JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2008

Vol. 9, No. 3, May–June 2010

Extracting State Models for Black-Box Software Components

Rajiv Ranjan Suman, Rajib Mall
Department of Computer Science & Engineering,
Indian Institute of Technology, Kharagpur, West Bengal- 721302, India

Srihari Sukumaran, Manoranjan Satpathy
GM India Science Lab, Bangalore -560066, India

We propose a novel black-box approach to reverse engineer the state model of software components. We assume that in different states, a component supports different subsets of its services and that the state of the component changes solely due to invocation of its services. To construct the state model of a component, we track the changes (if any) to its supported services that occur after invoking various services. Case studies carried out by us show that our approach generates state models with sufficient accuracy and completeness for components with services that either require no input data parameters or require parameters with small set of values.

1 INTRODUCTION

In the component-based software development paradigm, a large software is built by assembling pre-built and independently developed "plug and play" type of software parts, called software components. The desired system behavior is achieved through the collaborative actions of the assembled components. A component has certain prespecified contractual obligations that comprise the services it needs to provide. These services of a component are usually expressed as a set of named operations (or methods) using some Interface Definition Language (IDL). Most often, only the executable code and the IDL specifications of the components are available to the application programmer. In other words, the design and implementation aspects of a component are completely hidden and a component is used as a black-box by the component integrator.

A component is a generic term and is often designed and implemented as a single class or a collection of classes. A component may sometimes not be based on any class at all (in case of procedural implementation), or it may even integrate many smaller components. In spite of the apparent diversity in component implementations, as far as state modeling is concerned, a component in the component paradigm can be considered analogous to an object in the object paradigm since both a component and an object can be considered as black boxes that store some data and provide some externally visible behavior.

A state model of a component is a behavioral model that depicts the different states that the component may assume and transitions among the states that may occur in response to the stimuli received from its environment. Likewise, an object in an object-oriented system may transit through various states in response to its method invocations. Externally, the state model of a component is visualized in terms of the state-based behavior of the component as a whole rather than in terms of the individual objects that the component may be composed of. Due to this close analogy between an object and a component, as far as state-based behavior is concerned, in the rest of this paper, we use the terms *service* and *method* as well as *component* and *object* interchangeably.

An important advantage of the component paradigm is that a component in an application can effortlessly be replaced any time by another functionally equivalent component. For example, many present day software products such as Microsoft Internet Explorer allow upgrades (patches) to be downloaded on the fly. Each downloaded patch may change one or more components of the application. After every such change to a component of a critical application, regression testing of the application needs to be carried out to ensure that the various features continue to work satisfactorily even after component upgradation. The regression test suite is a subset of the existing test suite that is selected based on the relevance of a test case to the change. Selection of regression test cases for component-based software is considered a challenging task due to several issues that need to be handled. One particularly vexing problem is the following. Components often have significant states. However, components are usually not accompanied with their state models. In the absence of a state model, it is difficult to test the state behavior of a component. Satisfactory regression testing of a component-based software therefore is a challenging research problem[11, 12, 13].

State-based bugs are difficult to detect using traditional testing techniques[2]. There are several types of state-based bugs that are usually targeted by the state-based testing techniques. Examples of a few important types of state-based bugs are the following. A system might behave correctly to a user's requests in only some of the states but not in other states. It is also possible that the system may not transit to some required state even when all necessary conditions are satisfied (missing transitions) or may have improper transitions (sneak transitions) to certain states [2]. State-based software testing has therefore been accepted as a crucial type of testing that can help detect such insidious bugs. State models form an important basis for state-based testing in the component paradigm[20]. State coverage and transition coverage are two popular state-based testing techniques [2].

The services offered by components are usually documented as interface specifications. It, however, is the responsibility of the component integrator to ensure that the components are trustworthy. In addition to validating the functional behavior, dynamic behavior of the components need to be validated. As components are available to an application programmer (a component assembler) as black-box units with only their documented interface specifications, the state model of the components are not known. Therefore, it is desirable to develop a technique to extract the state model of a component from its observable behavior. In this paper, we propose a technique to extract the state model of a component from an examination of its external behavior. A preliminary position paper for this work was reported in [22].

Components are increasingly being used to build embedded systems, distributed control applications, and several types of real-time systems. These applications mandate ensuring high degrees of reliability, safety, and security. In this light, state model-based testing assumes importance. Besides its use in testing, the extracted state model of a component has several other applications as well. These include understanding the state-based behavior of a component and re-engineering of a component to meet new requirements or constraints. A state model can also be used to estimate the complexity and effort needed for state-based testing, as well as to estimate the reliability of a component. Availability of a state model of the component can also be useful for system-level impact analysis.

Several formalisms are at present being used to represent state models. These include finite state machines (FSMs)[9], statecharts [4], etc. These formalisms are of varying complexities and expressive powers[10]. However, out of the available formalisms, FSMs are considered to be more intuitive and are therefore extremely popular. Keeping these factors in view, we attempt to recover FSM-based state models of components in our work.

The rest of this paper is organized as follows. In section 2, we present our methodology for extraction of the state model of a component. Section 3 presents the results of using our methodology of state model extraction on few sample problems. In section 4, we compare our work with related work. Finally, section 5 concludes this paper.

2 OUR STATE MODEL EXTRACTION METHODOLOGY: COSMOD

In the following section, we first present some basic concepts related to our methodology. This is followed by an overview of our proposed approach. We have named our algorithm COSMOD (reverse engineering COmponent State MODel from its external behavior). Finally, the pseudo code of COSMOD is presented with an example to illustrate its working.

Basic Concepts and Definitions

In this section, we present a few definitions and terminologies that we shall use in the rest of this paper.

Finite State Machine (FSM): An FSM[9, 10] is popularly used to model the behavior of a system with discrete inputs and outputs. An FSM consists of a finite number of internal states, transitions among these states, and actions. At any moment, it remains in any one of these states. The initial states are a subset of all of its states.

An FSM also has a finite (possibly empty) set of final states. A state of an FSM is a result of all of its past inputs. Formally, a deterministic acceptor FSM is a five tuple (Q, I, F, Σ, δ) where

1. Q is a finite, non-empty set of elements called states.
2. $q_0 \in$ Q is a distinguished state, called the initial state.
3. F $\subseteq$ Q, is a (possibly empty) set of the final states.
4. Σ is a finite, non-empty set of input symbols or input events (called alphabet) associated with the state transition of the system
5. δ is a partial function called transition function that maps a state-symbol (or state-event) pair to at most one state, i.e. δ : Q X $\Sigma \rightarrow$ Q.

Non-deterministic acceptor FSM: It is the same as its deterministic counterpart except that the transition function δ maps a state-symbol (or state-event) pair to a subset of Q, i.e. δ : Q X $\Sigma \rightarrow 2^Q$, where 2^Q is the power set of set Q.

Deterministic FSM with Guard Conditions: Sometimes guard condition*s* are associated with the transitions of an FSM. In this case, a transition would fire only when its input event occurs and the associated guard condition evaluates to `true`. Formally, a deterministic FSM with guard conditions may be defined as δ : Q X Σ X G $\rightarrow$ Q, where G is the set of guard conditions which are usually expressed as boolean expressions. This is interpreted as when the FSM is in state $q_1 \in$ Q and receives an input symbol (or event) e $\in \Sigma$, then if the guard g $\in$ G is true then the FSM transits to a state $q_2 \in$ Q.

Terminology used in COSMOD:

(1) methodSet of a component: Services of a component may be described in terms of method names in its IDL specification. So, for notational convenience, we use the term `method` for a `service` described in the IDL specification of a component. We denote all the methods of a component as its `methodSet`.

(2) activeMethods: At any point of time during the lifetime of a component, only a subset of all the methods specified in its interface may be meaningfully executed, that is, only a subset of all of its services are active. We use the term *activeMethods* to denote the set of active methods. The set of remaining methods are therefore called inactive methods at that point of time. Invocation of an inactive method **m** would either generate an exception or display message such as “method not active in the present state”. For example, invocation of the method pop() of a *stack* component when the stack is empty would result in producing an error message. In a bank ATM component, the method performTransaction() cannot be invoked when the ATM is in a state where it is waiting for the customer to enter the PIN.

(3) State S of a component and **activeMethods**[S]: At any point of time, we characterize the state *S* of a component in terms of *activeMethods* at that time. The component would stay in state *S* as long as its *activeMethods* do not change. As a result, we call such *activeMethods* as *activeMethods*[S] which denotes the methods that can be meaningfully executed in state S. Whenever a service of a component is requested, its *activeMethods* may change, that is, some methods in the set *activeMethods* may become inactive (non-callable) and removed from this set, and some other methods that were inactive before the invocation of the service may become active now and are added to the set *activeMethods*. Change in *activeMethods* corresponds to a state transition. After execution of a method, if we observe that the *activeMethods* has changed, we can conclude that component has transited to a different state.

(4) targetState[S, m(P)] and **source state S**: This is the state to which a component transits when its method m is invoked with parameter value-set P in its state S. We call it the target state of S for method invocation m(P). S is called the source state of this transition. The component may transit to different states on invocation of method m with different values of P.

(5) Symmetric states, symmetricStateList$[S_v]$ and **equivalent states**: States $S_1, S_2, ..., S_n$ are called symmetric if they have identical *activeMethods,* that is, *activeMethods*$[S_1]$ = *activeMethods*$[S_2]$ = ... = *activeMethods*$[S_n]$. Notationally, we use the term *symmetricStateList*$[S_v]$ to denote a list of states that are symmetric to state S_v. Due to the existence of symmetric states in the state model, *activeMethods* at the different states of the component are not all distinct. Even when the *activeMethods* are the same in two different situations, a component may actually be in different states. We consider the symmetric states in order to identify even those states in the state model that are not distinguishable from other states only on the basis of *activeMethods*, otherwise only a partial state model may be generated. We discuss this issue further in section 2.

Two symmetric states S_1 and S_2 (*activeMethods*$[S_1]$ = *activeMethods*$[S_2]$) are considered equivalent if their target states are symmetric for each of their active methods, that is, if *targetState*$[S_1,\ m(\text{P})] = S_x$, *targetState*$[S_2,\ m(\text{P})] = S_y$, and states S_x and S_y are symmetric, for each $m \in$ *activeMethods*$[S_1]$ and for all valid values of the parameter set P. States S_1 and S_2 are considered non-equivalent if there exists at least one value of m or P for which the target states S_x and S_y are not symmetric. Intuitively, two states are equivalent if the component exhibits same behavior in the two states.

In the state model generated by COSMOD, two or more symmetric states exist as different states only if they are non-equivalent states. A symmetric state is maintained as a separate state during the process of generation of the state model as long as it is not identified as equivalent to an older state. Once it is determined that two symmetric states are equivalent to each other, they are merged together. In Figure 5, states S_1 and S_2 are symmetric but different (non-equivalent) states. They are symmetric states because *activeMethods*$[S_1]$ = *activeMethods*$[S_2]$ = {create(), ready()}. They are different states because on invocations of the method ready() in states S_1 and S_2, the component transits to states S_3 and S_4 respectively, but states S_3 and S_4 are not symmetric states because *activeMethods*$[S_3]$ = {create(), ready(), swap()}, whereas *activeMethods*$[S_4]$ = {create(), ready(), swap(), finish()}. Likewise, states S_3 and S_6 are symmetric (but different) states. States S_4, S_5, S_7 and S_8 are also symmetric states.

(6) Naming convention for states: COSMOD assigns numbers 0, 1, 2, ... to the states of the component in the order in which they are discovered. State numbers are used to identify the states for various purposes. However, for the sake of readability, we prefix the letter 'S' before the state numbers and we use the symbols S_0, S_1, S_2, etc to refer to the states 0, 1, 2, etc.

(7) Initial state of a component: The initial state of a component is denoted by S_0. This is the state that a component assumes just after it has been created/instantiated and no service of the component has been invoked.

(8) End (or final) state of a component: This is a state from which no outbound transition is identifiable. There may be zero, one or more than one end states of a component. For example, the end states of a `purchaseOrder` component may be the states corresponding to `fulfilled order` and `rejected order`. The `scheduler` component of Figure 5 does not have any end state.

(9) stateSet: It is a set of all the states of the component that have been discovered at any point of time during the execution of COSMOD..

(10) parameterSet [m]: It stores the different combinations of the parameter values generated for invocations of method m. In our present version of COSMOD, we generate the parameter values randomly. However, more effective algorithms may be used to generate the parameter values.

(11) SRMS[S]: SRMS stands for `State Recovery Method Sequence`. Before the services of a component can be invoked at a specific state, the component must be made to assume that state. This can be done by executing a certain sequence of methods (with suitable parameter values) in the initial state S_0 of the component. SRMS[S] denotes the specific sequence of methods, along with their input parameter values, that are to be executed in state S_0 in order to make the component transit to state S.

(12) transitionList [S, m]: It is a list of pairs $[(P_{i1}, S_{i1}), (P_{i2},S_{i2}), ..., (P_{in},S_{in})]$. This list stores the information that if the method m is invoked in state S with parameter value sets $P_{i1}, P_{i2}, ..., P_{in}$ then this makes the component transit to states $S_{i1}, S_{i2}, ..., S_{in}$ respectively. The variable **transitionList** (without S and m) denotes the set of all such lists of state-method pairs.

(13) cycleMethodSequence: This is the sequence of methods (along with their parameter values) found as the labels of transitions (edges in a graphical representation) when a cycle of the FSM (the state model under construction) is traversed. We consider only loops (cycle of length 1) and simple cycles (those cycles in which only one vertex (the beginning and ending vertex) appears twice and all other vertexes appear only once). The state nearest to the starting state S_0 is considered as the first state of a cycle, next nearest state is the second state and so on. Any tie on distances from state S_0, may be resolved arbitrarily. As a result, the first method of a *cycleMethodSequence* is the method present as the label of the transition from the first state to the second state of the corresponding cycle.

(14) Repetition factor r: This decides the number of repeated executions of the *cycleMethodSequences* of the partly constructed FSM. The value of r may be fixed heuristically or empirically.

(15) State model: By state model of a component, we mean the FSM model of its state behavior which is generated by the COSMOD algorithm. This state model may be a `deterministic FSM with guard conditions` (if guards are identifiable) or it may be a `nondeterministic FSM` when guards are not identifiable. Next state of a component is determined by its present state, method invoked together with their input parameters, and the guard condition (if any).

An Overview of Our Approach

The COSMOD algorithm to extract the state model of a component is based on observing its behavior in its different states and it involves performing three basic activities which are as follows:

(A) Invocation of active methods at a state, the function call **invokeActiveMethods**(S_u)

(B) Confirming or merging a potential state, the function call **confirmOrMergeState**(S_u)

(C) Repeated executions of *cycleMethodSequences* of the FSM, the function call **executeCycleMethodSequence**(r)

By an "activity" we mean a set of steps to be carried out by our algorithm. On page 86, we present the pseudo code of COSMOD algorithm in which function calls invokeActiveMethods(S_u), confirmOrMergeState(S_u) and executeCycleMethodSequence() correspond to performing the activities (A), (B) and (C), respectively.

Activities (A) and (B) are performed in sequence first for the initial state S_0 and subsequently for all the discovered states in the order in which they are discovered. The states are numbered sequentially as S_1, S_2, ...etc. As shown in the pseudo code of COSMOD (on page 86), the activity (B) is performed only for those states that are marked as "potential state". After carrying out the activity (B) for a state S_u, the state S_u is either merged to an earlier state $S_i \in \mathit{symmetricStateList}[S_u]$ that is already marked as a "confirmed state", or S_u itself is made a "confirmed state".

Performing activities (A) and (B) at every state, some or all the states of the component are discovered. To discover the remaining states or transitions (if any), COSMOD performs the activity (C). After performing the activity (C), COSMOD repeats the activities (A) and (B) on all the states generated due to the most recent application of activity (C), followed by performing activity (C) again on the FSM. Construction of FSM is complete when no "confirmed state" is generated after performing the activity (C).

Algorithm COSMOD;
Input: (1) IDL specifications and executable code of a component (2) Repetition factor r
Output: An FSM representing the state model of the component

```
 1: mark $S_0$ as "unexplored state" and "confirmed state"
 2: SRMS[$S_0$] ← [ ]; $stateSet$ ← {$S_0$}
 3: $activeMethods[S_0]$ ← findActiveMethods($S_0$)
 4: repeat
 5:     while $stateSet$ has an "unexplored state" do
 6:         $S_u$ ← next "unexplored state" of $stateSet$
 7:         InvokeActiveMethods($S_u$); // perform activity (A)
 8:         if $S_u$ is marked as "potential state" then
 9:             confirmOrMerge($S_u$); // perform activity (B)
10:         end if
11:     end while
12:     executeCycleMethodSeq($r$); // perform activity (C)
13: until there is no "unexplored state" in $stateSet$
```

Function invokeActiveMethods(S_u); *// This implements activity* (A)
Input: An unexplored state S_u of a partly constructed FSM of a component
Output: Modified FSM with transitions of S_u added to it

```
1: Instantiate component to state $S_0$
2: for each method $m$ in activeMethods[$S_u$] do
3:     $parameterSet[m]$ ← generateParameterCombinations($m$)
4:     for each parameter combination in $parameterSet[m]$ do
5:         executeMethodSeq(SRMS[$S_u$]) // set the component to state $S_u$
6:         next state $S_v$ ← execute($m, P_v$)
7:         processAndSaveState($S_u, S_v, P_v, m$)
8:     end for
9: end for
```

Function processAndSaveState(S_u, S_v, P_v, m);
Input: A newly generated state S_v of a partly constructed FSM
Output: Various bookkeeping data about the state S_v

```
1: markState($S_v$, "unexplored state"); addToSet($S_v, stateSet$)
2: $transitionList(S_u, m)$ ← createStateTransition($S_u, S_v, m, P_v$)
3: SRMS[$S_v$] ← append(SRMS[$S_u$], ($m, P_v$))
4: $activeMethods[S_v]$ ← findActiveMethods($S_v$)
5: $symmetricStateList[S_v]$ ← markConfirmedOrPotentialState($S_v$)
```

Function confirmOrMergeState(S_u); *// This implements activity* (B)
Input: A "potential state" S_u of a partly constructed FSM of a component
Output: Modified FSM with state S_u either made a "confirmed state" or merged
to a symmetric state already marked as "confirmed state"

```
 1: for each state $S$ in symmetricStateList[$S_u$] do
 2:     $toBeMerged$ ← TRUE
 3:     for each method $m$ in $activeMethods[S_u]$ do
 4:         for each parameter combination $P$ of $m$ do
 5:             if NOT symmetric($targetState[S, m(P)]$, $targetState[S_u, m(P)]$) then
 6:                 $toBeMerged$ ← FALSE; break and continue the outermost $forloop$
 7:             end if
 8:         end for
 9:     end for
10:     if $toBeMerged$ == TRUE then
11:         mergeState($S_u$,$S$); break
12:     end if
13: end for
14: if $toBeMerged$ == FALSE then
15:     markState ($S_u$, "confirmed state")
16: end if
```

```
Function executeCycleMethodSequence(r); // This implements activity (C)
Input: (1) A partly constructed FSM of a component (2) Repetition factor r
Output: Same or modified FSM with zero or more new "unexplored states"
 1: $cycleList \leftarrow$ enumerate cycles in the FSM
 2: for each cycle $cyc$ in $cycleList$ do
 3:    Let the state sequence of $cyc$ be $< T_1, T_2, ...., T_x, T_1 >$ where $T_1$ is the first state of the sequence and let the
       $cycleMethodSequence$ of $cyc$ be $m1(P_1), m2(P_2), ...., m_x(P_x)$, where $x$ is the cycle-length, $1 \leq x \leq n$, and $n$
       is the maximum number of states in the FSM
 4:    executeMethodSeq(SRMS[$T_1$])  // set the component to state $T_1$
 5:    for $count$ = 1 to $r$ in step 1 do
 6:       for $i$ = 1 to $x$ in step 1 do
 7:          next state $S_v \leftarrow$ executeMethod($m_i(P_i)$)
 8:          $activeMethods[S_v] \leftarrow$ findActiveMethods($S_v$)
 9:          if symmetric($S_v, T_{i+1}$) then
10:             continue  // transition to next state in the cycle, hence start next iteration of
                          // the inner for-loop at line 6
11:          else if symmetric($S_v, S_a$), where state $S_a \in stateSet$ and $S_a \neq T_{i+1}$ then
12:             if pair ($P_i, S_a$) is already in $transitionList[T_i, m_i]$ then
13:                // i.e, if there is already an edge labeled $m_i(P_i)$ from state $T_i$ to state $S_a$ then
14:                discard state $S_v$  // as this state is very likely to be merged later
15:             else
16:                processAndSaveState($T_i, S_v, P_i, m_i$); // a new edge detected from existing state $T_i$
                                                            // to existing state $S_v$
17:                break the two inner for-loops and start next iteration of the outermost for-loop at line 2
18:             end if
19:          else
20:             markState($S_v$,"confirmed state")  // $S_v$ not symmetric to any existing state, hence
21:             processAndSaveState($T_i, S_v, P_i, m_i$);  // a new ''confirmed state'' $S_v$ was discovered
22:             break the two inner for-loops and start next iteration of the outermost for-loop at line 2
23:          end if
24:       end for
25:    end for
26: end for
```

In the following three consecutive subsections, we discuss the activities (A), (B) and (C) in detail.

Activity (A): Invocation of Active Methods

This activity involves invoking each active method of the component in a state S_u which is called the state being explored. Each method is invoked with different values of parameters in state S_u. COSMOD assumes that the component transits to state S_v after invocation of a method m. Therefore, it always creates a new state S_v and a new transition from state S_u to state S_v after invocation of a method m. If the $activeMethods[S_v]$ is different from the $activeMethods[S_i]$ for all the earlier states S_i (i = 0 to v -1), then the component has undoubtedly undergone a state transition and COSMOD marks S_v as "confirmed state" and saves it. However, some of the method invocations may not result in state transition. In that case $activeMethods[S_u]$ and $activeMethods[S_v]$ will be the same. But, matching active methods may also be due to state S_v being a state symmetric to (but different from) state S_u. Similarly, a method invocations may result in transition to an earlier state S_i where i may have any value from 0 to u-1 ($i = u$ corresponds to no state transition). In such cases, $activeMethods[S_u]$ and $activeMethods[S_i]$ will be the same. In this case also, matching active methods may be due to state S_v being a state symmetric to (but different from) state S_i.

Therefore, in situations where *activeMethods*$[S_v]$ is same as any one of the *activeMethods*$[S_i]$ (i= 0 to v -1), that is, state S_v is symmetric to an earlier state S_i (where S_i is one of the states from S_0 up to S_u), it is to be tested whether S_v is equivalent (to be merged with) or only symmetric (but not equivalent) to the earlier state S_i. COSMOD resolves this ambiguity later by performing activity (B) on state S_v just after completion of activity (A) on it, that is, after discovering all possible transitions from state S_v. For this, it marks such state S_v as "potential state" during activity (A) indicating that activity (B) needs to be performed on this state. It adds all the states S_i (i= 0 to v -1) to the *symmetricStateList*$[S_v]$ for which *activeMethods*$[S_u]$ and *activeMethods*$[S_i]$ are the same.

Activity (B): Confirming or Merging a Potential State

Activity (B) compares the behavior (compares target states on invocations of the same active method with same input parameters) of the component in states S_u and in all the states S in *symmetricStateList*$[S_u]$ where state S has already been marked as a "confirmed state". If the state S_u is found equivalent (refer to section 2 for definition) to a state S, then it is merged with state S. Otherwise, its mark is changed from "potential state" to "confirmed state". Thus, activity (B) either converts a "potential state" to a "confirmed state" or merges it to an equivalent symmetric state that is already marked as "confirmed state". Therefore, after determining the transitions from a state S_u by performing activity (A) on it, COSMOD checks how it is marked. If S_u is marked as "confirmed state", then activity (B) is skipped. On the other hand, if it is marked as "potential state", then COSMOD calls the function **confirmOrMergeState(S_u)** to perform activity (B) that resolves whether the state S_u is a different state or equivalent to one of its symmetric states:

Activity (C): Repeated Execution of *cycleMethodSequences*

We discuss (a) the motivation and (b) the process of performing the activity (C) of COSMOD.

Motivation for activity (C)

As per the assumptions of COSMOD, a transition from state S_u to state S_v is externally visible only when S_u and S_v are non-equivalent states. For certain states, a single method call may not cause an externally visible state transition to occur. Repeated calls to one or more methods with the same or different parameter values may cause the transition to be taken. In the following, we discuss some examples to illustrate such situations.

Let us consider the case of *libraryMember* component of a *Library Information System*. Its partial state model has been shown in Figure 1. On invocation of the method borrowBook(*book*) in state S_2, the component remains in the same state as long as the number of books borrowed by the member is less than the maximum number of books that can be borrowed. After that, next execution of borrowBook(*book*) method causes the component to transit to state S_3. Suppose that the member's

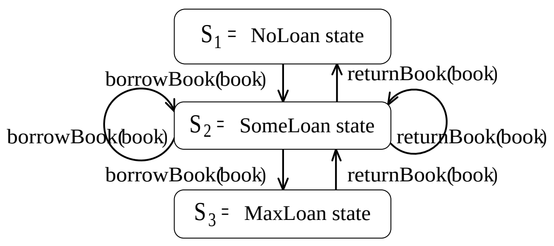

Figure 1: Partial state model of a *library-Member* component of a library information system

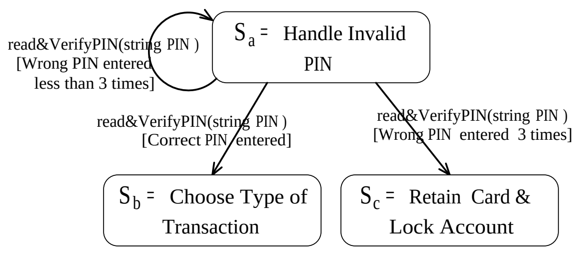
read&VerifyPIN(string PIN)
[Wrong PIN entered
less than 3 times]
Sa = Handle Invalid PIN
read&VerifyPIN(string PIN)
[Correct PIN entered]
read&VerifyPIN(string PIN)
[Wrong PIN entered 3 times]
Sb = Choose Type of Transaction
Sc = Retain Card & Lock Account

Figure 2: Partial state model of the *transaction session* component of a bank ATM system

loan limit is n books (where $n > 2$). When the library member borrows the first book, the *libraryMember* component transits from state S_1 to state S_2 because the method returnBook($book$) does not become active until the first book is borrowed. A single execution of the method borrowBook($book$) in state S_2 of the component would not make it transit to state S_3. One must execute the borrowBook($book$) method n-1 times in state S_2 to drive the component to the state S_3, which is the state of maximum loan limit. Here, the parameter values (e.g., book's accession number) in the repeated call to borrowBook($book$) could be any valid parameter value, since every time the member borrows a book, the number of borrowed books increases by one. It is worth noting that the state S_3 is different from the states S_1 and S_2 as the method borrowBook($book$) is not allowed (i.e. inactive) in state S_3, whereas it is allowed in states S_1 and S_2.

Figure 2 illustrates the fact that a *bank ATM* component may transit to different states on invocation of the same method with different sets of valid parameter values. However, invocation of the method read&VerifyPIN(string PIN) at state S_a with wrong value of the parameter PIN causes formation of a self loop (a cycle of length 1), that is the state remains unchanged. It is required to invoke the method read&VerifyPIN(string PIN) three times with wrong value (any arbitrary value, same of different, except the correct value) of PIN in order to transit the component to state S_c. These two examples indicate that we need to repeatedly execute the methods along the cycles (that is, the *cycleMethodSequence*) of an FSM in order to discover new states, such as the state S_c of Figure 2 and state S_3 of Figure 1.

Figure 3 shows the partial state model of a power window controller (PWC) which is obtained after performing the activities (A) and (B) of the COSMOD algorithm (given on page 86). That is, by completely executing the *while-loop* (lines 5 to 11) with only one iteration of the outer *repeat-until loop* (lines 4 to 13) and without performing the activity (C), i.e. without executing the function call executeCycleMethodSeq(r) at line 12. Comparing this model with that in Figure 4, it can be observed that several states and transitions are missing in this state model. This happens due to existence of global variables. Many methods become active and inactive depending on the values of these global variables. The value of any local or global variable or constant is not visible in case of a black-box component.

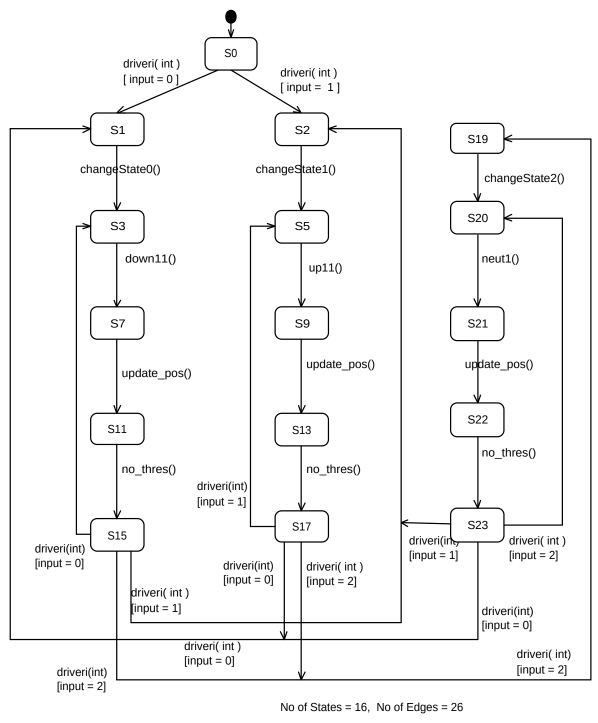

Figure 3: Partial state model of a power window component

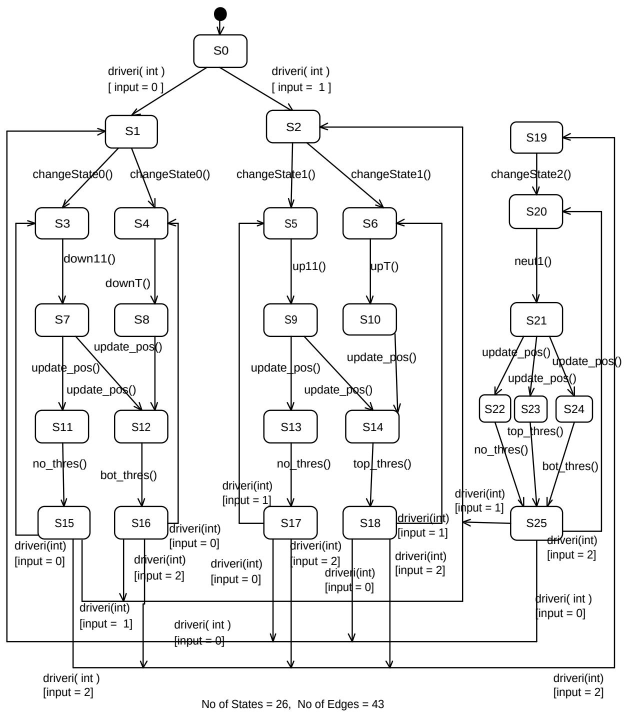

Figure 4: State Model of PWC component constructed by applying the COSMOD algorithm

However, executions of the valid method sequences of the component repeatedly is likely to cause these variables to assume values that they may not assume when the method-sequences are invoked just once in a state. This, in turn, may result in a new set of methods to become active (i.e. *activeMethods* of the component changes) indicating the discovery of a new state.

The process of activity (C)

COSMOD calls the function **executeCycleMethodSequence(r)** to perform activity (C). During activity (C), it enumerates all the cycles in the FSM and determines the *cycleMethodSequence* for each cycle. The first method of a method-sequence is the method causing the transition from the first state (node) to the second state of the corresponding cycle, where the first state is the state that is nearest to the starting state S_0. For example, $< S_3, S_7, S_{11}, S_{15}, S_3 >$ is a cycle in the FSM of Figure 3. We list the states in a cycle within angular brackets. The first state of this cycle is S_3 and its *cycleMethodSequence* = down11(), update_pos(), no_thres(), driveri(int), which is a sequence of four methods having the method down11() as its first method. Each *cycleMethodSequence* is executed r times as an attempt to discover as many "confirmed states" as might be existing in the state model, where r is the repetition factor chosen to uncover the states. Its value can be set (as input to COSMOD) by the user either arbitrarily or intuitively depending on the information about the application of the component.

COSMOD finds cycles in the FSM using the algorithm of [18], though the algorithm of [19] could also be used. It considers the cycles in increasing order of their lengths (number of edges in the cycle). That is, it considers the self loops first, then the cycles containing two edges are considered, and so on. It repeatedly (r times) executes the methods in the *cycleMethodSequences*.

As an illustrative example, there are 15 cycles in the FSM of Figure 3. There is a cycle of states $< S_3, S_7, S_{11}, S_{15}, S_3 >$ that forms the method-sequence {down_11(), update_pos(), no_thres(), driveri(int)}. Another cycle of states $< S_5, S_9, S_{13}, S_{17}, S_5 >$ forms the method-sequence {up_11(), update_pos(), no_thres(), driveri(int)}. Similarly, the cycle $< S_2, S_5, S_9, S_{13}, S_{17}, S_1, S_3, S_7, S_{11}, S_{15}, S_2 >$ generates the method-sequence {changeState1(), up_11(), update_pos(), no_thres(), driveri(int), changeState0(), down_11(), update_pos(), no_thres(), driveri(int)}. Repeated executions of *cycleMethodSequence* for cycles $< S_3, S_7, S_{11}, S_{15}, S_3 >$ and $< S_5, S_9, S_{13}, S_{17}, S_5 >$ of Figure 3 discovers the state S_{12} and S_{14} (shown in Figure 4), respectively, both of which are “confirmed states”.

Iterating the executions of the *cycleMethodSequence* for cycle $< S_1, S_3, S_7, S_{11}, S_{15}, S_{19} S_{20}, S_{21}, S_{22}, S_{23}, S_1 >$ creates a new transition from state S_1 to S_4 and doing so along the cycle $< S_2, S_5, S_9, S_{13}, S_{17}, S_{19} S_{20}, S_{21}, S_{22}, S_{23}, S_2 >$ discovers the transition from state S_2 to S_6. Repeated executions of *cycleMethodSequences* for the remaining cycles do not generate any new state or transition. We then repeat the activities (A) and (B) of the COSMOD algorithm (mentioned in section 2). Performing these activities for newly discovered state S_{12} discovers three new states S_{16}, S_4 and S_8 and their transitions. Similarly, performing activities (A) and (B) for state S_{14} generates three more new states S_{18}, S_6 and S_{10} along with their transitions. We then perform activity (C) of COSMOD on the modified FSM that uncovers new cycles of the modified FSM. Executions of the *cycleMethodSequences* for these cycles do not generate any new state and transitions which indicates completion of the constructions of the FSM. The completed state model (FSM) has been shown in Figure 4.

Selection of parameters during repeated executions of a *cycleMethodSequence*

During repeated executions of the method borrowBook(*book*) of the *book* component of Figure 1, each invocation of this method requires different values of the parameter *book* because a library member is not supposed to borrow multiple copies of the same book. In case of repeated invocations of the method read&VerifyPIN(*PIN*) in state S_a of the *bankATM* component of Figure 2, the value of parameter *PIN* can be any arbitrary value, same or different, except the correct value of *PIN* in order to transit the component to state S_c. In case of PWC component of Figure 4, the method driveri(int *buttonPressed*) is the only method having a parameter. Every invocation of this method in various *cycleMethodSequences* needs to be made with the same parameter value during the repeated executions of the method-sequences, otherwise we will get the partial state model of Figure 3 rather than the complete state model of Figure 4. Therefore, during activity (C) for repeated executions of the method-sequences, COSMOD first uses the same set of values of the method-parameters for each invocation of a method. If, on doing so, a method invocation generates exception or COSMOD is unable to discover new "confirmed states", then it attempts to execute the methods in the method-sequences with different set of values of parameters for each invocation of a method.

Up to a certain number of repeated executions, state transitions follow the same sequence of states in the cycle. But during the process of repeated execution, some global variables may achieve values that cause a different execution path to be taken resulting in a transition to a state not in the cycle under consideration. If this state is not symmetric to any of the existing states, then we have discovered a new state that is a "confirmed state". On the other hand, if this state is symmetric to an existing state, then it could be a "potential state" which is likely to be merged to one of its symmetric states.

In order to curb state explosion, COSMOD discards a generated state T_j if T_j is symmetric to an existing state S_a and there is already a transition (an edge in the graphical FSM) from state T_i to the state S_a. This is because states T_j and S_a are very likely to be found equivalent and merged together at later stage. We would miss certain states in the state model in case the state T_j is not equivalent to any other state. But this check is not possible until we store T_j in *stateSet* and later perform activities (A) and (B) on it. Doing so will require us to handle several such states of which very small (or even zero) number of states may be found non-equivalent to any other state and finally become the part of the state model. Handling many such states would increase the computation time enormously. Therefore, COSMOD follows the policy of discarding such states.

Determining the value of the repetition factor r

The exact number of repeated invocations of *cycleMethodSequences* depends on the specifics of the underlying code. In the example of Figure 1, the borrowBook(*book*) method is to be executed n-1 times in state S_2 to transit the component to state S_3, where n is the number of books that a user is entitled to borrow, which is solely an application specific quantity and cannot be decided automatically. However, it is a very hard problem to determine the exact number of times a method sequence needs to be executed to discover all the states and transitions. For example, in a trial version shareware game allowing the user to play 1000 times without purchasing and registering the software, the value of r may need to be set to 1000 in order to transit the unregistered shareware component to a state when it asks the user to register before continuing. We have let the value of the repetition factor (r) to be determined by the user of COSMOD either arbitrarily or intuitively.

As already pointed out, the activities (A), (B) and (C) need to be carried out multiple times. However, for an FSM with large number (such as more than 20) of states, computation time of activity (C) may become quite large due to detection of all the cycles in the FSM followed by repeated execution of method-sequences along all the cycles. Hence, number of iterations of the activities (A), (B) and (C) may be kept small, such as 2 or 3 with a hope that it would uncover all (or most of) the states and transitions of the state model of the component under consideration. However, some states and transitions may still remain undetected, resulting in an incomplete state model.

Parameter Value Generation

We assume that parameters of methods specified in the IDL specification of a component are flat values and not objects. The domains of all parameters are assumed to be discrete and finite. Besides, component specifications are assumed to describe the constraints on parameter values. Constraints in the form of method preconditions simplify the parameter generation by restricting the domains of the parameter values, provided that the parameter values can be chosen independently. On the other hand, constraints in the form of predicates involving more than one parameter values may require each parameter combination to satisfy the predicate.

We proceed as follows to generate the parameters value combinations of a method $m(p_1, p_2, ..., p_n)$. Suppose the domain of the parameter p_i is d_i, $\forall i$. We assume that meaningful partitions of the domain d_i of each parameter p_i is specified in the IDL specifications of the component. Based on this, domain d_i is partitioned into q_i contiguous partitions denoted by $< d_{i1}, d_{i2}, d_{i3}, ..., d_{iqi} >$. Taking all possible combinations of partitions of every domain creates $\prod_1^n q_i$ combinations, such as $< d_{11}, d_{21}, ..., d_{n1} >, < d_{11}, d_{21}, ..., d_{n2} >, ..., < d_{1q1}, d_{2q2}, ..., d_{nqn} >$, etc. For each such combinations, we generate parameter value combinations by randomly selecting a value from each partition of the combination that generates a combination of n values one for each parameter p_i of the method m. For example, we can take the combination of domain-partitions $< d_{11}, d_{21}, ..., d_{n1} >$ and generate combinations of n values $< v_1, v_2, ..., v_n >$ where v_i is randomly chosen from the i_{th} domain-partition d_{i1} of the combination.

Identification of Guard Conditions of State Transitions

A guard condition is a predicate that needs to hold true for a transition to occur. A guard is usually specified as a boolean expression involving method parameters, local and global variables, constants, etc. However, in the absence of the source code, the only observable values are the parameter values in terms of which the guards would have to be expressed. Needless to say that guards may only be incompletely discovered. In COSMOD algorithm, the *transitionList*$(S,\ m)$ is used to record the method names along with their parameter values for every state transition. A simple (that is, incomplete) guard may be expressed as discrete values of the parameters of the method invocation that cause a state transition. However, different values of a parameter may cause the same state transition and therefore they are considered equivalent. The set of these parameter values form an equivalence class. A complete guard may be expressed in terms of equivalence classes of the involved parameters, local and global variables rather than in terms of their discrete values. Synthesis of a guard condition requires equivalence class partitioning, that is, partitioning the domains of the parameters into intervals (equivalence classes) such that all the values in an interval cause similar state transitions to occur. Finding such intervals for the parameters of a black-box component is a difficult task. Therefore, it is likely that COSMOD may not identify some of the guards in a state model.

Time Complexity Analysis of COSMOD

We have analyzed the worst case time complexity of COSMOD algorithm and computed as $T_{COSMOD[activities(A)and(B)]} = O(n{*}k^2{*}(n{*}T_{max}{+}T_p))$ and $T_{COSMOD[activity(C)]} = O((n + e) * c) + O(c * n * r * k * T_{max})$, where n is the total number of states generated including the potential states merged to earlier states, k is the number of methods, T_{max} denotes the maximum of the time complexities of all the k methods of the component, and T_p is the worst case time complexity of the algorithm generateParameterCombinations(m) used to generate the parameter values for a method invocation, e is the number of edges in the final FSM, c is the number of distinct cycles in the FSM, r is the repetition factor for activity (C) of COSMOD, and j is the number of times activities (A), (B) and (C) are iterated, i.e. the number of times the *repeat-until* loop of the COSMOD algorithm (see on page 86) is executed.

Therefore, overall time complexity of the COSMOD algorithm $T_{cosmod} = O(j* (n * k^2 * (n * T_{max} + T_p)) + ((n + e) * c) + (c * n * r * k * T_{max}))$. It can be observed that the worst case time complexity of COSMOD is a polynomial function of k, r, n, e, c, j T_{max}, and T_p. A detailed description of this analysis is available in [14].

3 CASE STUDIES

We have carried out several case studies to investigate the effectiveness of our COSMOD algorithm. We have chosen components from the following applications: (i) the *power window controller (PWC)* component of an Automobile System (AS) (ii) the process *scheduler* and (iii) the *thread* components of an Operating System (OS) (iv) the *vending machine (VM)* component of a Coffee Vending Machine (CVM) (v) the *book* and (vi) the *libMember* (denoting a library user) components of a Library Information System (LIS) (vii) the train ticket *reservation* component of a Train-ticket Booking System (TBS) (viii) the *order* (denotes a purchase order received by a vendor) component of a Trade House Automation System (THAS) (ix) the *session* (stands for a session of ATM transaction) component of a Bank ATM System (BAS) (x) the *elevator*

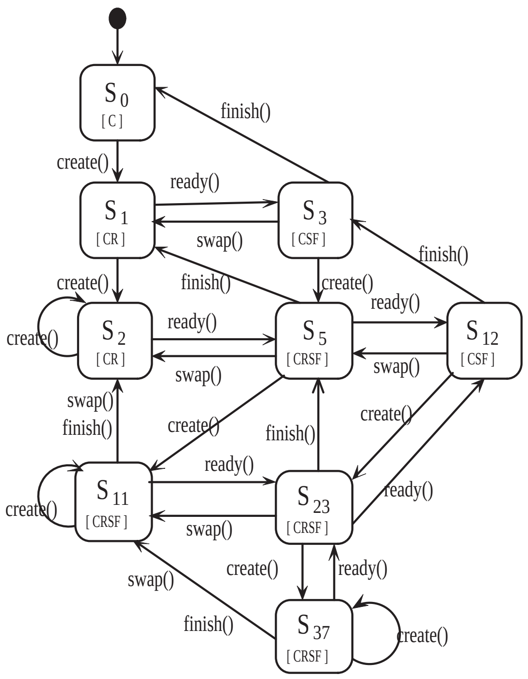

Figure 5: State model of a *process scheduler* component generated by COSMOD

component of an Elevator Control System
(ECS). The details of these applications, components and case studies have been reported in [14]. We briefly outline the two of these case studies.

State Model of a Process Scheduler Component

IDL specification of a *process scheduler* component has been shown below. Though

```
interface scheduler {
        int create(); // creates a new process and puts in waiting queue
        void swap();  // takes an active process and puts it in waiting queue,
                      // and takes one ready process and makes it active (running)
        void ready(); // takes a waiting process and puts it in ready queue, in case
                      // the processor is idle, makes this process active
        void finish();// removes the active process from memory, and takes one ready
                      // process and makes it active
}
```

the *process scheduler* component of a typical operating system contains many methods, we restrict our discussion to a *scheduler* component with only four methods, create(), ready(), swap() ,and finish() due to space constraints and to make the illustration easy to understand. Tasks performed by various methods of the scheduler component is described in the IDL specification itself with the help of embedded comments. The state model of the scheduler component obtained by applying the COSMOD algorithm has been shown in Figure 5.

The actual state model of this scheduler is not available to us. However, it is likely to contain very large number of states depending up on maximum number of processes allowed in the system at a time. Therefore, a complete state model of this is difficult to generate practically. COSMOD generates an abstracted state model on the basis of active methods. Due to unavailability of its actual state model for comparison, many cells in Table 1 (on page 97) are blank in the row corresponding to the *scheduler* component. Detailed explanation of applying the steps of all the activities of COSMOD algorithms, all the potential states generated, computation of SRMS, etc are available in [14].

State Model of a Power Window Controller (PWC) Component

The PWC component is based on the controller that is used in modern automobiles to control the movement of the window glasses. A single PWC component consists of three buttons goUp, goDown and neutral/stop to control the glass movements. At any point of time, the PWC remains in either of the three states dd (going down), uu (going up), or nn (neutral). When a button is pressed, the PWC may transit to another state. It then calculates the amount of movement, which is 1, -1 or 0, and moves the glass accordingly. It does not move the glass when the glass is touching the top (or bottom) position of the window and goUp (or goDown) or neutral button is pressed. After moving the glass, it checks whether the glass has touched the top or bottom of the window and remembers this position for computing the next movement.

Apart from initialization method, the PWC component supports 13 other methods which are (i) driveri(int *buttonPressed*)(ii) changeState0() (iii) changeState1() (iv) changeState2() (v) up11() (vi) upT (vii) down11() (viii) downT() (ix) neut() (x) update_pos() (xi) top_thres() (xii) bot_thres() (xiii) no_thres(). Of these methods, only the method driveri(int *buttonPressed*) takes a parameter that indicates the direction of the glass movement. All the methods are visible to an external entity as the PWC acts like a middle-ware. We applied the COSMOD algorithm to this component. The state model constructed by COSMOD for this component has been shown in Figure 4. A discussion regarding performing activity (C) on the partial state model of Figure 3 to obtain the state model of Figure 4 has already been presented in Section 2 under the subsection "The process of activity (C)" on pages 90 and 91. We observed that COSMOD could detect all the 24 states and 39 transitions of the PWC component that were present in a state model constructed using the knowledge of the source code. IDL specifications of the PWC component and detailed explanations of the steps in obtaining the state model of Figure 4 are available in [14].

It can be observed in Figure 4 that from state S_0, the component transits either to state S_1 or state S_2 upon invocation of the same method driveri(int). This function takes an integer value as input parameter. As per the IDL specifications of the PWC, input can be 0, 1 or 2 which denotes the power window button pressed (such as down, up, or stop) by the user which is detected by some sensor device. Transitions to states S_1 or S_2 depend on the input parameter of the method driveri(int *buttonPressed*). This input value is treated as the guard condition for transitions from state S_0 to states S_1 and S_2 as shown in Figure 4. The same guard condition is applied for transitions from states $S_{15}, S_{16}, S_{17}, S_{18}$, and S_{23} upon invocations of the function driveri(int *buttonPressed*). However, no guard condition could be detected for transitions to two different states S_{11} and S_{12} from state S_7 and to states S_{13} and S_{14} from state S_9 upon invocations of the method update_pos(). Transitions to different states from these states depend on changes in value of some internal local or global variables which are not visible to an external observer. However, changes in the values of internal variables change the active methods. COSMOD detects changes to the active methods by making repetitive calls to the same method in a state.

Values of Repetition Factor *r*

In case of PWC component, we tried with different values of $\boldsymbol{r}$ starting with a value 2. Setting $\boldsymbol{r}$ up to value 9 generates only the partial model of Figure 3. Applying the COSMOD algorithm to the PWC component with $\boldsymbol{r} = 10$ generates the state model of Figure 4. Higher values of $\boldsymbol{r}$, such as 11, 12, 15, 20, etc does not add any more state or transition to the FSM of Figure 4. In case of *process scheduler* component, setting $\boldsymbol{r}$ to 1, 2, or any higher value generates the same state model of Figure 5, that is, performing activity (C) does not generates any new state or transition in this case. Column 6 of Table 1 (on page 97) lists the values of $\boldsymbol{r}$ used in various case studies carried out by us.

Other Results

A summary of the results of the case studies carried out by us has been shown in Table 1 (on page 97). Columns 4 and 5 of this table list the number of states and transitions respectively of the considered components as computed manually on the basis of the design of the components. Column 6 lists the value of repetition factor given as input while applying the COSMOD algorithm. Columns 7 and 8 list the number of states and transitions discovered by applying the COSMOD algorithm. Columns 9 and 10 express the data presented in columns 7 and 8 as percentage of actual number of states and transitions (as listed in columns 4 and 5 respectively) successfully discovered. Columns 11 and 12 show the number of states and transitions discovered erroneously. Values in these columns show that the COSMOD algorithm has not detected any false states or transitions for any component. This substantiates our assumption that different sets of active methods characterize different states.

Table 1: Performance of COSMOD algorithm

1	2	3	4	5	6	7	8	9	10	11	12
		Computed by Component Designer			Giv-en by user	Computed using COSMOD Algorithm					
CS#	CN (AN)	NM	NS	NT	RF	NSD	NTD	%CSD	%CTD	FS	FT
1.	PWC (AS)	13	26	43	10	26	43	100%	100%	nil	nil
2.	scheduler (OS)	4			1	9	34			nil	nil
3.	thread (OS)	10	7	11	1	7	11	100%	100%	nil	nil
4.	VM (CVM)	3	3	4	1	3	4	100%	100%	nil	nil
5.	book (LIS)	8	8	26	1	8	26	100%	100%	nil	nil
6.	libMember (LIS)	4	5	9	10	5	9	100%	100%	nil	nil
7.	reservation (TBS)	10	8	35	10	8	26	100%	74.2%	nil	nil
8.	order(THAS)	6	8	11	1	8	11	100%	100%	nil	nil
9.	session (ATM)	13	14	25	3	14	25	100%	100%	nil	nil
10.	elevator (ECS)	7	10	72	1	3	12	30%	16.7%	nil	nil

Abbreviations used in the Table :
CS = Case Study, CN = Component Names, AN = Application Names, NM = Number of Methods, NS = Number of States, NT = Number of Transitions, RF = Repetition Factor used by COSMOD, NSD = Number of States Discovered, NTD = Number of Transitions Discovered, CSD = Correct States Discovered, CTD = Correct Transitions Discovered, FS = number of False States discovered, FT = number of False Transitions discovered.

It can be observed from columns 9 and 10 of Table 1 that the COSMOD algorithm successfully detects all (100%) the states and transitions for most of the components. In case of case study 2, we had a *process scheduler* component for which actual state model was not available and was difficult to generate manually. Hence many cells in Table 1 corresponding to this case study are blank. In case of study 7, *train ticket reservation* component denotes the berth reservation status of an individual passenger. On invocation of the active methods, its state depends on the ticket booking data stored in a centralized database that is not visible to COSMOD. However, the component accesses and possibly updates this database during the execution of its active methods. It is very difficult to simulate all possible states of this database by repeated executions of the cycle method sequences of the partial state model of this component. In such situations, it is not always possible for COSMOD to detect all the states and transitions. Therefore, it detected all the states, but could not detect many transitions of this component. In case study 10, the *elevator* component had several symmetric states which were different but could not be distinguished by COSMOD and merged. As a result, very few number of its states and transitions could be detected by COSMOD.

The component *libMember* (case study 5) requires repeated call to its methods in its various states. As discussed earlier (on page 92), it is very difficult to determine the repetition factor $\boldsymbol{r}$. However, while applying the COSMOD algorithm, we used the knowledge about the application that might use this component. In this case, COSMOD detected all its states and transitions by setting $\boldsymbol{r}$ = maximum number of books a library member is permitted to borrow at a time.

Effectiveness of the COSMOD algorithm in detecting the states and transitions in these cases depends on the following three factors (i) Effectiveness of the input data (parameter values) generation algorithm (ii) The exact number of times the *cycleMethod Sequences* are invoked (iii) The number of symmetric states that are not distinguishable from the other symmetric states because their target states match for each active method. Such states are considered as indistinguishable states by COSMOD.

A Critical Analysis of the Results

There can be three types of methods of a component (i) methods requiring no input parameter (ii) methods requiring input parameters whose values can be enumerated, such as boolean types (only possible values are *true* and *false*), fixed (possibly short) length strings and other enumerated types (iii) methods with input parameters (such as integer, float, arrays, structures, etc) for which enumeration of values is not possible or enumeration may result in too many values to be handled. COSMOD can effectively generate the state models for components containing methods of types (i) and (ii) only. This is reflected by accurate state and transition extraction by COSMOD in Table 1 (given on page 97). Thus, the COSMOD can satisfactorily be used for generating the state models of these type of components. However, if the state model of the component contains many symmetric states some of which cannot be proved different (non-equivalent) by observing the target states on invocations of their common active methods, then many different symmetric states may be regarded as equivalent states by COSMOD and merged (or left out) together resulting in an incomplete state model as happened in case of case study number 10 of Table 1.

For components containing methods of type (iii), some of the parameter value combinations may not be generated during repeated invocations of such a method. It may cause certain execution paths of the method code left uncovered even after repeated executions of the method with different parameter values. As a result, certain states of the component may not be reached and our COSMOD algorithm may generate only a partial state model of the component. Therefore, effectiveness of our approach for components containing methods of type (iii) is dependent on the effectiveness of the algorithm used for generation of data for input parameters of method executions.

4 COMPARISON WITH RELATED WORK

Research work addressing state model extraction of components have scarcely been reported in the literature. Due to unavailability of any directly comparable work, we compare our work with the reported work on state model extraction using code and other design artifacts.

Whaley et al [17] proposed multiple submodels of a component interface. Each submodel was a sliced FSM that represented valid sequences of the methods which access the same member field. A state in a submodel was a method that could produce a side-effect (that is, modify some attribute values). The model also shows which side-effect free methods were callable by which side-effect producing method. An instance of the component was considered to be in a state represented by a method that was called last at that point of time. They detected illegal method call sequences by static analysis of the source code of the component and extracted the submodels by executing the instrumented byte code. These submodels, however, may not represent the behavior of the component in its different states where a state has a more generic semantics rather than being represented by a method. Further, as this approach is based on analysis of the source code, its usability is restricted to the components whose source code is available.

In [15], Xie et al extracted sliced object state machines (OSMs) for components by invocation of methods with generated parameter values. States of the ISM were considered to be comprised of the values of the member fields of the component. The huge number of states resulting due this were sliced (abstracted) by considering only one member field at a time that produced a sliced ISM. Further, they proposed two more ways of abstraction of object states in [16]. One was observer abstraction approach that represented a state of an object by the return values of observer (non void) methods. In another abstraction, they defined a state in terms of the combinations of various branch coverages achieved by different methods calls. Each of the three abstractions produces different slicing of ISM and represents the object behavior from a specific perspective. In contrast to the sliced OSMs of Xie and sliced FSMs of Whaley, our generated state model shows the aggregate behavior of a component as is observable externally.

Lorenzoli et al [21] proposed a technique to generate behavioral models of software systems in the form of extended FSMs (FSMs) by monitoring software executions. Their definition of EFSM is similar to the FSM used in COSMOD in which transitions are annotated by method names, input parameters and a predicate (we called it *guard conditions*) in terms of parameter values and program variables with difference that COSMOD assumes that local and global variables of a software component are not visible, and therefore, not the part of guard conditions of the FSM generated by COSMOD.

Work reported in [1, 3, 5, 6, 7] propose techniques for automatic generation of state models of object-oriented programs through static and/or dynamic analysis of the source code. Yu *et al* [5] refactored the source code based on the programmer's comments and constructed the state model based on a static analysis of the refactored code. Systa *et al* [3] instrumented the source code of an object-oriented program, executed the program and, based on the analysis of a log file, generated typical use case scenario diagrams. They construct the state diagram using the scenario diagrams in which the object participates. Kung *et al* [7] generated the state diagram through symbolic executions[8] of the class methods and observing the changes to the values of the member variables. The approaches of Gupta [6] and Tonella *et al* [1] to generate statecharts of objects are similar to Kung's approach. Gupta [6] used *class method contracts* (*CMCs*) which are pre-conditions and post-conditions of method invocations together with other constraints (called invariants) that are to be satisfied at all times by all instances of the class. This method works satisfactorily for object-oriented programs consisting of many different data types, such as strings, object references, etc. However, availability of *CMCs* is a stringent requirement, severely restricting the applicability of Gupta's method. Tonella's [1] and Kung's [7] symbolic execution approaches work well for simple programs.

Most of the above methods are based on analysis of source code. As a result, those approaches cannot be used in an obvious way to extract the state model of black-box components. The methods used in [15, 16] do not use source code. However, each of their sliced OSMs generated for a component present only a specific aspect of the state behavior and differ from the generic behavioral view of components. Our approach analyzes the behavior of a component as can be observed externally, and synthesizes the state model without any reference to its source code.

5 CONCLUSION

The source code for off-the-shelf components are usually not available. Therefore, most of the related work cannot directly be used for state model discovery of black-box components. We perform repeated service invocations and analyze the component behavior to synthesize the state model. Our experimentation shows that our approach is able to generate only partial models for components having methods taking parameter values from large domains. However, it is more effective in generating the state models of components containing methods that either take no parameter or their parameters assume a small set of discrete values.

Our generated model is essentially an FSM which may need to be converted to a statechart representation if required. The identified states are, at present, being manually assigned meaningful names. We are now investigating assignment of meaningful names to states through intelligent inferencing based on the names of the triggering methods.

References

[1] Tonella, P., and Potrich, A.: “Reverse Engineering of Object-Oriented Code”, Springer-Verlag, Berlin, Germany, 2005.

[2] Binder, R.V.: “Testing Object-oriented Systems: Models,Patterns, and Tools”, Addison-Wesley Longman Publishing Co, Inc, Boston, MA, 1999.

[3] Systa, T., and Koskimies, K.: “Extracting state diagrams from legacy systems”, In Proc. of Workshop on Object-Oriented Technology, Lecture Notes in Computer Sc, Springer-Verlag, London, 1997, Vol. 1357, pp. 272-273.

[4] Harel, D.: “Statecharts: A visual formalism for complex systems”, Science of Computer Programming, 1987, Vol. 8(3), pp. 231-274.

[5] Yu, Y., Wang, Y., Mylopoulos, J., Liaskos, S., Lapouchnian, A., and Leite, J. C. S. P.: “Reverse Engineering Goal Models from Legacy Code”, In Proc. of the 13th IEEE International Conference on Requirements Engineering (RE’05), IEEE Computer Society, 2005, pp. 363-372.

[6] Gupta, A.: “Unit Testing of Object Oriented Programs”, PhD. Thesis, CSE Dept, IIT Kanpur, India, 2007, pp. 127-149.

[7] Kung, D., Suchak,N., Hsia, P., Toyoshima, Y., and Chen, C.: “On Object State Testing”, In Proc. of IEEE COMPSAC’94, IEEE Computer Society Press, 1994, pp. 222-227.

[8] Clarke, L.: “A system to generate test data and symbolically execute programs”, IEEE Transaction on Software Engineering, 1976, Vol. 2(3), pp. 215-222.

[9] Wagner, F., Schmuki, R., Wagner, T., and Wolstenholme, P.: “Modeling Software with Finite State Machines: A Practical Approach”, CRC Press, 2006.

[10] Holcombe, M., and Ipate, F.: “Correct Systems - Building Business Process Solutions”, A volume in the Applied Computing Series, Springer-Verlag. 1998.

[11] Gao, J. Z., Tsao, H. S. J., and Wu, Y.: “Testing and Quality Assurance for Component-based Software”, Artech House publication, Norwood, USA, 2003.

[12] Ramachandran, M.: “Testing software components using boundary value analysis”, In Proc. of the 29th Conference on EUROMICRO, 2003, pp. 94-98.

[13] Cai, K., Chen, T. Y., Li, Y., Ning W., and Yu, T. Y.: “Adaptive testing of software components”, In Proc. of the 2005 ACM Symposium on Applied Computing, Santa Fe, New Mexico, 2005, pp. 1463-1469.

[14] Suman, R. R.: "State Model Extraction of Software Components by Analyzing Their External Behavior", Technical Report No 01/2009, CSE dept, IIT Kharagpur, India.

[15] Xie, T., and Notkin, D.: "Automatic extraction of sliced object state machines for component interfaces", In Proc. of the 3rd Workshop on Specification and Verification of Component-Based Systems at ACM SIGSOFT 2004/FSE-12 (SAVCBS 2004, Oct. 2004), pp. 39-46.

[16] Xie, T., Martin, E., and Yuan, H.: "Automatic extraction of abstract-object-state machines from unit-test executions", In Proc. of the 28th International Conference on Software Engineering, ICSE '06, Shanghai, China, May 2006, pp. 835 - 838.

[17] Whaley, J., Martin, M. C., and Lam, M. S.: "Automatic extraction of object-oriented component interfaces", In Proc. of the International Symposium on Software Testing and Analysis, 2002, pp. 218-228.

[18] Johnson, D. B.: "Find all the elementary circuits of a directed graph", SIAM Journal on Computing, Vol. 4(1), 1975, pp. 77-84.

[19] Liu, H., and Wang, J.: "A new way to enumerate cycles in graph", In Proc. of the Advanced international conference on telecommunications and international conference on Internet and web applications and services, 2006, pp. 57-59.

[20] Gallagher, L., Offutt, J., and Cincotta, A.: "Integration testing of object-oriented components using finite state machines", Software Testing, Verification and Reliability, Vol. 16(4), Jan 2006, pp. 215 - 266.

[21] Lorenzoli, D., Mariani, L., and Pezze, M.: "Automatic Generation of Software Behavioral Models", ICSE '08. ACM/IEEE 30th International Conference, May 2008, pp. 501-510.

[22] Suman, R. R. and Mall, R.: "State model extraction of a software component by observing its behavior", ACM SIGSOFT Software Engineering Notes Vol. 34(1), Jan. 2009, pp. 1-7.

ABOUT THE AUTHORS

Rajiv R. Suman is a PhD student in the Department of Computer Science and Engineering at Indian Institute of Technology (IIT), Kharagpur. He can be reached at rrsuman@cse.iitkgp.ernet.in and rrsuman2001@yahoo.com.

R. Mall is a Professor in the Department of Computer Science and Engineering, Indian Institute of Technology (IIT), Kharagpur. He has published over 100 refereed research papers and has authored two books. He is a member of the domain experts board of the International Journal of Patterns (IJOP). He was the general chair of IEEE Indicon 2004 and program chair for CIT 2005. He was also a program committee member for a large number of international conferences. His current research interests include analysis and testing of object-oriented pro- grams. He can be reached at rajib@cse.iitkgp.ernet.in.

Srihari Sukumaran is a Senior Researcher at GM India Science Lab, Bangalore. He can be reached at srihari.sukumaran@gm.com.

Manoranjan Satpathy is a Staff Researcher at GM India Science Lab, Bangalore. He can be reached at manoranjan.satpathy@gm.com.