ISSN : 0974 - 7435 Volume 8 Issue 6

BioTechnology
An Indian Journal

FULL PAPER

BTAIJ, 8(6), 2013 [815-822]

Model-based testing for UML statechart diagram via extended context-free grammar

Liping Li[1]*, Tao He[2], Xiaolin Cao[1]

[1]Computer and Information Institute, Shanghai Second Polytechnic University, Shanghai, (CHINA)

[2]Software Engineering Department, Shenzhen Institute of Information Technology, Shenzhen, (CHINA)

E-mail: liliping@sspu.edu.cn; he_tao@foxmail.com; xlcao@sspu.edu.cn

ABSTRACT

This paper proposes an approach to checking the consistency and generating test cases from UML statechart specification through extended context-free grammar (ECFG) and model checking. UML statechart, test coverage criteria and ECFG are input to the system, to perform an automated consistency simulation and property verification for UML specification. ECFG is considered as external events and test coverage criteria are expressed as trap properties in CTL. A Simulation-Tree is introduced to simulate the execution of the system with the trigger events. The result of the simulation is the refined FSM consistent with the specification. Finally, test cases which satisfied with the specified test coverage are generated based on the refined FSM for UML statechart.

KEYWORDS

Model checking;
Consistency checking;
Context-Free grammar;
UML statechart diagram;
Test cases.

INTRODUCTION

Model checking is a model-based, property verification, automatic method to verify a system model with respect to its specification[1]. Unified Modeling Language (UML) is the current industrial de-facto standard to modeling the object-oriented system. UML statechart diagram can be used to construct software specification of object-oriented systems, embedded systems and reactive systems etc. As they are the most formalizable aspects of UML, statecharts provide a natural basis for test cases generation[2].

A test case is a set of conditions or variables which satisfied with certain test coverage criteria. With the help of test cases, a tester will determine whether a software system is working correctly or not. Test cases can be generated automatically from a formal system model. This paper presents an approach to checking the consistency of UML statechart specification by extended context-free grammar (ECFG) and generates test cases based on model checking. Our consistency testing system input UML statechart diagram and extended context-free grammars to perform an automated scenario simulation for consistency checking. First, we design an algorithm to transform statechart diagram to a FSM. Then, the consistency simulation is defined in the form of a derivation tree, called Simulation-tree, where each branch from the root to a leaf corresponds to a possible statechart run on a sequence of external event inputs, expressed by ECFG (Extended Context-free

FULL PAPER

Grammar). The result of the simulation generates a refined FSM which consistent with the specification. Finally, we generate test cases from the FSM according to the specified test coverage criteria expressed by the CTL (Computation Tree Logic).

The remainder of this paper is organized as follows: Section 2 presents a brief introduction to context-free grammar, model checking and UML statechart diagram. Section 3 introduces how to transform UML statechart diagram to FSM. Section 4 shows our approach of UML statechart consistency testing with ECFG. Section 5 generates test cases according to test coverage criteria and FSM. Section 6 discusses some related works for UML statechart diagram. Section 7 concludes this paper and discusses the future work.

THE PRELIMINARIES

Context-free grammar

Context-free grammar (CFG) is a set of recursive rewriting rules (or *productions*) employed to generate patterns of strings. CFGs can be used to describe the syntax of the input of the SUT (System Under Test). And, test cases can be generated conforming to the grammar of the context-free grammar.

A context-free grammar G is a quarter-tuple (V, Σ, W, P), where

- V is a finite set of non-terminals; each element $v \in V$ is called a non-terminal or a variable.
- Σ is a finite set of terminals, which corresponding to a set of external events, $V \cap \Sigma = \varnothing$.
- W is the start variable used to represent the whole program. It must be an element of V.
- P is a finite relation $V \rightarrow (V \cup \Sigma)^*$. The members of P are called the production rules of the grammar.

A language L is a context-free language if and only if there exists a context-free grammar *G* such that $L = L(G)$. Set Σ^* denotes the set of all strings over Σ including λ—the empty string.

Context-free grammars are an effective tool that can be used to generate test cases[3-5]. For example, Maurer[3] developed a data generation generator "dgl" which translates CFG into test generator and showed CFG is a remarkably effective tool that can be used to debug any program.

Model checking

Model checking starts with a model described by users, and discovers whether properties asserted by users are valid or not on the model[1]. In general, the model is finite state transition systems and the properties are temporal logic formulas, both of them can be modeled by the description language of a model checker [6]. Model checkers build a finite state system and exhaustively explore the reachable state space searching for violations of the properties being checked[1]. If the property fails, a counterexample is generated in the form of a sequence of states from its initial state to a state where the violation occurs.

Model checking is based on temporal logic. CTL (Computation Tree Logic) is a *branching-time* logic, meaning that its model of time is a tree-like structure in which the future is not determined[1]. The `verified` properties on model can be formulized by CTL. The syntax of CTL formulas can be defined by Backus-Naur form:

$\phi ::= \top \mid \bot \mid p \mid \neg\phi \mid \phi \wedge \varphi \mid \phi \vee \varphi \mid \phi \rightarrow \varphi \mid \mathbf{AX}\phi \mid \mathbf{EX}\phi \mid \mathbf{AF}\phi \mid \mathbf{EF}\phi \mid \mathbf{AG}\phi \mid \mathbf{EG}\phi \mid \mathbf{A}[\phi\ \mathbf{U}\ \varphi] \mid \mathbf{E}[\phi\ \mathbf{U}\ \varphi]$, where

- T and ⊥ is true and false, the atomic proposition *p* is a CTL formula.
- If ϕ, φ is CTL formula, then $\neg\phi, \phi \wedge \varphi, \phi \vee \varphi, \phi \rightarrow \varphi, \mathbf{AX}\phi, \mathbf{EX}\phi, \mathbf{AF}\phi, \mathbf{EF}\phi, \mathbf{AG}\phi, \mathbf{EG}\phi, \mathbf{A}[\phi\ \mathbf{U}\ \varphi]$ and $\mathbf{E}[\phi\ \mathbf{U}\ \varphi]$ is CTL formulas.

Pay attention to each of the CTL temporal connectives is a pair of symbols. The first of the pair is one of A or E. A means 'along All paths from the current state' and E means 'There Exists one path from the current state'. The second one of the pair is the temporal operator X, F, G, or U, meaning 'neXt state', 'some Future state', 'all future states (Globally)', and 'Until'.

`For example,` EF*q* denotes there is a reachable state satisfying *q*. AF¬*q* denotes along all paths, *q* is violated sometime in the future. If AF¬*q* is violated, then denotes there existing an infinite path where *q* always holds. This path is called a counterexample of AF¬*q*.

UML statechart diagram

UML statechart diagram shows a state machine with states, transitions, events and actions. Statechart diagrams mainly express the information of the state tran-

sitions and actions response for a finite state system according to external events inspires[13]. A transition is a relationship between two states, indicating a possible change from one state to another. Transitions allowed at each state, the events can trigger transitions to occur and the actions may perform in response to events.

A typical Web application, Online Flight Reservation System (OFRS), is used to introduce the statechart diagram briefly. The statechart diagram for OFRS is shown as Figure 1. There are four states except *start* and *end* states, *NoReserve*, *PartlyReserve*, *ReserveComplete*, *ReserveClosed*. State *start* is the only initial state in the statechart diagram. But there may exist more than one *end* states. And these states are changed by events. For example, when system stays at the *NoReserve* state and at this point the event *reserve* is triggered, the system will transit from state *NoReserve* to state *PartlyReserve*, and the action *reservedNum+=reserveNum* will be performed. In the model, *reservedNum* is a variable, means the number of reserved flight tickets, it is updated when the event *reserve*() and *return*() are performed. The guard conditions are in solid brackets "[]", such as [*reservedNum=vacancyNum*]. If event *reserve* be triggered and at the same time the guard condition [*reservedNum=vacancyNum*] is true, the state *PartlyReserve* will transfer to state *ReserveComplete*. When *cancelFlight*() event triggered, state *NoReserve* will transfer directly to *end* state. The more detailed introduction of UML statechart diagram can be found in[13,14].

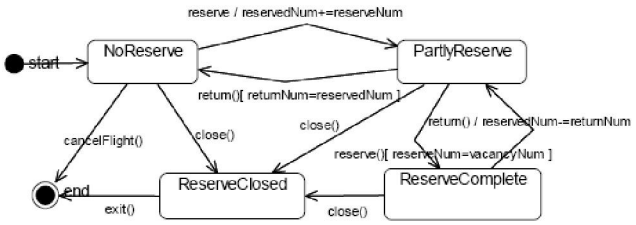

Figure 1 : An Online flight reservation system

CONVERTING STATECHART TO FSM

System specification is modeled by the statechart diagrams using IBM Rational Rose. Through analyzing, we know the UML diagrams are stored as a tree structure in the.mdl file. But the file is more than complex and redundant for read and test cases generation. So, we design an algorithm to draw the key information of statechart diagrams. We found a very simple rule in statechart diagram, all relevant statechart information is included in the directory of corresponding label, such as "(object State", and organizes in the "()" block, and so on. Therefore, we use a dynamic matching extraction method to draw the useful information, and stored in strings. Then we write these strings to XML file conforming to *state* label and *transition* label.

For the hierarchical and concurrent statechart diagrams, we need to flatten them. All useful information for the statechart diagram are stored in labels <State> </State> and <Transition> </Transition>, respectively.

A FSM is defined as quarter-tuple $T = (Q, \Sigma_T, \delta, q_0)$, Where,

- Q is a finite set of states;
- Σ_T denoted as a finite set of input `alphabet`;
- δ is a transition function, if q, q' $\in$ Q and σ Σ_T, then $q' = \delta(q, \sigma)$;
- $q_0 \in Q$ is the initial state.

From the definition of the FSM, we can convert UML statechart diagrams to FSMs as follows:

- The states of UML statechart diagram are corresponding to the states Q of FSM;
- All the input symbols and transitions in the UML statechart diagram are converted to the alphabet Σ_T and δ in FSM, respectively.
- The initial state of the UML statechart diagram is corresponding to q_0.

In the FSM, nodes represent states and edges denote transitions between states. According to the characteristic of statechart diagram, there is a unique start

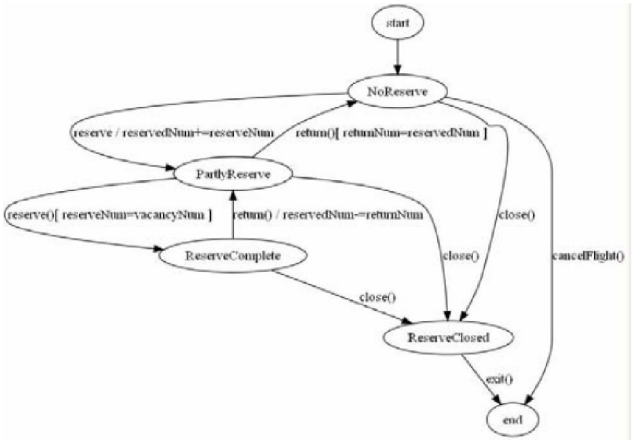

Figure 2 : The Direct diagram for OFRS

node that corresponds to the initial state and one or more end nodes represent the final states. We use Graphviz[15] to visualize the FSM. Figure 2 is the FSM automatically converted from statechart diagram for the Online Flight Reserve System based on our refined XML file. The FSM explicitly expressed flows of the statechart diagram.

CONSISTENCY TESTING FOR STATECHART USING ECFG

The ECFG for OFRS Example

In this paper, context-free grammar simulates the moves of the FSM. We extended CFG by " represents a set of external events with guarded conditions. CFG describes the syntax of the input of the SUT and test cases can be generated conforming to the syntax of the CFG. Referred to[4,16], we extend context-free grammar with guard conditions along state transitions, actions and scripts along transitions or alternate productions, called it extended context-free grammar (ECFG). Our works focus on the consistency checking and test cases generation from UML statecharts based on ECFG. Events of UML statechart diagrams are considered as terminals in ECFG, and the states are assumed to be non-terminals of ECFG, where the initial state of the UML statechart diagrams is the start variable in ECFG.

For the OFRS example, the ECFG is shown as follows:

- *start* variable W= NoReserve,
- non-terminal set V= {NoReserve, PartlyReserve, ReserveComplete, ReserveClosed, end};
- terminal set Σ = {*reserve*(), *return*(), *close*(), *cancelFlight*(), *exit*()},
- and P is a set of production rules defined as follows:

  W → ε | *close*() A | *reserve* () B | *cancelFlight*()
  B→*reserve* () [*reservedNum=vacancyNum*] Y
  B→*return*()[*returnedNum=reservedNum*] W
  B→*close*() A
  Y→*return*() | *close*() A
  A →*exit*()

The Simulation-tree

Our simulation system, inputs an UML statechart diagram specification *S* and a ECFG *G*, mainly check whether the transitions of statechart diagram will react consistently to any external event sequence $w \in L(G)$. In our system, each state of the FSM corresponds to one non-terminal symbol of the ECFG *G* and each transition of the FSM corresponds to one production rule of the ECFG. Refer to paper[16-18], we also introduces a derivation tree, called a Simulation-tree, to systematically simulate all possible moves of FSM running with the ECFG. Given an external event *e*, continuously executes the steps associated with enabled internal events until the simulation system reaches a state where no further internal or hidden events can be carried out. If there exists a corresponding Simulation-tree with all success branches, then the specification of UML statechart diagrams is consistency. A triplet

(Q, W, B),

Where *Q* is the state of the FSM, *W* is the unread part of the input string of ECFG, and *B* denotes by *True* or *False* whether the state is a violating one, is called an instantaneous description (ID) of the simulator.

A move from one ID to another will be denoted by the symbol |-, there are two kinds of move in the system.

(1)Terminal move

If *e* is an external/terminal event, that is, $e \in \Sigma$ and $W \in \{ \Sigma \cup V\}^*$,

$(Q_1, eW, B_1) \vdash (Q_2, W, B_2)$ is possible if and only if $(Q_2, B_2) \in \delta((Q_1, B_1), e)$.

(2)Nonterminal move

If *E* is a non-terminal variable in V, $(Q, EW, B) \vdash (Q, D_1 \ldots D_n W, B)$ is possible if and only if $E \to D_1 \ldots D_n$ is an ECFG production in *P*.

A same ID can move to different terminals or non-terminals because of multiple CFG productions from the same variable to represent different transitions.

Definition 1 (Simulation-tree). Let *T* be a FSM and *G* an extended context-free grammar, the Simulation-tree for (*T, G*) is defined as follows:

- Each node of the tree is an instantaneous description (ID).
- The root node is $(Q_0, V_0, False)$, where Q_0 is a set of initial states, V_0 is the start variable in the ECFG *G*.
- Let (*Q, W, B*) be a node in the tree. This node has

an arc to one of his children (Q_1, W_1, B_1) for each possible move (Q, W, B) |- (Q_1, W_1, B_1). W is the unprocessed part of the grammars of ECFG.

- The following nodes have no children:
  I. (Q, W, *True*) denotes as a violating leaf node,
  II. (Q, ε, *False*) denotes as a success leaf node.

Consistency checking of the statechart

Consistency of the statechart specification requires that in each state, only a single transition is triggered by a given event[8]. The Simulation-tree can simulate the running of UML statechart diagram with the input of extended context-free grammar. The Simulation-tree for testing the Online Flight Reserve system is shown as Figure 3. Figure 3 uses a derivation tree illustrating how the running states move during the simulation, where the bold solid box and the dashed one denote a leaf node and a variant node, respectively. A node is called a variant of another if both nodes in the Simulation-tree have the same triplet running state[16,17].

The abbreviations from S0 to S4 represent five different object states during the simulation, where,

S0: NoReserve, S1: PartlyReserve, S2: ReserveComplete, S3: ReserveClosed, S4: end

In the Simulation-tree, a non-terminal move is labeled with an applied production rule, such as W '!*reserve*() B, while a terminal move is labeled with the corresponding leftmost terminal, such as *reserve*(). The UML statechart diagram is consistent if all the finite branches in the Simulation-tree are success branches. So, the consistency testing of statechart specification is a traversal of its Simulation-tree to check whether there have any nodes which *B* equals to *True*. Due to the recursion nature of the context-free grammar, there may exist many infinite branches in the Simulation-tree, or the finite branch could be any long[14]. In this paper, we use the depth-first search to traverse the Simulation-tree. When traversing, we add a flag to remember if an ID is visited; any repeated IDs visit later during the traversal will not be explored again. From Figure 3, we can draw a conclusion: the UML statechart diagram of WFRS example is consistency for the corresponding Simulation-tree has all success branches.

The result of the simulation will generate a refined FSM which consistent with the specification. Then we generate test case automatically based on the refined FSM. Test case generated in our system will be the effective because this FSM is the result of consistency simulation.

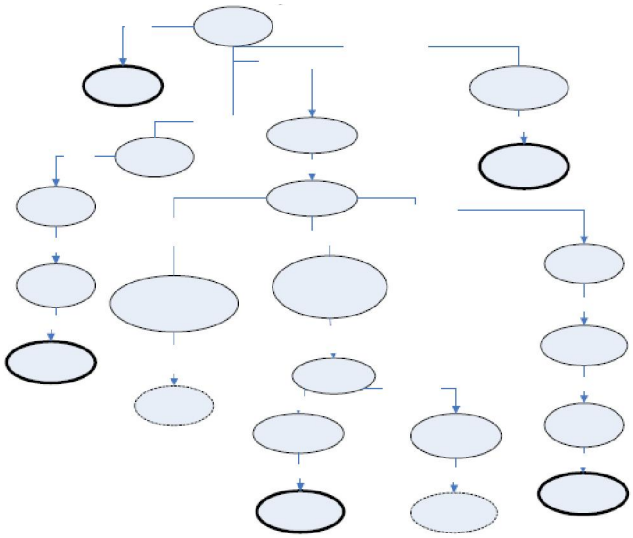

Figure 3 : The Simulation-tree for OFRS

TEST CASES GENERATION]

A test case is a set of external event sequences which satisfied with certain test coverage criteria. Test cases can be generated automatically from a formal system model. And test cases generated can be fed into the SUT for conformance testing[18]. Our work focuses on generating test cases while the UML statechart consistency checking.

A test criterion is a rule or collection of rules that impose test requirements on a set of test cases. Test criteria mainly answer these questions: what should be tested, how the testing goals could be achieved and when to stop. In this paper, we only concentrate on the *transition coverage criterion* and *state coverage criterion* for statechart diagrams.

The model checker SMV[19] takes a model and the properties to be verified as input, a counterexample will be generated when the property is discovered to be *false*. A counterexample is a sequence of states which start from initial page to the state that violates the property.

State coverage

Test cases *TS* satisfy the state coverage *iff* all states of the system must be covered at least once, i.e., *TS*

enable each variable of the model to take all possible values in its domain at least once. State coverage can be considered as a reachability property. Reachability property for a state ***s*** is defined in CTL formula with EF operator: EF *s*, The trap property is defined by AG operator: AG¬*s*. All trap reachability properties of OFRS in Figure 1 are listed in TABLE 1. We omit the start and end states in TABLE 1.

TABLE 1: Reachability properties for OFRS

No.	State	Trap property
S_1	NoReserve	**AG¬** NoReserve
S_2	PartlyReserve	**AG¬** PartlyReserve
S_3	ReserveComplete	**AG¬** ReserveComplete
S_4	ReserveClosed	**AG¬** ReserveClosed

The detail of the model checker SMV to generate the counterexamples can be found in paper[6,19]. The test sequences which satisfied state coverage criterion for Online Flight Reservation System is shown as follows:

NoReserve→*reserve* () / *reservedNum*+=*reserveNum* PartlyReserve →*reserve*() [*reservedNum=vacancy-Num*] ReserveComplete→ *close*() ReserveClosed→*exit()* end;

Transition coverage

Test cases *TS* satisfy the *transition coverage iff* all transitions of the system must be covered at least once. In general, behavioral claims include safety claims and liveness claims. A notion of liveness means that, under certain conditions, something will ultimately occur. Transition coverage properties denotes the *liveness* of the model, i.e., the system implements all the legal transitions. The liveless property is

EF(*state-s*→ (*transition-s*' ∧ EX *state-s*'))

The corresponding trap property can be represented as

AG (*state-s* →¬ (*transition -s*' EX *page-s*'))

For example, one of the *liveness* properties for the OFRS example is

AG (NoReserve→ ¬ (*close* ∧ EX ReserveClosed))

The other *liveness* properties for the OFRS example are similar with above; we do not list all of them in this paper. The test sequences generated satisfying the transition coverage criterions for Online Flight Reservation System are shown as follows:

- NoReserve→*reserve* () / *reservedNum*+=*reserveNum* PartlyReserve →*reserve*() [*reservedNum=vacancy- Num*] ReserveComplete →*close*() ReserveClosed→ *exit()* end;
- NoReserve → *reserve* () / *reservedNum*+=*reserveNum* PartlyReserve → *return*()[*returnedNum=reservedNum*] NoReserve → *close*() ReserveClosed → *exit()* end;
- NoReserve → *reserve* () / *reservedNum*+=*reserveNum* PartlyReserve '!*reserve*() [*reservedNum=vacancy- Num*] ReserveComplete → *return*() / *reservedNum=returnedNum* PartlyReserve → *close*() ReserveClosed → *exit()* end;
- NoReserve → *cancelFlight*() end;

RELATED WORKS

A lot of works[2,7-12] have focused on the automatic testing of UML statechart diagram. Offutt et al[2] generates test cases automatically from UML statechart diagram at the system level testing. They developed several useful coverage criteria based on UML statecharts, like transition coverage, full predicate coverage and transition-pair coverage. Kansomkeat et al[7] proposed a method for generating test sequences using UML statechart diagrams. They transform the statechart diagram into a flattened hierarchical structure of states called Testing Flow Graph (TFG). Test cases are generated by traversing the TFG from the root node to the leaf nodes. Kim et al[8] introduced a method to generating test cases for class testing using UML statechart diagrams. They first transformed statechart diagrams to extended finite state machines (EFSMs) to generate test cases based on control flow. Then, they transformed the EFSMs into data flow graphs, to which conventional data flow analysis techniques can be applied. R Swain et al[9] also proposed an approach to generate test cases from UML statechart diagrams. First, they constructed the statechart diagram for a given object. Then the statechart diagram is traversed using DFS al-

gorithm, conditional predicates are selected and these conditional predicates are transformed to source code. Then, the test cases are generated and stored by using function minimization technique. Andrews et al[10] used FSM with constraints to model and test web applications. They model system with hierarchical aggregate FSM in which the FSM transitions are compressed by selecting a reduced set of inputs in an input constraint language. Murthy et al[11] proposed a test ready UML statechart model. Various test cases can be generated automatically from this test ready model by determining all the sentential forms derivable from an equivalent extended context free grammar model.

Some of above work do much help to the test case generation from UML statechart diagrams. But they all first assume the specification modeled by UML statecharts is correctness. Unfortunately, the specification is often incomplete, inconsistent and ambiguous because it is usually constructed by the cooperation of users, domain experts and system engineers[12]. There exist consistency problems in UML behavioural models for the hierarchy and concurrency features of statechart diagrams. Errors of the specification will cost much more expensive to amend in the later phases of the development life cycle. So, the early consistency checking of the UML specification is significant.

CONCLUSIONS AND FUTURE WORKS

This paper presents a method of consistency checking and test cases generation for UML statechart specifications based on extended context-free grammar (ECFG) and model checking. Our goals are checking the consistency of statechart diagram and generate test cases while the consistency simulation going on. Our consistency testing system input UML statechart diagram and ECFG to perform an automated scenario simulation for consistency checking. The result of the simulation generates a refined FSM which consistent with the specification. Then test cases are generated automatically based on the refined FSM and the coverage criteria expressed by the CTL. Test cases generated in our system will be the effective test cases because they are the result of consistency simulation.

Our future works include improving our system prototype and generating test cases using symbolic grammars and model checking.

ACKNOWLEDGMENT

This work was supported by a grant from Shanghai Second Polytechnic University Key Discipline Development, Software Engineering (XXKZD1301), Computer Application Technology (XXKPY1301), in part by a grant from Natural Science Foundation of Guangdong Province, China (S2011040000672), Guangdong Provincial Education and Science Project of the 11th "five-year plan"(2010tjk411).

REFERENCES

[1] M.P.E.Heimdahl, S.Rayadurgam etc.; Auto-generating Test Sequences Using Model Checkers: A Case Study, Third International Workshop on Formal Approaches to Testing of Software (FATES 2003), Spinger, Montreal, Quebec, Canada, October, 2003, 42-59 **(2003)**.

[2] J.Offutt, S.Liu, A.Abdurazik, P.Ammann; Generating test data from state-based specifications, Software Test Verification and Reliability, **13**, 25-53 **(2003)**.

[3] Peter M.Maurer; Generating testing data with enhanced context-free grammars, IEEE Soft-ware, **7(4)**, July **(1990)**.

[4] L.P.Sobotkiewicz; A New Tool for Grammar-based Test Case generation, Master Thesis. University of Victoria, **(2004)**.

[5] A.G.Duncan, J.S.Hutchison; Using attributed grammars to test designs and implementations, In: International Conference on Software Engineering (ICSE'81), 170-178 **(1981)**.

[6] H.Miao, H.Zeng; Model Checking-based Verification of Web Application. In Proceedings of the 12th IEEE international Conference on Engineering Complex Computer Systems (ICECCS 2007), July 2007, IEEE Computer Society, Washington, DC, 47-55 **(2007)**.

[7] S.Kansomkeat, W.Rivepiboon; Automated-generating test case using UML statechart diagrams, In In Proceedings of SAICSIT, ACM, 296-300 **(2003)**.

[8] Y.G.Kim, H.S.Hong, D.H.Bae, S.D.Cha; Test cases generation from UML state diagram, Software Testing Verification and Reliability, 187–192 **(1999)**.

FULL PAPER

[9] R.Swain, V.Panthi, P.K.Behera, D.P.Mohapatra; Automatic Test case Generation From UML State Chart Diagram, International Journal of Computer Applications (0975-8887), **42(7)**, 26-36 March **(2012)**.

[10] A.Andrews, J.Offut, R.Alexander; Testing web applications by modeling with FSMs, Systems and Modeling, **4(3)**, 326-345 July **(2005)**.

[11] P.V.R.Murthy, P.C.Anitha, M.Manish, R.Subramanyan; Test ready UML state chart models", In Proceedings of international workshop on Scenarios and state machines: models, algorithms, and tools (SCESM '06). ACM, New York, USA, 75-82 **(2006)**.

[12] Zs.Pap, I.Majzik, A.Pataricza, A.Szegi; Completeness and Consistency Analysis of UML Statechart Specifications, Proc. Of IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop, 83-90 **(2001)**.

[13] J.Rumbaugh, I.Jacobson, G.Booch; The Unified Modeling Language Reference Manual. Addison-Wesley, **(1999)**.

[14] Grady Booch, James Rumbaugh, Ivar Jacobson, Unified Modeling Language User Guide, (2nd Edition). Addison-Wesley, **(2005)**.

[15] Graphviz-Graph Visualization Software. http://www.graphviz.org.

[16] H.F.Guo, W.Zheng, M.Subramaniam; L2c2: Logic-based lsc consistency checking, In 11th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP), 183-194 September **(2009)**.

[17] H.F.Guo, W.Zheng, M.Subramaniam; Consistency checking for lsc specification", In 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2009), 119-126 July **(2009)**.

[18] Songqi Liu, Liping Li, Hai-Feng Guo; Generating Test Cases via Model-based Simulation, In 13th IEEE International Conference on Information Reuse and Integration (IRI2012), in press, **(2012)**.

[19] K.L.McMillan; The SMV System for SMV version 2.5.4, http://www.cs.cmu.edu/~modelcheck/smv/ smvmanual.ps.