 Second Check
validpin[bank_pin = uer_pin] || show_menu
Badtry[bank_pin <> uer_pin] || badtry=badtry+1
3.4 Third Check
invalid_pin [badtry=3]
Hold Card

Figure 11. Verify PIN superstate in ATM

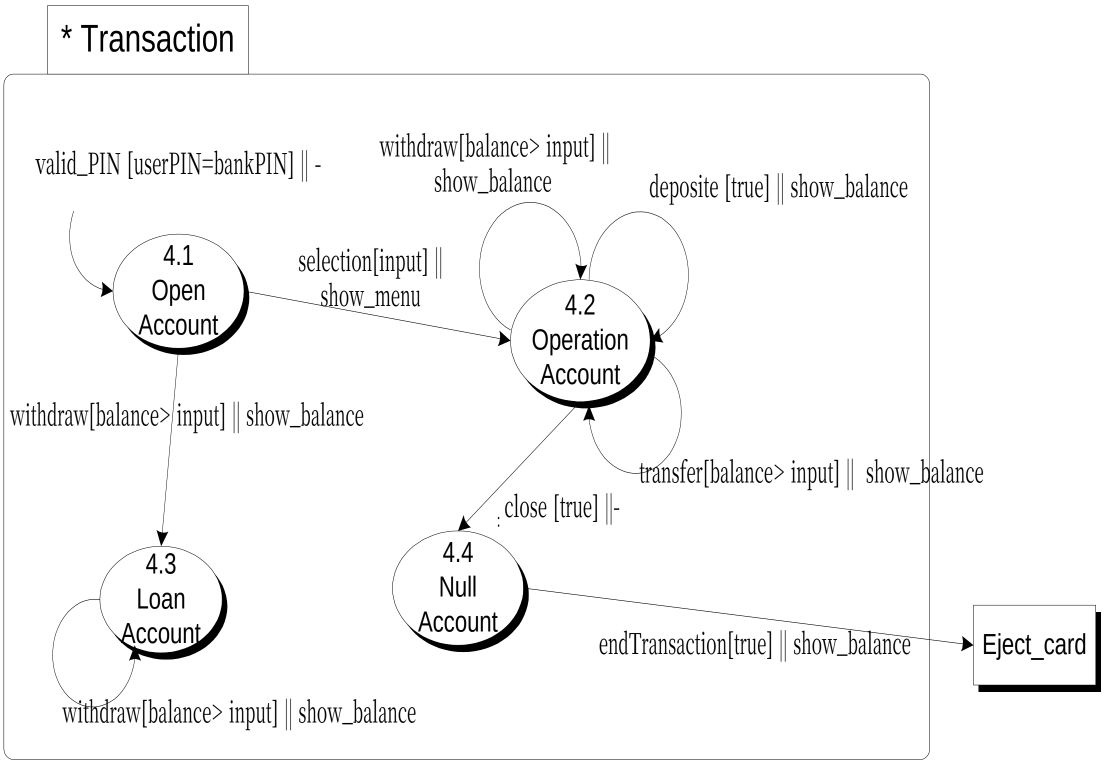

* Transaction
valid_PIN [userPIN=bankPIN] || -
withdraw[balance> input] || show_balance
deposite [true] || show_balance
4.1 Open Account
selection[input] || show_menu
4.2 Operation Account
withdraw[balance> input] || show_balance
transfer[balance> input] || show_balance
close [true] ||-
4.3 Loan Account
4.4 Null Account
endTransaction[true] || show_balance
Eject_card
withdraw[balance> input] || show_balance

Figure 12. Transaction superstate in ATM

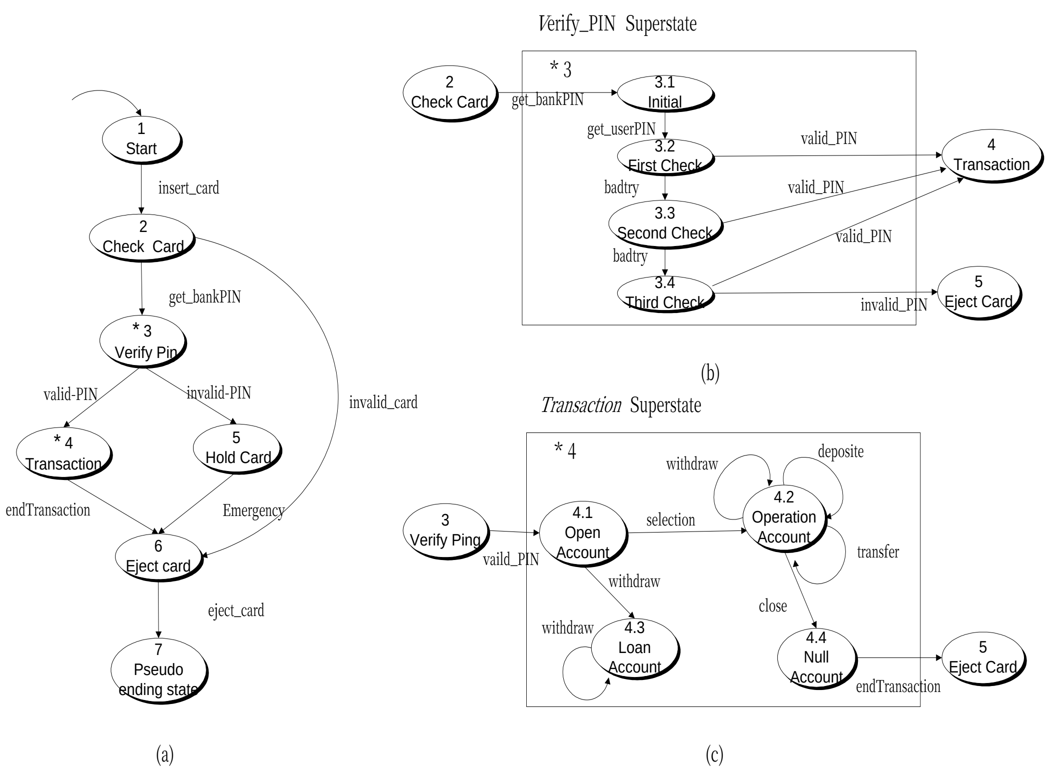


Figure 13. Convert the RSML specification to a corresponding finite automata

The test data generation has been specified in section 4. Since the generation process is based on the MISS criteria, it takes some effort to finish it. For the sake of space, we will 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*, *operation_account*, *loan_account* and *null_account*. Each state associates with a set of methods which causes the transitions from one state to another state. These transitions and test data are given in Fig.14.

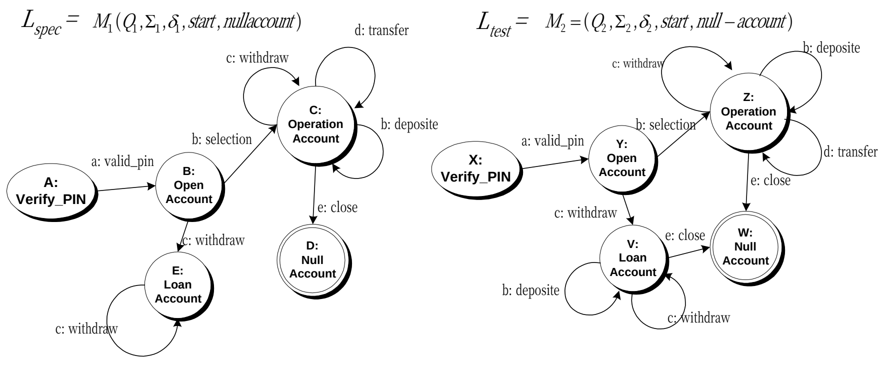
[Verify_PIN] $\longrightarrow$ valid_pin • [Open_Account]
[Open_Account] $\longrightarrow$ selection • [Operation_Account]
[Open_Account] $\longrightarrow$ withdraw • [Loan_Account]
[Operation_Account] $\longrightarrow$ withdraw • [Operation_Account]
[Operation_Account] $\longrightarrow$ transfer • [Operation_Account]
[Operation_Account] $\longrightarrow$ deposit • [Operation_Account]
[Operation_Account] $\longrightarrow$ close • [Null_Account]
[Loan_Account] $\longrightarrow$ deposit • [Loan_Account]
[Loan_Account] $\longrightarrow$ withdraw • [Loan_Account]
[Loan_Account] $\longrightarrow$ close • [Null_Account]
[Null_Account] $\longrightarrow \in$

Figure. 14 Test data of Transaction state in the form of regular definition

6.3 Test Result Analysis

6.3.1 Completeness Analysis

Now we would like to 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 Fig.14. Both of them can be represented by two corresponding finite automata (see Fig.15). The RSML specification is denoted as L_{spec}, and the test data is denoted as L_{test}.



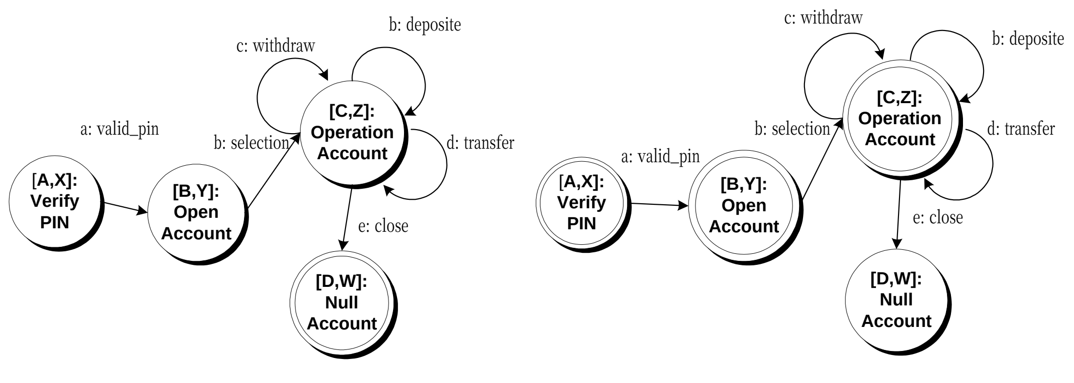
(a) RSM Specification of Trasaction (b) Test Data MISS of Transaction

Figure 15. An illustration of the testing framework: using transaction state.

By the result of Theorem 1, L_{spec} is *complete* ($L_{test} \subseteq L_{spec}$) iff $(L_{test} \cap \overline{L_{test} \cap L_{spec}}) \cup (\overline{L_{test}} \cap L_{test} \cap L_{spec}) = \phi$. So we have to construct a finite automata (FA) to accept $(L_{test} \cap \overline{L_{test} \cap L_{spec}}) \cup (\overline{L_{test}} \cap L_{test} \cap L_{spec})$. 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 in Section 5. In order to facilitate the construction, we adopt a *divide and conquer* manner to construct the FA. In the first step, we construct two automata, called $M_{1\cap 2}$ and $M_{\overline{1\cap 2}}$, to accept $L_{spec} \cap L_{test}$ and $\overline{\mathrm{L}_{spec} \cap \mathrm{L}_{test}}$ respectively where $M_{1\cap 2}$ denotes $M_1 \cap M_2$, and $M_{\overline{1\cap 2}}$ denotes $\overline{M_1 \cap M_2}$.

Let $M_{\overline{1\cap 2}} = \overline{M_1 \cap M_2}$ = ($Q_1 \times Q_2$, $\Sigma_1 + \Sigma_2, \delta$,[A,X],[D,W]) shown in Fig.16-(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 $M_{1\cap 2}$ is $\delta([A,X],a)$ = $[\delta(A,a),\delta(X,a)]$ = [B,Y]; $\delta([B,Y],b)$ = $[\delta(B,b),\delta(Y,b)]$ =[C,Z]; $\delta([C,Z],b) = [\delta[C,b],\delta[Z,b])$ = [C,Z]; $\delta([C,Z],c) = [\delta[C,c],\delta[Z,c])$=[C,Z] ; $\delta([C,Z],e) = [\delta[C,e],\delta[Z,e])$ =[D,W]

Let $M_{\overline{1\cap 2}}$ = $\cap$ = ($Q_1 \times Q_2$, $\Sigma_1 + \Sigma_2, \delta$,[A,X],{[A,X],[B,Y],[C,Z]}) shown in Fig.16-(b) which accept $\overline{L_{spec} \cap L_{test}}$, where $Q_1 \times Q_2$= {[A,X],[B,Y],[C,Z], [C,Z]}, This transition δ in $M_{\overline{1\cap 2}}$ has the same set that $M_{1\cap 2}$ has. However, the final state of $M_{\overline{1\cap 2}}$ is {[A,X], [B,Y], [C,Z]} which is the complement of the final state of $M_{\overline{1\cap 2}}$ (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.



(a) Intersaction $M_1 \cap M_2$ (b) Intersaction $\overline{M_1 \cap M_2}$

Figure 16. Illustration for $L_{spec} \cap L_{test}$ and $\overline{L_{spec} \cap L_{test}}$ respectively

Subsequently, we would be able to construct another two automata, called M_3 and M_4, to accept L_{test} $\cap$ $\overline{L_{spec} \cap L_{test}}$ and $\overline{L_{test}}$ $\cap (L_{spec} \cap L_{test}$)and respectively where M_3 = M_2 $\cap$ $\overline{M_1 \cap M_2}$ and M_4 = $\overline{M_2}$ $\cap (M_1 \cap M_2)$. In this case, M_4 which accepts $\overline{L_{test}}$ $\cap (L_{spec} \cap L_{test}$) is an empty string, so we can ignore M_4 As a result, the final automata to accept $(L_{test} \cap \overline{(L_{test} \cap L_{spec}}))\cup(\overline{L_{test}} \cap (L_{test} \cap L_{spec}))$ is M_3 shown in Fig.17.

Because it is not an empty string, the *Transaction* state is incomplete. By the same way, we can check the other states in this ATM specification.

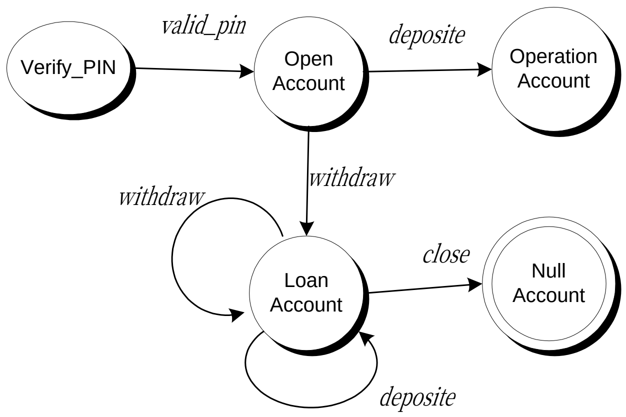


Figure 17. Illustration for $(L_{test} \cap \overline{(L_{test} \cap L_{spec})}) \cup (\overline{L_{test}} \cap (L_{test} \cap L_{spec}))$

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 shown in section 4, so we can check the completeness of OO specification state by state.

6.3.2 Consistency Analysis

In the above section, we have shown that an inconsistent OO specification may contain *stuck* state or *unreachable* state. Also, two algorithms to detect them have been proposed. In the following, we would like to use the *Transaction* superstate again as an example to illustrate the two algorithms. In this case, the *Loan-Account* state which can not reach final state will be shown to be the *stuck* state. Let's consider Fig.15 again. For example, given an invocation sequence `` *valid_pin* • *withdraw* • *deposit* • *withdraw* • *close*" chosen from test data L_{test}, the transition $\delta(Verify\_PIN, vlid\_pin \bullet withdraw \bullet deposit \bullet close)$ will stick to the *state Loan-Account* and will not go to final state.

$$\begin{aligned}\delta(start, invocation\_sequence) &= \delta(Verify\_PIN, valid\_pin \bullet withdraw \bullet deposit \bullet close) \\ &= \delta(\delta(Verify\_PIN, valid\_pin), withdraw \bullet deposit \bullet close)\end{aligned}$$

$$= \delta(\delta(Open\_account, withdraw), deposit \bullet close)$$
$$= \delta(Loan\_account, deposit) \bullet close)$$
=stick in Loan_Account

In the following, we depicts the process of checking consistency by running a test scenario which performs a *withdraw* operation from a savings account (see Table 1).

Table 1: A test scenario for checking consistency

Test Scenario= *insert_card*• *get_bankPIN*• *get_userPIN*• *valid_pin*• *selection*• *withdraw*• *close*• *end_transaction*•*eject_card*			
Transition: $\delta(current\_state, method\_invocation) \to nextstate$			
Current State	*method_invocation*	*Next State*	*Event*
1: Start	insert_card	2: Check_card	Card insert
2: Check_card	get_bankPIN	3.1: Initial	Get bankPIN
3.1: Initial	get_usrPIN	3.2: First_check	Get userPIN
3.2: First_check	valid_PIN	4.1: Open_Account	Vaid PIN entered
4.1: Open_Account	selection	4.2: Operation_Account	Input selection
4.2:Operation_Account	withdraw	4.2: Operation_Account	Withdraw money
4.2:Operation_Account	close	4.4: Null_Account	close the account
4.4: Null_Account	end_transaction	6: Eject_card	Transaction done
6: Eject_card	eject_card	7: End_state	Eject card

7. Conclusion

In this paper, we provide an OO testing framework to test inconsistency and incompleteness of OO software specification early in the software development. Our approach is to build a state-based model and then to test this model to ensure that the model match the desired properties: completeness and consistency. To accomplish this, we build a new object-oriented testing framework from specification to test data generation. We successfully apply finite automata to construct this OO testing framework in which object-oriented specifications are represented with RSML and test data are generated in the form of regular expression. 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. This issue involves improving the ability of finite automata. As noted, 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. In the future research, we will try to use Petri-net to strengthen this OO testing framework.

Reference

[1] Sergio Antoy and Dick Hamlet, ``Automatically Checking an Implementation against Its Formal Specification," *IEEE Transaction on Software Engineering*, Vol. 26, No. 1, pp. 55-69, Jan. 2000.

[2] M. Lijter, S. Meyers, and S.P. Reiss. "Support for maintaining object-oriented programs," *IEEE Transactions on Software Engineerin*g, 18(12), pp. 1045-1052, Dec. 1992.

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

[4] Mats P.E. Heimdahl and Nancy G. Leveson, ``Completeness and Consistency in Hierarchical State-Based Requirements," *IEEE Transaction on Software Engineering,* vol. 22, no. 6, pp. 363-377, June, 1996.

[5] Elaine Weyuker, `` Axiomatizing software test data adequacy," *IEEE Transaction on Software Engineering,* SE-12, 12, pp. 1128-1138, Dec. 1986.

[6] Elaine Weyuker, `` The evaluation of program-based software test data adequacy criteria," *Communication ACM,* 31, 6, pp. 668-675, June, 1988

[7] E. J. Weyuker and B. Jeng, ``Analyzing Partition Testing Strategy," *IEEE Transaction on Software Engineering,* vol. 17, no. 7, pp. 703-711, July, 1991.

[8] Ana Cavalcanti and David A. Naumann, ``A Weakest Precondition Semantics for Refinement of Object-Oriented Programs," *IEEE Transaction on Software Engineering,* vol. 26, No.8, pp. 713-727, Aug. 2000.

[9] Deway E. Perry and Gail E. Kaiser, ``Adequate Testing and Object-Oriented Programming," *Journal of Object-Oriented Programming,* pp. 13-19, Jan. 1990.

[10] Chi-Ming Chung, Timothy K. Shih, Chun-Chia Wang, and Ming-Chi Lee, ``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, Jan. 1997.

[11] C.M. Chung and M.C. Lee, ``Object-Oriented Programming Testing Methodology," *International Journal of Mini and Microcomputer,* vol. 16, no. 2, pp.773--81, 1994 .

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

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

[14] R.Duke, P.king, G.Rose, G.Smith, ``The Object-Z Specification Language, Version 1 Technical Report 91-1," Software Verification Research Centre, Department of Computer Science, University of Queensland, 1991.

[15] Ming-Chi Lee, ``An Object-Oriented Testing Framework Specified with Z Notation," Tamkang Journal of Science and Engineering, vol. 2, No. 1, pp. 11-22, 1999.

[16] R. Lutz, ``Targeting Safety-Related Errors During Software Requirements Analysis," *Proc. First ACM SIGSOFT Symp. The Foundations of Software Engineering,* pp.102-111, 1993.

[17] Phil Stocks and David Carrigton, ``A Framework for Specification-Based Testing," *IEEE Transaction on Software Engineering*, vol. 22, no. 11, pp. 777-793, September, 1996.

[18] Ian J. Hayes, ``Specifications Directed Module Testing," *IEEE Transaction on Software Engineering*, VOL. SE-12, no. 1, pp. 124-133, Jan. 1986.

[19] B.W. Boehm, ``Verifying and validating software requirement specification and design specification," *IEEE Software*, 1: 61-72, 1986.

[20] J. Rushby and F. von Henke, ``Formal Verification of Algorithms for Critical Systems," *IEEE Trans. on Software Engineering*, vol. 19, no. 1, pp. 13-23, Jan. 1993.

[21] H. Richel and A.P. Ravn, ``Requirements Capture for Computer Based Systems,"

Technical Report ID/DTH HR 2/2, Technical Univ. of Denmark, pp. 111-120, Oct. 1990.

[22] A.P. Ravn and H. Richel, ``Requirements Capture for Embedded Real-Time Systems," *IMACS Symp. MCTS*, pp. 33-44, 1991.

[23] J. Atlee and J. Gannon, ``State-Based Model Checking of Event-Driven System Requirements," *Proc. ACM SIGSOFT '91 Conf. Software for Critical Systems Software Engineering Notes*, vol. 16, No. 5, pp. 22-33, 1991.

[24] J.R. Burch, `` Symbolic Model Checking: 10^{20} States and Beyond," *Proc. Fifth Ann. Symp. on Logic in Computer Science*, pp.23-30, June 1990.

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

[26] D. Harel and A. Naamad, ``The STATEMATE Semantics of State-Charts," Technical Report CS95-31, The Weizmann Institute of Science, pp. 34-45, Oct. 1995.

[27] T.J. McCabe, ``A Software Complexity Measure," *IEEE Transaction on Software Engineering,* 2(6) pp. 308-320, 1976.

[28]G. Booch *Object-Oriented Analysis and Design with Application*, The Benjamin/Cummings Publishing Company, Inc., second edition, pp. 105-125, 1993.

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

Appendix --- Algorithms for three regular expressions.

Appendix outline the data structures and algorithms for three primitive regular expression properties: *Complement, Intersection* and *Union*. They are used to help test the OO specification completeness shown in Section 5. They are described by C++ languages.

```
Struct State{
        String      name;
        Int          state_number;
}

struct Event {
        State      source;
        State      dest;
        EventArray    methods;
}

Class Automata{
      Private:
              Event    invoke;
              State      initial,final;
              StateList      set;
              Char trans_table[256][256];
      Public:
              Initial_state(State  *initial)  {ifstream  ipt("input.dat");  ipt.read((State*)
                                    &initial);}
              Final_state(State   *final)   {ifstream   ipt("input.dat");   ipt.read((State*)
                                    &final);}
              Void transition(State, Event);
}

Automata:: void transition (State state, Events invoke)
        for (int i=0; i< size_of(state.set);++i)
            for (int j=0; j< size_of(invoke.methods);++j)
                 if (( state.set[i]==true)&& (invoke.dest[j]==true))
                     trans_table[i][j]=state.state_number;
}

Automata *Union(Automata M1,M2)
{
     Automata *new; new= (Automata *) malloc (size_of(Automata));
     for (i=0; i< size_of (M1.set+M2.set);++i)
         new->set=add(M1.set,M2.set);
```

```
            for (j=0;j<size_of(M1.invoke.methods+M2.invoke.methods);++j)
                  new->invoke = sum (M1.invoke, M2.invoke);
            transition (new->set, new->invoke);
            return(new); }

Automat *Intersection (Automata M1,M2)
{
        Automata *new; new=(Automata *) malloc(size_of(Automata));
        for (i=0; i< size_of (M1.set+M2.set);++i)
              if (M1.set.name==M2.set.name)
              new->set=extract(M1.set,M2.set);
            for (j=0;j<size_of(M1.invoke.methods+M2.invoke.methods);++j)
                  if (M1.set.invoke==M2.set.invoke)
                  new->invoke = common (M1.invoke, M2.invoke);
            transition (new->set, new->invoke);
            return(new);
}

Automata *Complement(Automata M)
{
    Automata *new; new=(Automata *) malloc(size_of(Automata));
    swap(&M.initial, &M.final);
    new->set=&M.set;
    new->invoke=&M.invoke;
    transition(new->set, new->invoke);
    return(new);
}
```