REACHABILITY IN A FINITE DISTRIBUTED SYSTEM PROTOCOL MODEL BY BACKWARD TRAVERSAL

Tapas Samanta,VECC, Kolkata, India
Dipankar Sarkar, IIT, Kharagpur, India
Samarpita Mukherjee, Jadavpur University, India

Abstract

Distributed system protocol verification has the intrinsic problem in mechanizing the reasoning pattern and the resultant state space exploration. The former arises in case of theorem proving approach due to the ingenuity involved in constructing a proof and the latter is encountered in model checking approach while carrying out composition of a large number of processes that constitute a typical distributed system. A combined approach of the above two methods has been devised that eventually considers the reachability in finite distributed system protocol model. It computes the reachability in backward traversal on the fly. In this paper a C++ implementation of the on-the-fly backward traversal algorithm is reported.

INTRODUCTION

A wide class of distributed system protocols comprises a large no. of identical processes, each with an identical behaviour having finite state-space. Behaviour of each of the processes can be captured by a state transition diagram defined in terms of a FSM structure. Since a distributed system model is having a state-space of the order $\mathbf{m^n}$ where **m** is the state-space of an individual constituent process and **n** is the number of constituent processes which is typically few hundreds. Due to this state-space explosion [1, 2] while composing the FSMs for large number of processes, the reachability analysis suffers from large computational complexity. The framework [3] explores the merits of tableau proof framework and devises a new backward traversal algorithm to improvise a completely mechanized verification technique containing the state-space explosion problem for distributed system protocols having identical participants. In [3] the observation is made that if a flat composition can be avoided and selective composition, as and when necessary, can be computed, then the state-space explosion can be contained to a great extent. In this paper, we present a C++ implementation of the backward traversal process which basically tests on-the fly the reachability of an atomic predicate in a given distributed system protocol model.

The paper is organized in the following way: Section 2 briefly explains the C++ implementation details of the Modified Backward Traversal Algorithm followed by an example of the implementation of the Backward Traversal Algorithm in the Section 3.

A C++ IMPLEMENTATION OF THE BACKWARD TRAVERSAL FOR LEADER ELECTION PROTOCOL

Data Structure Used in the Modified Algorithm

1. Class Table - In this program, for the FSM (Finite State Machine) structure of the processes, we take the input from fsm.data file and have stored it in thee class Table. The class Table contains static vector containers (dynamic lists) those store the state, transition, guard and action conditions of the transitions as state, transition, guard and action objects respectively
2. Class Network - To store the Network structure of the Distributed system, we have used class Network to store the input from the NetworkFile.data
3. Class triplet - To store the Triplet node that contains a node's information in the form, <EndState, Process, TransitionIndex> we have declared the class triplet. In the class Table, there is a static vector <triplet> Triplet list that stores all the nodes of the computational paths.
4. Class tranCont - The class tranCont stores index of two transitions (in vector<transition> Table::transitions) those are contradictory to each other. The static vector<tranCont> Table::TranContradiction stores all the contradictory transition pairs of the system.
5. Class stateCont - The class stateCont stores index of two states (in vector<state> Table::states) those are contradictory to each other. The static vector<stateCont> Table::StateContradiction stores all the contradictory state pairs for any process of the system.
6. Class stReachability – The class stReachability stores the lists of states of each process that has been reached so far. This helps to check if for any computational path, a process's current state contradicts with the states that had been so far reached by that process. This keeps track of the process's states so that if there is any state contradiction, then the path can be flagged as invalid.

Functional Construct of the Modified Backward Traversal Algorithm

Let ξ be the set of states where atomic predicate $\neg p_i$ holds. For each $s_j \in \xi$, construct a backward forest by invoking the functions:

1. i) FSM * fsm= new FSM("/home/.... /fsm.data"); // Creates objects of the FSM class taking input from fsm.data file

Copyright © 2012 by the respective authors

ISBN 978-3-95450-124-3

ii)Network* net=new Network("/home/…./ networkFile.data"); // Creates objects of the Network class taking input from networkFile.data file

iii) Forest::Forest (FSM * fsm); // Create object of the Forest class passing FSM object.

2. i) void Forest::createRootNodes(atomicPredicate AP, Network *net); // Checks at which states atomicPredicate AP holds by calling the function vector<int> FSM::statesAPHolds(atomicPredicate AP); and create Triplet Root nodes (in the form <EndState, Process, TransitionIndex>) for each of these states.

   ii) vector<int> Class triplet::childIndex; // Stores the index of child nodes in Table::Triple with parentIndx field that stores index of the parents of these child nodes.

3. i) void Forest::insert_Contradiction(Network * net); // Checks for contradictory states pairs and transition pairs by calling the function bool transition:: find_Contradiction(transition& x); and stores in Table::StateContradiction and TranContradiction; respectively.

   ii) bool transition::find_Contradiction(transition& x); // Calls the function action::compare_Contradiction(guard grdCond); to check if Action of transition Ti contradicts with Guard of transition Tj.

4. void Forest::initPath(Network * net); // Creating children of the Root Triplets nodes based on message passing.

5. void Forest::createPath(Network * net) –

   i) This function is recursively creating the children of the leaf triplet nodes (with childIndx.size()==0). This function creates child tripletNode similarly as the previous function void Forest::initPath(Network * net) but in a recursive way until all the tree branches close with initial states S1. As it is an Inverted path, the triplet <EndState, processId, TransitionId> contains the end state of the transition, so to find the initial state, we are looking for transition T1 (tranId==0), which will lead to the initial state S1.

   ii) While creating each new child node (Figure 4), this function checks if the transition pair of parent and child triplet node, is a contradictory transition pairs.

   iii) After all the inverted paths (from final state to initial state) are created, we check the list Table::Triplet to find a triplet node representing initial state (i.e. transition id==0 and end state==1) and store the index of these triplet nodes with initial state in a list vector<int> PathRoot[2].

   iv) bool Forest::checkSt(int es, int pno) – While creating this inverted paths, in every node, we check the States reached by that process in the triplet node, by calling the function bool Forest::checkSt(int es, int pno). If there is any contradictory state reached, we discard that path by setting a flag, and if no contradictory state has been reached, then we store the new state of that process in the new child triplet node in the list Table::StateReached

6. This algorithm ends when all the branches of the triplet trees are closed.

Implementation Difficulties

During the implementation of the Backward Traversal Algorithm [3], following difficulties were faced.

1. This algorithm [3] is modelling a distributed scenario. Each tree node has a list of pointers to his child nodes and a pointer to its parent node. If we implement the algorithm exactly the way it is, then we have to move back and forth between parent and child nodes again and again to update the composite state of the current node, depending upon the value of the composite state of either the child node or the parent node of the current tree node and the vice versa. These increase the complexity of the program with the increase in the number of processes in the system.
2. During the Step 4. Refine Parents, and Step 6. Refine Ancestor [3], there can be more than one possible state in which a process can be, and if that is the case, then a node can have more that one incarnation. This causes the updating of not only those nodes, but also updating of the all the parent pointer information in all there child nodes and the child pointer information in the parent nodes of the newly incarnated nodes, and creation of new edges from that node to all its parent and child nodes. These situations cause the original tree to branch out and may split to create forests.
3. Using the child node pointers, it is easy to traverse downward in the computational tree branches, but it is complicated to traverse upward in the tree branches using parent pointers. So in case of a split in the computational branch, it becomes difficult to update all the composite states of the newly created branches or trees of the forest.
4. Here in this algorithm each node has a composite state, where each process in the system is in some state. But to find the reachability, we don't need the state information of all the processes at each node, for all possible tree branches as, in the future steps most of these branches would be purged.

CASE STUDY: AN UNIDIRECTIONAL RING WITH THREE PROCESSES

To find out the reachability of state where the atomic predicate L(2) holds, we start with the triplet state of the form <State_Id, Process_Id, Transition_Id> where L(2) holds. From Figure 1 we notice that S4 and S5 are the only such states for which L(2) holds, and for state S4 and S5 the respective enabled transitions are T4 and T5. So we start with two Root Triplet nodes N1 = < S4, 2, T4 > and N2 = < S5, 2, T5 > from which the backward traversal has to be taken up.

Step 1: Create Roots: Designate the Roots as N1 = < S4, 2, T4 > and N2 = < S5, 2, T5 >.

Step 2: Initialize the Paths: From Figure 2 we can see that action of transition T4 of process 2 enables the guard condition of transition T2 of the neighbouring Process 1

ISBN 978-3-95450-124-3

Copyright © 2012 by the respective authors

with state S2. Similarly action of transition T5 of Process 2 enables the guard condition of transition T3 and T4 of the neighbouring Process 1 with state S3 and S4 respectively. This creates child Triplet node N3 = < S2, 1, T2 > of node N1 and child Triplet node N4 = < S3, 1, T3> and N5 = < S4, 1, T4 > of node N2.

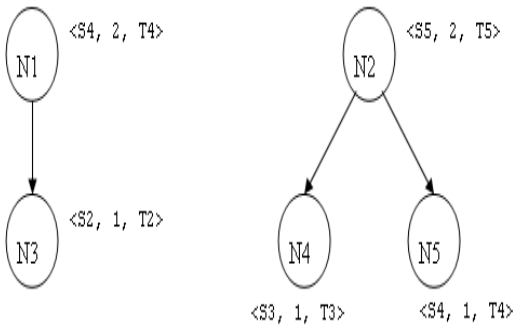

Figure 1: Step 2 Initialize the Paths.

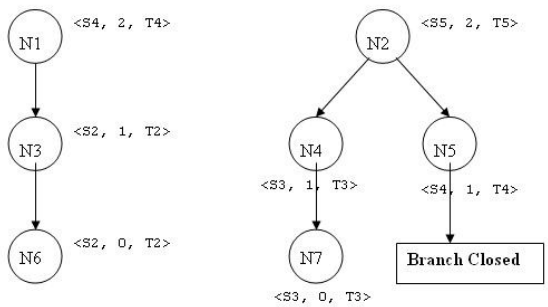

Figure 2: Step 3 Create Paths.

Step 3: i) Create Paths: Now similar to the previous step we keep creating children of the leaf nodes in the branches. So we create the child nodes of node N3, N4 and N5. While creating children of the leaf node, we find that from node N5 = < S4, 1, T4 > no child node can be created as the action of transition T4 of process 1 (in node N5) do not enables the guard condition of any transition without creating contradiction. So this branch with leaf node N5 is closed unsuccessfully. This creates child Triplet node N6 = < S2, 0, T2 > of node N3 and child Triplet node N7 = < S3, 0, T3> of node N4.

Step 3.i): Now we keep creating the child nodes of the leaf nodes in an iterative way, until the branches are successfully or unsuccessfully closed. The child node of N6 is N8 = < S2, 2, T1 > and N9 = < S2, 2, T2 > and child node of N7 is N10 = < S4, 2, T4 >. Here the branch with leaf node N8 is closed successfully as the Process 2 is in state S2 with transition T1 which will change the Process 2's state to initial state S1. So this branch is closed successfully (Figure 3).

Path 1: <S2, 2, T1> -> <S2, 0, T2> -> <S2, 1, T2> -> <S4, 2, T4>

Step 3.ii): Similarly we find that the branches with leaf node N15 = < S2, 2, T1 > and N16 = < S2, 2, T1 > are also closed successfully |(Figure 5).

Path 2: <S2, 2, T1> -> <S2, 0, T2> -> <S2, 1, T2> -> <S2, 2, T2> -> <S2, 0, T2> -> <S2, 1, T2> -> <S4, 2, T4>

Path 3: <S2, 2, T1> -> <S2, 0, T2> -> <S2, 1, T2> -> <S4, 2, T4> -> <S3, 0, T3> -> <S3, 1, T3> -> <S5, 2, T5>

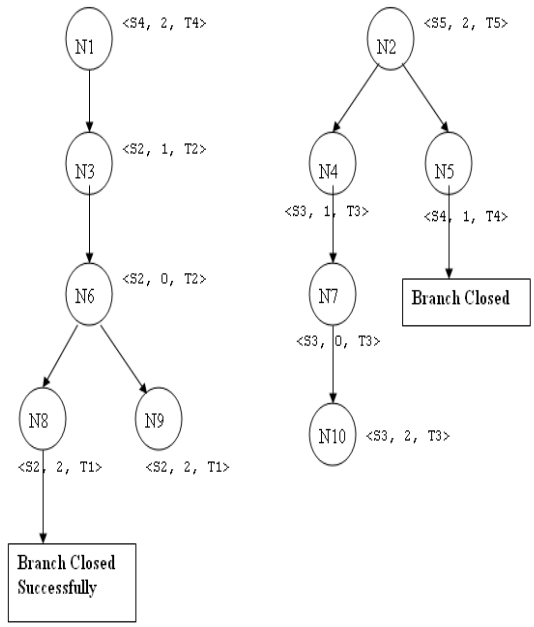

Figure 3: Path 1 successfully closed.

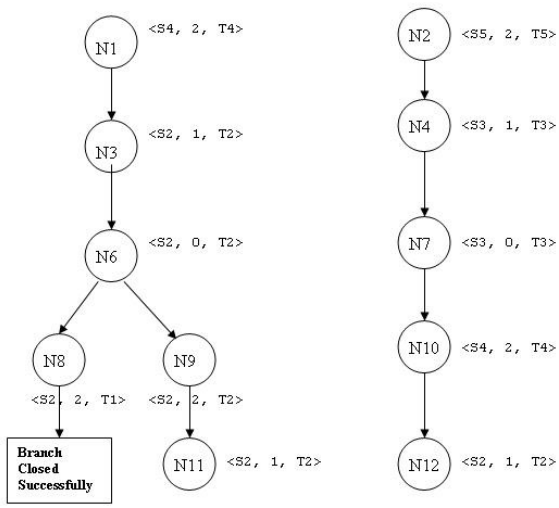

Figure 4: Creating Child Nodes N11 and N12.

Step 4: In this Backward Traversal is three computational paths are identified, Path 1, 2 and 3. There are also other paths but the Paths 1, 2 or 3 will be subsumed in those paths requirements.

Copyright © 2012 by the respective authors

ISBN 978-3-95450-124-3

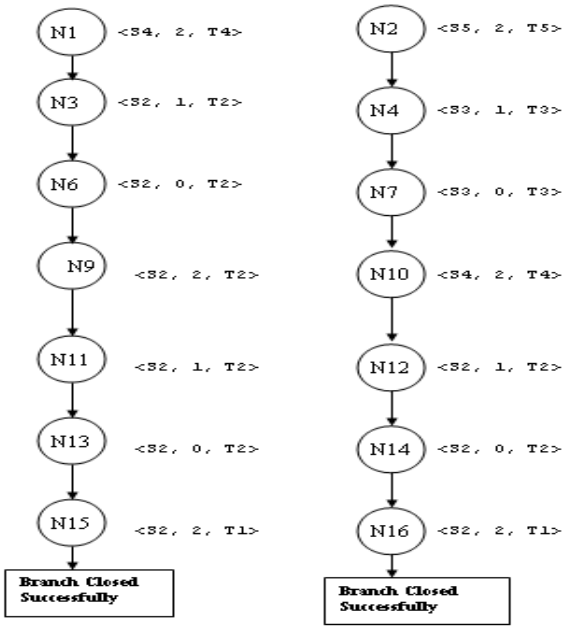

Figure 5: Path 2 & 3 are successfully closed.

CONCLUSION AND FUTURE SCOPE

In this paper we have implemented the modified algorithm to avoid the complexities of creating the composite states and composite transitions for each node of the tree branch. This program can verify the LEP protocol for any finite number of processes.

But it has some limitations. This method works only for LCP problem with no internal variables. In future, to make it work for any Distributed System Protocols we need to make this program more general. To take care of the internal variables, we can create a global static array (of size equal to the number of processes) that can store the internal variable values for each process, so that it will work for algorithms like Flood-Max algorithm, or HS Algorithm containing some internal variables for each process. In future work, we also need to modify the function comparing the Guard and Action conditions of the transitions to work it for different protocols.

REFERENCES

[1] Gerard Tel, "Introduction to Distributed Algorithms", ISBN-10: 0521794838, ISBN 13: 978-0521794831, October 2000.

[2] N. A. Lynch, "Distributed Algorithms", 1948, ISBN-13:978-1-55860-348-6 ISBN-10:1-55860-348-4.

[3] Tapas Samanta and D. Sarkar, "Distributed System Protocol Verification: A Tableau Based Model Checking Approach", Computer and Communication Technology (ICCCT), 2011, 2nd International Conference on: 246 – 252, 15-17 Sept. 2011, ISBN: 978-1-4577-1385-9.

Copyright © 2012 by the respective authors

ISBN 978-3-95450-124-3