Softw Syst Model (2016) 15:497–511
DOI 10.1007/s10270-014-0415-3

REGULAR PAPER

Extracting finite state representation of Java programs

Tamal Sen · Rajib Mall

Received: 24 May 2013 / Revised: 25 March 2014 / Accepted: 5 April 2014 / Published online: 24 June 2014

Abstract We present a static analysis-based technique for reverse engineering finite state machine models from a large subset of sequential Java programs. Our approach enumerates all feasible program paths in a class using symbolic execution and records execution summary for each path. Subsequently, it creates states and transitions by analyzing symbolic execution summaries. Our approach also detects any unhandled exceptions.

Keywords Software reverse engineering · FSM · System modeling

1 Introduction

Many program components, especially those used as interactive controllers, are state based. A state-based program may behave differently in different states on an input. A state model of a program is a behavioral model that depicts the different states of the program and the transitions among states that are triggered by system functions invocations. Many state model formalisms are available including FSM, Harel's [6] state chart and Petrinets [16]. State models are widely being used to define and validate the behavior of a variety of software systems including interactive systems, embedded controllers, compilers, operating systems, and telecommunication systems. A state model can be used in test case generation, test case selection, software metric computation and can also be helpful in understanding the dynamic behavior of a program. Despite their apparent benefits, state machines are rarely maintained over time. Dynamic nature of software evolutions makes it difficult to keep design artifacts up to date [24]. The need of reverse engineering state models from implementations is thus multifold.

1. Software maintenance activities account for over 50 % of the total software development costs, as the maintainers need to spend significant amounts of time to understand the code before carrying out changes. Object-oriented program features such as inheritance, polymorphism, and dynamic binding make it difficult to statically determine which methods would be bound at runtime. Additional features of these programs such as multi-threading and distributed execution further complicate the process of code understanding. To help code understanding, maintainers often reverse engineer the design models when these are either not available or are obsolete [4].
2. When using component-based development, the state models can be used for effective testing of an application. In [17], we had proposed a technique for regression test reduction based on analysis of component state model. Beydeda et al. [1] and Gallagher et al. [5] have also proposed techniques for efficient integration testing in a component environment based on the implicit assumption that the state transition specifications for individual components are available a priori. When the state transition specifications are unavailable, it becomes necessary to reverse engineer these from the code, before many of the available test techniques can be applied.
3. In the model-driven development (MDD) paradigm, software systems are typically developed by first building their state models, and then implementing these models.

Communicated by Prof. Gary Leavens.

T. Sen (✉) · R. Mall
Department of CSE, Indian Institute of Technology, Kharagpur, Kharagpur 721302, India
e-mail: tamal.sen@cse.iitkgp.ernet.in

R. Mall
e-mail: rajib@cse.iitkgp.ernet.in

The source code of such systems is generated essentially by mapping states and transitions into concrete programming constructs [18]. However, modifications in the generated code might affect states and transitions, and therefore, it might be necessary to reverse engineer the state model from the modified code to identify changes in the state transition behavior caused by the modification.

In this work, we propose a technique for extracting a *finite state model* from Java programs based on a static analysis-based technique by Kung et al. [12]. Our main objective of this work is to extract a reasonably useful model by overcoming some of the shortcomings of Kung et al.'s [12] technique. For a class *c* in a program, we construct an object state model (OSM), and we reuse it in the construction of OSM of any class that is dependent on *c*. We execute each method of a class symbolically to compute an execution summary for each execution path. By analyzing execution summaries, we define states as a partition over the field values, and transitions as the change in field values due to a method invocation. Transitions may be guarded by a condition which is essentially a boolean constraint over the parameter variables of the method that causes it. The extracted state model also includes exceptional states and exceptional transitions to represent occurrence of runtime exceptions that are not handled in a path and might cause the program to fail.

For model extraction, we analyze the Java bytecode instead of source code, since bytecode analysis has a number of advantages. A few of these advantages are as follows:

– Often, source code is not available. This is especially true in a component-based development environment.
– A bytecode analyzer need not carry out tasks such as name resolution and type checking as these are ensured by the Java compiler.
– A bytecode analyzer needs to deal with much fewer constructs than a source analyzer and thereby is less complex [14].

We have named the technique **State** model extractor for **J**ava programs (StateJ).

This paper is organized as follows: in Sect. 2, we discuss several background concepts; in Sect. 3, we formalize various aspects of an *object state model*; in Sect. 4, we describe our approach for constructing an *object state model*. In Sect. 5, we provide the details of experiment setup and results of the experiments. In Sect. 6, StateJ is compared with other related approaches and finally Sect. 7 concludes the paper.

2 Background concepts

In this section, we discuss the background concepts based on which we have developed our approach.

2.1 Symbolic execution of Java programs

Symbolic execution [9] is a static program analysis technique that considers symbolic values as the inputs to the program instead of actual ones (concrete values). The outputs are expressed as a function of the symbolic inputs [15]. We implement our symbolic execution engine by extending symbolic execution extension (jpf-symbc) of a well-known Java model checker called Java PathFinder (JPF) [8]. JPF is developed in NASA, and it has been widely used to find faults in complex Java components. JPF is available as a highly extensible open-source tool. JPF systematically explores possible execution states and automatically detects anomalies such as deadlock, race conditions, unhandled exceptions. It is also capable of verifying user-defined properties specified as assertions. Following are the important features supported by JPF.

– *Symbolic/concrete method execution* JPF can be configured to execute a method either symbolically or concretely. A concrete method is invoked with the arguments that are actually passed to it from the program. On the other hand, if a method is marked as symbolic, its parameters are considered to be unrestricted in the sense that these can hold any value of its domain at runtime. The actual arguments passed to a symbolic method are ignored, and these are initialized with a symbolic value of proper type before the symbolic execution of the method begins. A method parameter of non-primitive type (object of another class or an array) is not initialized immediately before starting symbolic execution, but initialized lazily when it is first accessed during the symbolic execution. The fields of an object can also be configured to be concrete or symbolic. A symbolic field is initialized in the same way that a parameter of a symbolic method.
– *Stack* JPF simulates the stack operations during symbolic execution. Each thread has a stack which stores the local variables including method parameters for each invoked method.
– *Heap* The symbolic execution engine also maintains a heap, referred to as symbolic heap, in order to store objects. The heap is represented as a graph in which nodes represent objects, primitive values and symbolic expressions. An edge in the heap connects a node representing an object and a node representing a field of the object.
– *Execution state* In JPF, a symbolic execution state (should not be confused with object state) consists of a global heap and a stack (in general, one stack for each thread, but we consider here only single-threaded programs), as well as program counter (address of the instruction currently being executed). Additionally, each symbolic state also stores the path condition.

– *Choice handling* JPF explores all feasible paths of a program recursively using a depth first search approach. During symbolic execution, while evaluating a branch condition, sometimes we do not have enough information to decide which branch to follow. In this case, it is necessary to *non-deterministically* follow both branches to safely simulate all possible real executions. At a branching point (such as, a conditional instruction), JPF registers a *choice generator* to explore all available branches. The *choice generator* is initialized with the set of non-deterministic choices available at that point. Subsequently, the current symbolic execution state is saved, path condition is updated with the first choice, and the execution proceeds. In this way, when execution reaches the path end, it backtracks to a previously saved state and the unprocessed choices in that state are explored; it backtracks again if no choice is left for processing. The execution terminates if no unprocessed choice is left. JPF also supports registering choice generators with user-defined choices at any point of a program. JPF executes the rest of the path for each user-defined choice in the same way as it handles branch conditions.

2.2 State model extraction using static analysis

Depending on the values of member variables of an object, a method may produce different output even when invoked with the same set of arguments. This feature can be exploited to model an object as a *state transition system* in which the states of the object are defined by the values assumed by a subset of member variables. However, it is not possible to describe each possible set of values of member as a distinct state because of the inherent combinatorial explosion of the number of states. Therefore, equivalence classes on the values of member variables need to be introduced [21].

Several researchers including Tonnela et al. [21] and Walkinshaw et al. [23] had pointed out the difficulty in automatically extracting the states from an arbitrary program. In their approaches, they assumed that abstract states (and sometimes abstract interpretation of the methods also) are identified manually while the transitions among states are identified automatically by static analysis. However, Kung et al. [12] proposed an approach which identifies states of a program automatically by analyzing conditions on member variables that appear in conditional statements and control the execution path. They computed intervals on the values of each member variable of the class. Using these intervals, they computed a partition over the values that member variables may assume. Subsequently, by examining symbolic execution summaries, they identified the state changes (transition) based on the modifications to the member variables in the path.

```
class AddSub{
 int one,two;
 int meth(int arg1, int arg2){
    if(two>2 && one <1
     && arg2>arg1
      && arg2 + two == 4){
      two=2;
      return arg1+arg2;
    }
    else
     return arg1-arg2;
} }
```

Fig. 1 An Example class to demonstrate the different types of atomic conditions

We provide an overview of working of Kung et al.'s technique by applying it to the Java program shown in Fig. 1. Symbolic execution of the method *meth* explores a set of paths. Path conditions of these paths contain the following conditions on the state variables.

– $\text{two} > 2 \wedge \text{one} < 1$
– $\text{two} > 2 \wedge \text{one} \geq 1$
– $\text{two} \leq 2 \wedge \text{one} < 1$
– $\text{two} \leq 2 \wedge \text{one} \geq 1$

By examining these conditions, the following intervals on the state variables are computed: ($\text{two} \leq 2$), ($3 \leq \text{two}$), ($\text{one} \leq 0$), ($1 \leq \text{one}$). Kung et al'.s technique identifies states (S0, S1, S2, S3 and S4) by computing all possible combinations on the intervals of *state variables* which are shown below.

– The object is uninitialized :S0
– $(\text{two} \leq 2) \wedge (\text{one} \leq 0)$:S1
– $(3 \leq \text{two}) \wedge (\text{one} \leq 0)$:S2
– $(\text{two} \leq 2) \wedge (1 \leq \text{one})$:S3
– $(3 \leq \text{two}) \wedge (1 \leq \text{one})$:S4

After creating states, the proposed technique creates the transitions based on how state variables are updated in a path. For example, consider the path having path condition $(\text{two} > 2) \wedge (\text{one} < 1) \wedge (\text{arg2} > \text{arg1}) \wedge (\text{arg2} + \text{two} == 4)$. This path can only be executed when an object of the class is in S2. For this reason, Kung et al.'s technique determines S2 as a **pre-state** for the transitions that are in the path. The path updates the value of variable *two* with 2 but variable *one* remains unchanged; therefore, the state after execution of the path is S1, which is referred to as **post-state**. Kung's technique creates a transition between the **pre-state** (S2) and the **post-state** (S1) pairs. The transitions in the complete state model of `AddSub` have been shown in a tabular form in Table 1.

3 Framework

In this section, we first present algorithms for extracting states and transitions from symbolic execution summaries. Subsequently, we define product of two object state models and prove that the complete state model of a derived class is the

Table 1 Transitions of the object state model of class `AddSub`

Id	Method	Source state	Target state	Guard	Returned expression
t1	meth()	S3	S1	arg2 > arg1	arg1 + arg2
t2	meth()	S3	S3	arg2 ≤ arg1	arg1 – arg2
t3	meth()	S2	S2	–	arg1 – arg2
t4	meth()	S1	S1	–	arg1 – arg2
t5	meth()	S4	S4	–	arg1 – arg2
t6	Example()	S0	S1	–	Void

product of its partial state model and the state model of its parent class.

StateJ uses the following formalisms concerning symbolic variables and symbolic expressions which have been adopted from Xie et al.'s work [25].

– Each symbolic variable has a type, which is considered to be one of the Java types.
– A symbolic expression is either a symbolic variable, a Java constant or an expression in which a set of symbolic expressions of the appropriate operand types are connected with an operator. For example, x1 and x2 may be each a symbolic variable (and thus also a symbolic expression) of type int and x1 + x2 and x1 > x2 are symbolic expressions of type int and boolean.
– A symbolic expression of type boolean is also referred to as a condition.

An *atomic condition* is either a symbolic variable of type boolean or an expression involving two symbolic expressions connected by a relational operator. A *compound condition* is a set of *atomic conditions* connected by conjunction ($\wedge$), disjunction ($\vee$) or negation ($\neg$) operator. A *conditional literal* in either an *atomic condition* or the negation of an *atomic condition* [12].

Example 1 If b_1 is a symbolic variable of type boolean, then, b_1 itself is an *atomic condition*. Similarly, $x_1 + x_2 > 2$ is an *atomic condition* where x_1 and x_2 are symbolic variables of primitive types (int, float, long, double, etc.). $b_1 \wedge x_1 + x_2 > 2$, $\neg(x_1 + x_2 > 2)$ are examples of *compound conditions* since these are formed by applying conjunction and negation operators on a set of *atomic conditions*. $b_1, x_1 + x_2 > 2$ and $\neg(x_1 + x_2 > 2)$—these three conditions can also be referred to as *conditional literals* since each of these is either an atomic condition or negation of an atomic conditions.

A *conditional literal* can be categorized into one of the following three subtypes.

Definition 1 (*Member dependent literal (MDL)*) A conditional literal is a Member Dependent Literal if the variables used in it are member variables of the enclosing class.

Definition 2 (*Parameter dependent literal (PDL)*) A conditional literal is a parameter dependent literal if the variables used in it are the parameters of a method of the enclosing class.

Definition 3 (*Mixed literal (MXL)*) A conditional literal is a mixed literal if it is neither an MDL nor a PDL.

Example 2 Consider the class `AddSub` shown in Fig. 1. The condition in line 4 consists of three conditional literals: two > *one*, arg2 > arg1 and arg2 == two. Among them, two > 2 and one < 1 are MDL since both of them consists of only state variables, arg2 > arg1 is PDL since arg2 and arg1 both are parameters of method `meth(int,int)` and arg2 + two == 4 is an example of MXL since arg2 is a parameter of `meth(int,int)` whereas *two* is a state variable.

For each type of conditional literals LT, we define a decision function called LT(l), where l is an arbitrary conditional literal. LT(l) returns *true* only if l is of type LT. For example, we define a function called MDL(l), MDL(l) to be *true* only if l is an MDL.

Example 3 For the conditional literal two > one, MDL(two > one) returns $true$. Similarly, MDL(arg2 > arg1) returns $false$.

3.1 Path summary

Symbolic execution explores every path of a method under consideration.[1] The parameter variables of the method as well as member variables of the class (which encloses the method) are treated as inputs to the method and these are considered as symbolic. During symbolic execution of a path, a set of information is recorded which is processed later in order to extract the states and transitions. Collectively we refer to these recorded information as *path summary*. A *path summary*, obtained by symbolic execution of a path, has the following attributes:

[1] Symbolic execution has inherent limitations, which makes it infeasible to explore all paths of an arbitrary method in the presence of loops and recursions. We discuss implementation details of our symbolic execution engine in Sect. 4.

- *pc*—The path condition of the path. It is represented as conjunction of a set of *conditional literals*. According to the definition of path condition [9], *pc* must be satisfied by the inputs, for the path to be executed.
- *fn*—The method to which the path belongs.
- *Vmap*—A set of name-value pairs of updated member variable identifiers and their symbolic values.
- *exception*—This attribute represents the type of an exception raised during execution of the path but that is not handled. This attribute is empty if there is no unhandled exception in the path.
- *r*—A symbolic expression returned by *fn* at the end of execution of the path.

Given a *path summary* ps, an attribute *a* of ps is referred to as $ps.a$. For example, *Vmap* attribute of ps is referred to as $ps.Vmap$.

3.2 State

Let us assume that, for a certain class, n paths have been explored by the symbolic execution of all methods of a class and the path summaries of these paths are denoted by $ps_0 \ldots ps_{n-1}$. Now let M be the set of all MDLs from all path conditions, i.e.,

$$M = \bigcup_{i=0}^{n-1} \{m | m \in cset(ps_i.pc) \wedge \text{MDL}(m)\}$$

where $cset(ps_i.pc)$ represents a set of conditional literals that have been used in $ps_i.pc$. The states of an object are defined as a partition over the set of values of member variables. In StateJ, a state s has two attributes: a unique state label called $s.label$ and a condition called $s.cond$. An object said to be in a state s if the values of its member variables satisfy $s.cond$. StateJ identifies the states using the steps presented in Algorithm 1.

Example 4 Let us assume that $\{m1, m2, m3\}$ represents the set of all MDLs. A $(true, true, true)$ valuation to these MDLs describes a state with state condition $m1 \wedge m2 \wedge m3$. Similarly, A $(false, true, false)$ valuation describes a state of state condition $\neg m1 \wedge m2 \wedge \neg m3$.

Theorem 1 *The set of all state conditions of an object defines a partition over the values of its member variables.*

Proof Let $M = \{m_1, m_2, m_3 \ldots m_k\}$ be the set of k *MDLs* over n member variables of a class and SC be the set of all state conditions identified using Algorithm 1. Let us assume $V = \langle v_1, v_2, v_3, \ldots, v_n\rangle$ is a vector that contains a set of values assumed by n members in order. Now, in order to prove that the set of conditions in SC defines a partition over the values of n members of the class, we need to show: (1) V satisfies at least one condition in SC, and (2) V satisfies only one condition in SC.

Algorithm 1 Algorithm for Creating States

```
Require: $M$: set of all *MDLs*, *PS*: set of all path summaries
Ensure: $S$: set of states.
 1: initialize $S$
 2: for all possible *true/false* valuation $v$ to $M$ literals do
 3:    Create a *compound condition* $cv$ such that,
       $cv = \bigwedge_{c_i \in M} \begin{cases} c_i & \text{if } c_i \text{ is true in } v \\ \neg c_i & \text{if } c_i \text{ is false in } v \end{cases}$
 4:    Create an empty state $s$
 5:    Assign an unique label to $s.label$
 6:    $s.cond = cv$
 7:    $S = S \cup s$
 8: end for
 9: Create an empty state $start$
10: $start.cond = false$
11: $start.label = 0$
12: $S = S \cup start$
13: for all $ps \in PS$ do
14:    if $ps.exception$ is set then
15:       Create an empty state $ex$
16:       Assign an unique label to $ex.label$
17:       $ex.cond = false$
18:       $S = S \cup ex$
19:    end if
20: end for
```

1. Let us choose an arbitrary condition sc from SC. As per our definition, s is a conjunction of $kMDLs$. Let us assume V satisfies k_1 literals in sc but contradicts remaining $(k - k_1)$ literals. Now there must be a literal sc' in SC in which those k_1 literals appear as it is, but the other $(k - k_1)$ literals appear as negated. Hence, V must satisfy sc', and therefore, it can be concluded that an arbitrary valuation V to n member variables will satisfy at least one condition in SC.
2. Let us assume that, for the sake of contradiction, V satisfies more than one state condition, that is sc and sc' $(sc, sc' \in SC)$ both. But as we know, there must be at least one $m_i \in M$ in sc which appears as negated in sc'. Therefore, if V satisfies both sc and sc', it has to satisfy both m_i and $\neg m_i$, which is not possible. Therefore, our assumption must be wrong that V satisfies more than one state condition. Thus, it can be concluded that V can satisfy only one condition in SC. □

3.3 Transition

The notion of transition in our approach is very similar to that of Kung et. al's approach. Execution of each path leads to one or more state transitions depending on the exact way in which the member variables of the object are modified in the path. In StateJ, a transition t has the following attributes: *pre* and *post* states, the method *fn* that triggers t, a guard condition g on the parameters of *fn* which must be satisfied for t to occur, and a return expression r which is a symbolic expression returned by *fn* when the transition *t* occurs. The steps presented in Algorithm 2 identifies the set of transitions.

Algorithm 2 Algorithm for Creating Transitions

```
Require: $S$: Set of all states, $PS$: Set of all path summaries
Ensure: $T$: Set of transitions
1: Initialize $T$ as empty set of transitions
2: for all $ps \in PS$ do
3:   for all $s_i \in S$ do
4:     if $satisfy(s_i.cond \wedge ps.pc)$ OR $isConstructor(fn)$ then
5:       if $isConstructor(fn)$ then
6:         $pre = start$
7:       else
8:         $pre = s_i$
9:       end if
10:      for all $s_j \in S$ do
11:        if $satisfy(weak\_pre(s_j.cond, p.Vmap) \wedge ps.pc)$ OR ($ps.exception$ is set) then
12:          if $ps.exception$ is set then
13:            $post = find\_state(ps.exception)$
14:          else
15:            $post = s_j$
16:          end if
17:          $fn = ps.fn$
18:          $g = conjunct(\{l | l \in cset(weak\_pre(s_j.cond, p.Vmap) \wedge ps.pc) \wedge PDL(l)\})$
19:          $r = ps.r$
20:          Create transition $t = (pre, post, fn, g, r)$
21:          $T = T \cup t$
22:        end if
23:      end for
24:    end if
25:  end for
26: end for
```

The following are the subroutines that have been used in Algorithm 2.

- *satisfy*$(cond)$: returns true if $cond$ is satisfiable, false otherwise.
- *find_state*$(label)$: returns a state s for which $s.label = label$.
- $weak\_pre(cond, map)$: returns a condition after replacing a set of variables in $cond$ with their values as specified in map.
- $conjunct(cond\_set)$: returns a compound condition which is a logical conjunction of the literals in $cond\_set$.
- $cset(cond)$: returns the set of conditional literals used in a condition $cond$.
- *isConstructor*(fn): returns *true* if *fn* is a constructor method of the class under consideration.

Algorithm 2 analyzes *path summary ps* of each path to identify the transitions that can occur during execution of the path. In line 4, the algorithm checks whether the state condition of a state s_i is satisfiable in conjunction with the path condition $ps.pc$. If so, the path can cause a transition from the state *pre*. In line 5, the algorithm checks whether the method $ps.fn$ is a constructor method. For a path belongs to a constructor, the *pre* state of possible transitions is always the *start* state.

After that, in the lines 10–23 the algorithm iterates through set of all states to identify possible *post* states. Modification of member variables in a path leads to a state change in the object. A state s_j can be a *post* state for a transition occurs during execution of p if the weakest precondition is *true* in conjunction with the path condition of the path, i.e., $(weak\_pre(s_j.cond, p.Vmap) \wedge ps.pc)$ is *true*.

The block of lines 12–16 checks whether the *exception* attribute of the *path summary* $ps.exception$ holds a non-empty value, which indicates that the execution of the path causes an unhandled exception. In that case, the *post* state is nothing but an exception state which corresponds to the exception type stored in $ps.exception$.

In line 18, the algorithm identifies the set of all *PDLs* that must be satisfied in order for the transition to occur. The conjunction of those *PDLs* become the *guard* of the transition.

Note that, the condition in line 4 may be satisfied by more than one state condition. Similarly, multiple post-state can be matched satisfying the condition in line 11. Thus, a single path creates multiple transitions each of which is possibly feasible in the runtime.

We do not include *MXLs* in transition guard since *MXLs* may contain private member variables. Ignoring an atomic condition from the guard of a transition makes the guard weaker, and thus, it would not lead to miss a transition. However, there is a possibility that a transition exists in the state model but may never occur during real execution.

StateJ invokes SAT solvers to identify pre- and post-states during creating transitions. When a SAT call fails to return within a specified time bound, the return value is assumed to be true. This assumption may result in creating a transition which would not get created if the boolean formula is actually false, but that would not lead to miss transitions.

3.4 Object state model

Based on the states and transitions identified for an object, we define the object state model of a class as the following.

Definition 4 (*Object state model (OSM)*) An object state model of a class is a quadruple: $\langle S, start, Exception, T\rangle$ where

- S—finite set of states.
- $start \in S$—represents a state with uninitialized state variables. The state condition of start, $start.cond$, evaluates to $false$.

- $Exception \subseteq S$—The object transits to one of these states when an unhandled exceptions is raised.
- T—finite set of transitions.

Definition 5 (*Default OSM*) Default OSM of a class is $(S, start, Exception, T)$ where,

- $S = \{\{start\} \cup \{exception\} \cup \{s_0\}\}$.
  $s_0.cond = true$
  $exception.cond = false$
- $start = start\ state$
- $Exception = \{exception\}$
- $T = \{(s_0, s_0, fn, true, unknown) | fn\ is\ a\ method\ of\}$ $\{the\ class\} \cup$ $\{(s_0, exception, fn, true, unknown) | fn\ is\ a\ method\ of\}$ $\{the\ class\}$.

Definition 6 (*Product of two OSM*) Product of two given OSMs $osm_1 : (S_1, start_1, Exception_1, T_1)$ and $osm_2 : (S_2, start_2, Exception_2, T_2)$ yields another OSM $(S, start, Exception, T)$, for which:

- $S = \{\{S_1 \backslash start_1\} \times \{S_2 \backslash start_2\}\} \cup (start_1, start_2)$.
  The state condition of a state $(s_i, s_j) \in S$ holds the condition: $(s_i, s_j).cond \Leftrightarrow s_i.cond \wedge s_j.cond$.
- $start = (start_1, start_2)$
- $Exception = \{(s_1, s_2) | s_1 \in Exception_1\ or\ s_2 \in Exception_2\}$
- $T = \{((pre_1, pre_2), (post_1, post_2), fn, g, r) | ((pre_1, post_1, fn, g, r) \in T_1\ or\ (pre_2, post_2, fn, g, r) \in T_2) where (pre_1, pre_2), (post_1, post_2) \in S\}$.

Lemma 1 *If a set of methods is added to a class without affecting the set of member variables, the OSM of the class becomes a product of its existing OSM and the OSM of a class having same set of members but only new methods.*

Proof Let us assume the existing OSM of the class (say c_1) is $osm_1 : (S_1, start_1, T_1)$, the OSM of the class (say c_2) considering only the newly added method is $osm_2 : (S_2, start_2, T_2)$ and $osm_f : (S_f, start_f, T_f)$ is the OSM of the class (say c_f) constructed from all methods in the class including the new ones. We shall prove that osm_f is the product of osm_1 and osm_2. In order to do that, we need to show that all three conditions provided in Definition 6 are hold for this case.

1. First we shall prove $S_f = \{S_1 \backslash start_1 \times S_2 \backslash start_2\} \cup (start_1, start_2)$. The condition can only be true if for all $s_k \in S_f$, there exists a unique pair (s_i, s_j), where $s_i \in \{S_1 \backslash start_1\}$ and $s_j \in S_2 \backslash start_2\}$, such that $s_k.cond \Leftrightarrow s_i.cond \wedge s_j.cond$.
   Let V be an arbitrary vector of values assumed by member variables. Since S_1 and S_2 also defines partitions over the values of same set of member variables, if V satisfies $s_k.cond$, V must satisfy $s_i.cond$ for a unique $s_i \in S_1$, and $s_j.cond$ for a unique $s_j \in S_2$. Thus, it is proved that,

$$s_k.cond \rightarrow s_i.cond \wedge s_j.cond \tag{1}$$

   Each $MDL\ m_i$ appears in $s_k.cond$ also appears in $s_i.cond$ or $s_j.cond$, either as m_i or as $\neg m_i$, since if m_i is explored during construction of osm_f, it must have been explored during construction of osm_1 or during construction of osm_2. Now, if V does not satisfy $s_k.cond$, there exists at least one MDL, say m_j, in $s_k.cond$ which is not satisfied by V. Since m_j also appears in $s_i.cond$ or in $s_j.cond$, V will not satisfy at least one of these which contains m_j. Hence, if V does not satisfy $s_k.cond$ it fails to satisfy either $s_i.cond$ or $s_j.cond$. Therefore, it can be said that,

$$\neg s_k.cond \rightarrow \neg s_i.cond \wedge s_j.cond \tag{2}$$

   From Eqs. 1 and 2 it is concluded that

$$s_k.cond \Leftrightarrow s_i.cond \wedge s_j.cond \tag{3}$$

2. $start_f$ represents $(start_1, start_2)$ since $start_f.cond = false$ and both $start_1.cond = false$ $start_2.cond = false$, and thus, $s_k.cond \Leftrightarrow s_i.cond \wedge s_j.cond$ holds.
3. Consider a transition $(pre_k, post_k, fn, g, r) \in T_f$. Let us assume, during this transition, fn causes a change in values of member variables from V_{pre} to V_{post}, where V_{pre} and V_{post} are two vectors of values of member variables. Therefore, V_{pre} satisfies $pre_k.cond$ and V_{post} satisfies $post_k.cond$. We have already proved that there exists a unique $pre_i \in S_1$ and a unique $pre_j \in S_2$ such that $pre_k.cond = pre_i.cond \wedge pre_j.cond$. Therefore, V_{pre} satisfies both $pre_i.cond$ and $pre_j.cond$, and V_{post} satisfies both $post_i.cond$ and $post_j.cond$. Hence, there is a transition $(pre_i, post_i, fn, g, r)$ in osm_1 or there is a transition $(pre_j, post_j, fn, g, r)$ in osm_2. Thus, it is proved that for each transition $((pre_i, pre_j), (post_i, post_j), fn, g, r) \in T_f$ there is a transition $(pre_1, post_1, fn, g, r) \in T_1$ or $(pre_2, post_2, fn, g, r) \in T_2)$, where $(pre_1, pre_2) \in S$ and $(post_1, post_2) \in S$. □

Definition 7 (*Partial OSM*) The OSM of a class which is constructed only considering the methods that belong to the class (except those that are inherited from the parent class) is called partial OSM of the class.

Definition 8 (*Complete OSM*) The OSM of a class which is constructed considering all methods that the class can access (including those that are inherited from the parent class) is called partial OSM of the class.

Theorem 2 *The complete OSM of any class is a product of its partial OSM and the OSM of its parent class.*

Proof Let us assume a class c_j having a set of member variables I_j and a set of methods F_j extending another class c_i having a set of member variables I_i and a set of methods F_i. By extending c_i, c_j gains access to member variables of c_i and also methods of c_i. The set of member variables c_j can access is $I_i \cup I_j$. Though there can be some member variables of c_i that cannot be accessed from c_j (private variables), without loss of generality, it can be assumed that those variables are unused in c_j. Similarly, c_j can access $F_i \cup F_j$ methods assuming private methods of c_i are not used in c_j.

Now let the OSM of c_i be osm_i and the OSM of c_j without considering inherited methods is osm_j. Now the state model (say osm_f) of c_j considering inherited methods can be viewed as a state model of c_j after adding a set of method F_i with the existing set of methods (i.e., $F_i = F_i \cup F_j$). Hence, using Corollary of Lemma 1, it can be said that osm_f is of product osm_i and osm_j. □

4 Implementation

In this section, we discuss our state model extraction approach StateJ. StateJ statically analyzes Java bytecode and reverse engineers the states and transitions. To simplify the problem of reverse engineering state model, we assume the following:

- Programs do not contain static variables/methods.
- Member variables are always private and can be accessed through appropriate methods.
- There is no recursive method call.
- There is no circular dependency among classes.
- There is no concurrency/parallel execution of methods using threads.

Though the first two assumptions look serious since most of the real world programs would contain static variables/methods and public variables. However, any program having these kinds of features can easily be refactored so that it fits in the existing setup. For example, a public variable can be replaced with a private variable and a set of getter and setter methods. Similarly, static variables/methods can be handled by considering the class object associated with each object for state model extraction.

StateJ extracts state models in a modular fashion instead of carrying out full interprocedural symbolic execution of an input program. The OSM of each class is extracted individually, and it is reused during extracting OSM of a dependent class. The main advantage of modular analysis is reusability. The symbolic execution engine need not execute the same methods multiple times even if these are invoked in multiple classes. At first, StateJ analyzes dependencies among the classes of a Java program under consideration. It determines an ordering among classes in which state model should be constructed for individual classes. Subsequently, for each class, the state model is constructed by first symbolically executing all methods of the class and then identifying states and transitions by processing the path summaries.

4.1 Class dependency analysis

In StateJ, an OSM of a class c_i is reused during construction of OSM of another class c_j if one of the following conditions holds:

- c_j has an attribute which is a reference to c_i.
- c_i is used as parameter in the methods of c_j.
- c_i is instantiated locally in c_j.
- c_j extends c_i.

A *class dependency graph* (should not be confused with ClDG of Larsen et al. [13]) is constructed to represent above mentioned dependencies among the classes in a set. It is a directed graph in which each node represents a class, labeled by the class name; a directed edge from a node n_i to a node n_j denotes that the class represented by n_i is dependent on the class represented by n_j. Since we assume that, no circular dependency exists among the classes, no cycle exists in the *class dependency graph*. Consequently, the *class dependency graph* becomes a directed acyclic graph on which we compute a reverse topological order. In the reverse topological ordering of classes, a class c_i appears before a class c_j if there is a directed edge from the node representing c_j to a node representing c_i, i.e., c_j depends on c_i. The state model of the classes are constructed in the reverse topological order.

4.2 Symbolic execution of methods

In this step, each method of a class is executed symbolically in order to compute *path summaries* for all feasible paths. In StateJ, the role of the symbolic execution engine is to interpret various program features abstractly and encode these into *path summaries*. The symbolic execution engine is implemented by extending the JPF symbolic execution engine. It is capable of handling class inheritance, loops, exceptions and complex object interactions. In the following subsections we discuss how StateJ handles various features of a Java program.

4.2.1 Object manipulation

Object references can appear as class members, as method parameters or as local variables. For each object reference *ref* in the program, StateJ maintains two additional symbolic

Table 2 Classes and corresponding integer id

Class name	Integer Id
`unknown`	-1
`null`	0
`AddSub`	1
`UseAddSub`	2

```
11 class UseAddSub{
12  AddSub refA;
13  void meth1(){
14   AddSub refB=refA;
15   if(refB.meth(5,5)==10)
16    print("summed");
17   else
18    print("subtracted");
19 } }
```

Fig. 2 A class using an object of `AddSub` class shown in Fig. 1

variables of type `int`, called *ref.type* and *ref.state* so that conditional literals on these variables can be analyzed by a SAT solver. StateJ assigns a unique integer to each class in the program (refer Table 2). The variable *ref.type* holds the integer value that corresponds to the type of the object pointed by *ref* whereas *ref.state* holds the present state of the object pointed by *ref*. *ref.type* may hold '0' (`null`) to represent that no object is being pointed by *ref*. *ref.type* may also hold '-1' if the type of *ref* cannot be determined by the symbolic execution engine (bounded loop unrolling, etc.). Any operation on *ref* (such as, assignments, *null* checking, *instanceof* checking, initialization and method invocation) is interpreted as a manipulation of these two additional symbolic variables. Therefore, results of any complex operations on object references can be encoded into the *path summary* using these two special symbolic variables and the algorithms presented in Sect. 3 can be applied. Conditional literals using *ref.type*, *ref.state* are treated as MDL or PDL depending on whether *ref* is a member variable or it is a method parameter.

Lazy initialization As we pointed in Sect. 2.1, a reference variable *ref* belonging to method parameters or class members is not initialized until it is accessed by an instruction. When the execution first accesses *ref*, the reference is initialized non-deterministically with each possible compatible object types, in order to simulate possible real executions. The compatible object types of *ref* include *null* and also those object types that are derived from the declared type of *ref*. For example, if a class A extends another class B, during lazy initialization, a reference of type A will be non-deterministically initialized with null, an object of A and also with an object of B since the reference can hold any of these three at runtime.

StateJ uses JPF *choice generator* API to explore a path with all such choices of *ref.type*.

Assignments An assignment of an object reference into another (of the form *ref1=ref2*) is interpreted by copying the values of *ref2.type* and *ref2.state* into *ref1.type* and *ref1.state*, respectively.

Method invocation Invocation of a method on an object reference *ref* may update the member variables of the receiver object which may not be visible (when the method invocation changes private members of the receiver) in the sender object. The changes in those variables can be abstractly interpreted as a state change(i.e., a transition) of the receiver object. The invoked method causes such transition (say t) in the receiver object based on (1) the current state (current value of *ref.state*) and (2) the arguments passed to the method.

According to class dependency notion described in Sect. 4.1, the sender object is dependent on the receiver object, and therefore, the receiver object's state model should be available when another object (sender) invokes a method on it. If the invoked method belongs to a class for which state model is not constructed earlier (such as, for a JDK library class), a default OSM (refer to the Definition 6) is assumed for that class. From the OSM of the receiver object, StateJ identifies all transitions that are realizable due to this method call. After that, StateJ explores the current path, using JPF choice generator, assuming any of those transitions can occur at runtime. For each transition choice *(pre,post,fn,g,r)*, an atomic condition *ref.state==pre* is added into the current path condition. The method invocation changes the state of the object form *pre* to *post*. Therefore, the value of $ref.state$ should be updated with *post* after the method invocation. r is considered as the return value of the invoked method.

Example 5 Consider the class shown in Fig. 2, in Line 17, *refA* invokes a method `AddSub.meth(int,int)`. The method $meth$ can cause the following transitions: $t1, t2, t3, t4$ and $t5$ (refer Table 1). Therefore, execution should explore all these choices using JPF choice generator. Let us consider transition $t1 : (S3, S1, meth, arg2 > arg1, arg2 + arg1)$ as a choice taken by the execution engine. The path condition is updated with *refA.state == S3* (it becomes *refA.type==1* && *refA.state == S3* && (arg2 > arg1)), *refA.state* is updated with $S1$, and the return value of the method becomes 10 (since, $\text{arg1} + \text{arg2} = 5 + 5 = 10$).

4.2.2 Loop handling

Symbolic execution of programs with loops may potentially explore infinitely many paths. The existing symbolic execution engine in JPF unrolls a loop indefinitely which may result in non-termination. This is why, loops need to be specially handled in order to systematic exploration of feasible paths. In bytecode, loops are implemented using backjumps; an example is shown in Fig. 3. If a backjump instruction occurs and later, in the same path, an instruction is found which

```
...
while(i<5){
  s=args[i];
  i++;
}
...
```

```
 0: iconst
 1: istore
 2: iload
 3: iconst
 4: if_icmpge 17 //loop start
 7: aload
 8: iload
 9: aaload
10: astore
11: iinc
14: goto 4  //backjump
17: getstatic //loop end
```

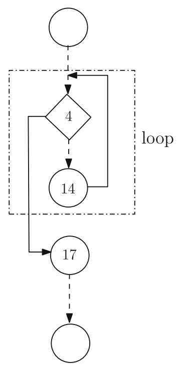

Fig. 3 A sample byte code representing a loop

has been executed earlier, a presence of loop is detected. The region between the jump location and the location of the backjump instruction is identified as the loop body. If a loop is detected in a path, StateJ unrolls the loop until k th iteration, where k is a user-defined parameter. If the loop does not terminate even after k iterations, the execution of the instructions inside the loop body are skipped. Since the number of loop iteration is not known a priori, the variables updated inside a loop are conservatively treated as *unknown*. StateJ assigns a special symbolic variable called *unknown* to all variables updated inside the loop body of a loop that iterates more than k times. If a method is invoked on a reference variable *ref*, *ref.state* is treated as unknown, and if an assignment instruction updates *ref*, *ref.type* and *ref.state* are both treated as *unknown*.

4.2.3 Inheritance

We have already established in Theorem 2 that, when a class extends another class, the derived class inherits the state transition behaviors also and the complete state model of the derived class becomes a product of its partial state model and the state model of its parent.

Interfaces/abstract classes cannot be instantiated, and thus, we do not extract state models from these. The methods/attributes of an interface/abstract class are flattened into the classes that implement/extend these according to their visibility in the derived class. For example, an abstract class having a `protected void m(){}` will be flattened into one of its derived classes as `private void m(){}`.

4.2.4 Exception handling

When execution of an instruction raises an exception and the handler location of the exception is available, the control is transferred to that location. In contrast, the exception is considered as *unhandled* and execution of the path is aborted if no handler is found for an exception. On occurrence of an unhandled exception, the *exception* attribute of *path summary* of the current path is updated with the type of the unhandled exception.

Example 6 Consider the Java class shown in Fig. 2, during lazy initialization if the type of *refA* is chosen 0 (*null*), a `NullPointerException` occurs at line 17. In that case, the *exception* attribute is updated with "NullPointerException"— the type of the exception that occurs.

4.2.5 Returned expression

For a method under symbolic execution, execution of the return statement is considered as the end of execution of the current path. Since elements of the *path summary* must not contain private variables of a class, the *returned expression* of a method need to be manipulated in case it contains internal variables of the class. We manipulate the *returned expression* by using the following heuristics:

1. If a *returned expression* is a constant it is retained as it is.
2. If a *returned expression* is composed solely of parameters of the method, it is retained as it is.
3. If a *returned expression* contains private member variables and the *path condition* of the current path have unique solution of those variables, the values obtained from the path condition are substituted in the *returned expression*. For example, if v_1, v_2 and v_3 are private member variables and *path condition* of the path is $(v_1 + 2v_2 = 3) \wedge (v_2 = 4) \wedge (v_3 > 50)$, by solving the *path condition* we get unique solutions for v_1 and v_2 $(v_1 = -5, v_2 = 4)$. Now if the *returned expression* is $2v_1 + v_2$, substituting values of v_1 and v_2 in the *returned expression*, we get -6 which does not contain private variables.
4. If a *returned expression* cannot be excluded from private member variables using the above steps, a symbolic variable called *unknown* is assigned into it.

4.2.6 Unknown values

The symbolic execution engine may have to handle *unknown* values since sometimes loops and method invocation on an object reference produce these. An *unknown* value is nothing but a symbolic variable of a specific type. When an *unknown* value is assigned to a variable, the variable is also treated as unrestricted in the sense it can hold any value of its domain. A symbolic expression consisting of a variable having *unknown* value is also treated as unknown. A conditional literal consisting of variables having *unknown* values is considered as *MXL* and thus ignored during creation states and transitions.

The presence of *unknown* values increases size of the model by increasing number of transitions. However, it does not lead to miss a transition since *unknown* is a superset of

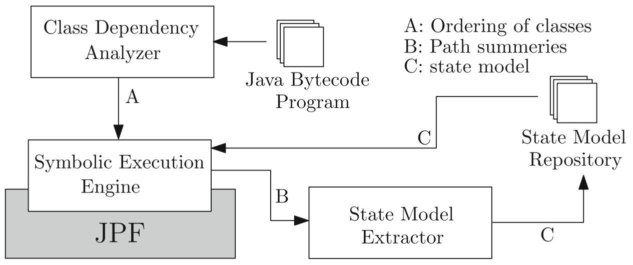

Fig. 4 Modules of StateJ

the exact value which cannot be determined by the symbolic execution engine.

4.3 Identification of states and transitions

At the end of execution of a path, *path summary* is computed which contains the path condition, name of the method under symbolic execution, name-value pairs of updated member variables and the returned expression.

Path summaries computed by the symbolic execution engine are analyzed in this step. Algorithm 1 is used to create states which include start state, normal states and exceptional states. After that, Algorithm 2 is used to create possible transitions including those that end in exceptional states (exceptional transitions).

After creating states and transitions, a depth first traversal is performed from the *start state*. The traversal yields the reachable states from the *start state*. The states those are not reachable from the *start state* cannot be assumed by the class at runtime. For this reason, these unreachable states as well as the transitions among these states are removed from the state model.

5 Experimental results

We have developed a prototype tool based on StateJ and carried out experimentation using a few Java programs. The modules of the developed tool are shown in Fig. 4. It has three modules namely: class dependency analyzer (CDA), symbolic execution engine (SEE) and state model extractor (SME). The CDA module is developed using Apache BCEL, a open-source bytecode engineering library. CDA module analyzes class dependencies and computes an ordering among the classes. The SME module is developed by extending symbolic execution engine of Java PathFinder [8]. It executes all methods of a class and produces a set of *path summaries*. Finally, the SME module constructs state model by analyzing path summaries. SME module also performs reachability analysis to remove unnecessary states and transitions.

As we have already discussed, we extract state models from a Java program in a bottom-up manner, by first extracting state models for independent classes and then for other classes those depend on the classes for which state model have been already extracted. To measure the complexity of a class from its source code, we have chosen four well-known metrics, namely: *number of member variables* (NV), *number of methods* (NM), *maximum value of cyclometric complexity* (MCC), and *estimated lines of code* (ELOC). For each program under study, we assumed that the programs has a class which acts as a main class (marked with a "(M)" after the class name, in the Table 3), the state model of which can be treated as the state model of the whole program. We also measured the complexity of the state model generated by our technique using following metrics: *number of states* (NS), *number of transitions* (NT), *number of exceptional states* (NES), *number of exceptional transitions* (NET) and *structural complexity of the state model* (CCS) which can be computed using the following formula:

$$\mathrm{CCS} = |\mathrm{NS}| + |\mathrm{NT}| + |\mathrm{NG}| + 2$$

where NS is the multi-set of transitions, NT is the multi-set of event triggers, and NG is the multi-set of atomic expressions in the guard conditions [22].

We have considered six Java programs that ranged from low to moderate complexities. We considered the following Java applications: *Television Remote Simulator*, *Vending Machine*, a *Queue implementation using Stack*, *Car Weather controller*, *eLibrary* and *Power Window Controller*. The details of these applications along with their source code complexities are tabulated in Table 3. The complexity of the extracted state models are tabulated in Table 4. The experiments were carried out with a moderately powerful hardware having Pentium 3 GHz CPU and 1 GB of physical memory. Execution time (in seconds) and physical memory used (in Megabytes) in the extraction process are also tabulated under the TIME and MEM column in Table 4.

As can be seen in Table 3, the complexity of the extracted state model increases with the increase in complexity of the source code of the input program. *Television Remote Simulator* is the simplest and *Power Window Controller* is the most complex one in terms of ELOC and MCC. The CCS values (refer Table 4) of the state model of these applications also indicate that *Television Remote Simulator* has the simplest state model and *Power Window Controller* has the most complex state model.

The application *Queue implementation using Stack* shows slightly different behavior. The state model of it has relatively more CCS value though it does not seem to have a complex source code. The reason is that, number of member variables (NV) are relatively high in the *Stack* class. Moreover, the stack class uses an array to implement its functionalities. For this reason, the number of member variables adds up with the

Table 3 Complexity of the chosen applications

Application name	Application details					
	NC	Class name	NV	NM	MCC	ELOC
Television remote	3	PowerState	3	5	3	61
		PowerState2	1	1	1	25
		Television (M)	4	9	3	65
Vending machine	2	Dispenser	3	3	5	43
		Vending machine (M)	5	6	3	82
Queue using stack	2	Stack	5	7	6	95
		Queue (M)	2	5	3	73
Car Weather controller	5	TemparatureSensor	2	3	2	20
		WeatherSensor	2	3	4	62
		WiperController	1	6	2	82
		ACController	5	7	4	130
		CentralController (M)	5	10	17	193
eLibrary	6	User	6	12	1	74
		InternalUser	1	2	2	23
		Book	0	2	1	21
		Document	6	18	2	92
		Loan	2	5	3	42
		Library(M)	4	19	3	192
Power window controller	3	PWCStates	5	7	3	85
		PWCEvents	12	14	3	119
		PWCMain (M)	4	16	6	675

Table 4 Experimental results

No.	Application name	NS	NT	NES	NET	CCS	TIME (s)	MEM (MB)
1	Television remote	11	18	1	1	37	3	13
2	Vending machine	11	32	1	3	56	5	17
3	Queue using stack	26	102	2	32	180	19	37
4	Car Weather controller	23	153	0	0	273	63	121
5	eLibrary	42	258	1	6	352	120	342
6	Power window controller	50	543	1	19	813	230	421

number of array elements take part in decisions, which in turn increases number of states and transitions for the class *Stack*. As the class *Queue* uses the class *Stack*, the large state space of *Stack* object also increases the state space of *Queue* object.

StateJ identifies exceptional states and exceptional transitions for several applications. Most of the application has only one exceptional state representing unhandled `NullPointerException`, though an additional exceptional state for `ArrayIndexOutOfBound` is detected for the application *Queue using Stack*.

From the above discussion, it can be concluded that StateJ can extract models from non-trivial Java programs having moderately complex source code complexities. No other approach, according to best of our knowledge, can automatically extract states and transitions from non-trivial Java programs.

A visualization tool can make use of these models for system understanding. Currently, we use GraphViz for visualizing these models. Figure 5 shows a GraphViz [7] screenshot of the object state model of *PowerState* class in *Television Remote Simulator* application

COTS (component off-the-shelf) components are usually released without source code and with informal specifications. Component vendors can release state transition specifications with the component to support easier integration of the component. StateJ can be useful for that purpose to generate state transition specifications for components written in Java. In addition, as we already have pointed out,

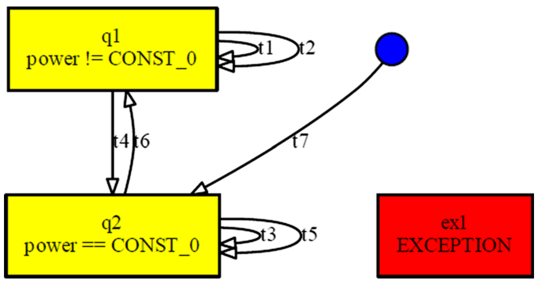

Fig. 5 Screenshot of the GraphViz visualization of the state model of PowerState class

StateJ can assist several regression test selection techniques in component-based environments [1,5,17].

As already have discussed, StateJ detects and models unhandled exceptions. Thus, StateJ can also be used to used to identify faulty method sequences that might lead to an unhandled exception.

6 Related work

Several research results have been reported in the area of reverse engineering of state model of a program from its implementation [2–4,10,12,18–20,26]. In this section, we present a brief review of these work and provide a comparison of our approach with these.

Kung et al. [12] executed the program symbolically to identify path conditions and computed intervals on the values of the member variables by analyzing the path conditions. They created transitions by analyzing how member variables are updated in a path. But, computing an interval becomes impractical when more than one member variable appears in an atomic condition. For example, consider the following conditions on integer domain: $a > 0, a == 5$. For these, the following intervals can be computed: $\text{INT\_MIN} \leq a \leq 0,\ 0 < a < 5,\ 5 \leq a \leq \text{INT\_MAX}$. But, if conditions include multiple variables, such as $a + b > 2, a + b + c == 2$, it is not always possible to compute intervals on individual variables. In contrast, we represent intervals (or partitions) abstractly in terms of boolean conditions: $(a + b > 2 \wedge a + b + c == 2)$, $(a + b \leq 2 \wedge a + b + c == 2)$, $(a + b > 2 \wedge a + b + c! = 2)$ and $(a + b \leq 2 \wedge a + b + c! = 2)$. Secondly, Kung's approach does not handle objects appearing in method arguments or the objects instantiated locally. Moreover, Kung et al. did not consider objects as parameters of methods. Due to these shortcomings, Kung et al.'s [12] approach cannot extract state/transitions from many applications used for experimentation (discussed in Sect. 5).

Tonella et al. [21] provided an approach for state model extraction from a class by static analysis. Walkinshaw et al. [23] also proposed a static analysis-based approach in which they identified transitions and also their functions from source code assuming that the states of the program is provided. Both of the approaches assumed that states are identified by manual inspection and transitions are created based on that. Our approach automatically identifies a set of states defined by the member variables of the class and creates transitions accordingly.

Koskimies et al. [11] proposed a technique for state chart extraction based on dynamic analysis. They synthesized sequence diagrams of the program from the traces collected from a large number of execution of the program and after that they synthesized state chart from the analysis of sequence diagrams. Xie et al. [26] and Dallmaier et al.[3] also presented dynamic analysis-based approaches to extract state model. Both of the approaches extract state transition behaviors from execution traces by describing states in terms of inspector methods of the class.

However, dynamic analysis reveals the behaviors only those that have been exercised by the test case executions. As a result, the accuracy of a reverse-engineered dynamic model depends on how the set of inputs are chosen to generate the traces.

Bandera [2] is another system that performs dynamic analysis to extract finite state models from source code. It extracts finite state models from Java source code and generates model checker specific representations. However, in a model checker specific finite state representation, states are defined on the set of all variables of a program and transitions are defined as state changes due to execution of an instruction. Consequently, the type of state model generated by Bandera is not an object state model and thus not directly comparable with our approach. Our approach extracts a finite state model which is an abstraction of an object of a class from an observer point of view. The potential use of our approach can be test case generation, test case selection, program understanding, etc.

Somé et al. [18] proposed a technique for reverse engineering the finite state model from C programs. Their technique extracts states and transitions by statically analyzing conditional constructs such as switch-case and if-then-else. However, their technique has been designed for procedural programs, and therefore, not suitable for object-oriented programs whereas our approach has been developed specifically targetted at object-oriented (Java) programs.

7 Conclusion

We have proposed an approach called StateJ to extract the state model of Java programs. The extracted state model can have several applications. For example, the model can be used for effective regression test selection in component-based environment or it can be used in program understanding in case of missing or obsolete design documents. StateJ represents occurrence of uncaught exceptions using exceptional states and exceptional transitions. So it is possible to identify method sequences for which a program might raise an exception. A prototype tool has been developed that implements our approach. However, the approach has several limitations as mentioned below: (1) size: the size of the state model in terms of number of states and transitions can be large. Imprecision introduced due to unknown values further increases the overall size of the model; and (2) performance: The time and memory requirements increase significantly as the complexity of the input program grows. To address the aforementioned issues, the following future works have been planned:

1. To introduce hierarchical and concurrent states to cope up with the size of the model, thus increasing comprehensiveness. Also, to develop an interactive UI to aid human users in code understanding.
2. Apply suitable heuristics to improve memory/cpu requirements.

Acknowledgments We are thankful to the anonymous reviewers who provided thoughtful comments on our work which, we believe, have greatly enriched this work.

References

1. Beydeda, S., Gruhn, V.: Black- and white-box self-testing cots components. In: International Conference on Software Engineering and Knowledge Engineering, pp. 104–109 (2004)
2. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Zheng, H.: Bandera: extracting finite-state models from Java source code. In: Proceedings of ICSE, pp. 439–448 (2000)
3. Dallmeier, V., Lindig, C., Wasylkowski, A., Zeller, A.: Mining object behavior with ADABU. In: Proceedings of the 2006 International Workshop on Dynamic Systems Analysis, WODA '06, pp. 17–24 (2006)
4. Guéhéneuc, Y.G.: Automated reverse-engineering of UML v2.0 dynamic models. In: Proceedings of the 6th ECOOP Workshop on Object-Oriented Reengineering (2005)
5. Gallagher, L., Offutt, J., Cincotta, A.: Integration testing of object-oriented components using finite state machines. Softw. Test. Verif. Reliab. **16**(4), 215–266 (2006)
6. Harel, D.: Statecharts: A visual formalism for complex systems. Sci. Comput. Program. **8**(3) (1987)
7. Graphviz—graph visualization software. http://www.graphviz.org
8. Java pathfinder. http://goo.gl/reTQe
9. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**, 385–394 (1976)
10. Koskimies, K., Mäkinen, E.: Automatic synthesis of state machines from trace diagrams. Softw. Pract. Exp. **24**(7), 643–658 (1994)
11. Koskimies, K., Systa, T., Tuomi, J., Mannisto, T.: Automated support for modeling OO software. Software, IEEE **15**(1), 87–94 (1998)
12. Kung, D.C., Suchak, N., Gao, J., Hsia, P., Toyoshima, Y., Chen, C.: On object state testing. In: Proceedings of Computer Software and Applications Conference, pp. 222–227. IEEE Computer Society Press (1994)
13. Larsen, L. Harrold, M.J.: Slicing object-oriented software. In: ICSE, pp. 495–505 (1996)
14. Logozzo, F., Fähndrich, M.: On the relative completeness of bytecode analysis versus source code analysis. In: Proceedings of CC '08, LNCS (2008)
15. Pasareanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M.R., Person, S., Pape, M.: Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In: ISSTA, pp. 15–26 (2008)
16. Peterson, J.L.: Petri Net Theory and the Modeling of Systems. Prentice Hall PTR, Upper Saddle River (1981)
17. Sen, T., Mall, R.: State-model-based regression test reduction for component-based software. ISRN J. Softw. Eng. **2012**, 1–9 (2012). Article No 561502. doi:10.5402/2012/561502
18. Somé, S.S., Lethbridge, T.: Enhancing program comprehension with recovered state models. In: IWPC, pp. 85–96 (2002)
19. Suman, R.R., Mall, R., Sukumaran, S., Satpathy, M.: Extracting state models for black-box software components. J. Object Technol. **9**(3), 79–103 (2010)
20. Systä, T., Koskimies, K., Müller, H.: Shimba-an environment for reverse engineering java software systems. Softw. Pract. Exp. **31**(4), 371–394 (2001)
21. Tonella, P., Potrich, A.: Reverse Engineering of Object Oriented code. Springer, Berlin (2005)
22. Wagner, S., Jürjens, J.: Model-based identification of fault-prone components. In: EDCC, pp. 435–452 (2005)
23. Walkinshaw, N., Bogdanov, K., Ali, S., Holcombe, M.: Automated discovery of state transitions and their functions in source code. Softw. Test. Verif. Reliab. **18**(2), 99–121 (2008)
24. Walkinshaw, N., Bogdanov, K., Holcombe, M., Salahuddin, S.: Reverse engineering state machines by interactive grammar inference. In: 14th Working Conference on Reverse Engineering, 2007. WCRE 2007, pp. 209–218. IEEE (2007)
25. Xie, T., Marinov, D., Schulte, W., Notkin, D.: Symstra: A framework for generating object-oriented unit tests using symbolic execution. In: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'05, pp. 365–381 (2005)
26. Xie, T., Notkin, D.: Automatic extraction of object-oriented observer abstractions from unit-test executions. In: ICFEM, pp. 290–305 (2004)

Tamal Sen obtained his M.S. degree from Indian Institute of Technology, Kharagpur. His research interests are software testing and verification. He also keeps a keen interest in latest web and mobile technologies.

Rajib Mall obtained all his professional degrees: Bachelor's, Master's, and the doctoral degrees from the Indian Institute of Science, Bangalore. He worked for a few years in the software industry before joining the faculty of Computer Science and Engineering at IIT, Kharagpur, where he is at present holds the position of a professor and Head of Department. He has guided 14 Ph.D. theses and has published more than 170 refereed international journal and conference papers. He has done consultancy projects for several organizations such as General Motors, Infosys, Honeywell, besides carrying out Government-sponsored projects. He works mostly in the areas of program analysis and testing for traditional as well as embedded software systems.