Using Multi Decision Diagram in Model Checking

Nguyen Truong Khanh
School of Computing
National University of Singapore
Email: dcsntk@nus.edu.sg

Quan Thanh Tho
Faculty of Computer Science and Engineering
Hochiminh City University of Technology
Email: qttho@cse.hcmut.edu.vn

***Abstract*—Model checking[1] is an automatic verification technique for finite concurrent systems. In this method, the assertion is verified by exhaustively searching over the state space. However, the number of states of the system will grow exponentially with the number of processes. It limits model checker to handle with complex systems. In explicit model checking[1], system states are explored one-by-one and stored in memory explicitly, so the verified system is restricted by the memory resource. Most of the memory is consumed by the hash table which contains the visited states and the queue of states whose successors are already generated. In this paper, we will present a new way of storing the visited states by using a tree. We show that our approach is memory efficient. Organization of the report: Section 1 is the introduction, and section 2 introduces PAT model checker. Section 3 describes how to implement the tree storing the visited states. Section 4 presents the heuristic to improve the performance for the tree. Section 5 is the experiment result. Lastly section 6 is the conclusion and future work.**

***Keywords*-model checking; state space explosion; multi decision diagram;**

I. Introduction

Nowadays, the hardware and software systems are playing a vital role in our life. They appear in all our devices from the smartcards to the air-traffic controllers. Computers are increasingly used in safety-critical applications such as medical treatment and mission control. The presence of bugs in such applications is unacceptable. Model checking is an effective way to find bugs, especially the subtle ones. It uses the model of the system and properties to do the verification. The verification procedure is to search exhaustively all the possible states. On the progress of traversing states, each visited state is stored in a certain data structure, often in a hash table, to ensure that each state is explored at most once. This process continues until either there is a counterexample found or the whole state space is explored or the model checker runs out of memory. The problem when the memory is used up is called state space explosion. To handle with that problem, many techniques are provided to search the state space efficiently before running out of memory. In this paper, we present a new way to store each state into a tree called Multi Decision Diagram (MDD). This approach has been implemented in the PAT model checker.

In this approach, the data structure is implemented as a tree and each visited state is a branch in that tree. The tree will store the states and can verify whether or not a state has been visited before. The performance of the using tree is better in memory compared with storing directly states in the built-in hash-table Dictionary in .NET Framework 2.0.

II. PAT input language

PAT (Process Analysis Toolkit) is an enhanced simulator, model checker and refinement checker. Its homepage is at http://www.comp.nus.edu.sg/ pat/ [2].

PAT[3, 4] supports a wide range of modeling languages including CSP# [5] (short for communicating sequential programs), which shares similar design principle with integrated specification languages like TCOZ[6, 7]. The input language supports high-level compositional constructs like choice, parallel, interrupt, etc, as well as low-level programming language constructs like shared variables, arrays, if-then-else, etc. A process is defined by the following syntax:

P = StopA | Skip | e $\rightarrow$ P | P ; P | P [] P | P ? P | P <b> P | P ||| P | P || P where A is a set of events, e is an event, b is a Boolean expression. Process StopA never engages in any event from the set A. Process Skip means "terminate successfully". Action prefixing e $\rightarrow$ P is initially willing to engage the event e and behaves as P afterward. The sequential composition P1 ; P2 behaves as P1 until P1 terminates and then behaves as P2. One way to introduce diversity of behaviors is through choices. A choice between two processes is denoted as P1 [] P2. P1 ? P2 behaves as P1 until the first event of P2 is engaged, then P1 is interrupted and P2 takes control. Process P <b> Q behaves as P is b evaluates to true. Otherwise, it behaves as Q. Interleave process P1 ||| P2 includes 2 processes P1 and P2 running parallel without synchronization. Parallel composition of two processes is written as P1 || P2, where common events of P1 and P2 are synchronized.

Example: The following specifies the classic dining philosopher problem [8]

Phil(i) = get.i.(i+1)%N $\rightarrow$ get.i.i $\rightarrow$ eat.i $\rightarrow$ put.i.(i+1)%N $\rightarrow$ put.i.i $\rightarrow$ Phil(i);

Fork(x) = get.x.x $\rightarrow$ put.x.x $\rightarrow$ Fork(x) [] get.(x-1)%N.x $\rightarrow$ put.(x-1)%N.x $\rightarrow$ Fork(x);

College() = ||x:0..N-1@(Phil(x)||Fork(x));

where N is number of philosophers, get.i.j (put.i.j) is the action of the i-th philosopher picking up (putting down) the

j-th fork. In this specification, the i-th philosopher will first get the (i+1)-th fork, and i-th fork and then eat. After eating, he will put down the (i+1)-th fork, i-th fork respectively. The definition of Fork(x) is used to make sure that each fork at each time can be used by only one philosopher. The reason is because Phi(x) processes and Fork(x) are synchronized in the definition of College(). So when a fork is got by one philosopher, other philosophers can not use that fork.

III. Implementation of MDD

The memory problem which PAT faces is the same with other tools. Dictionary is currently used to store the visited states and it uses a lot of memory. The state space explosion happens in PAT with the parallel processes. In these processes, the number of states will grow exponentially with the number of sub processes. So our new approach will focus on the parallel processes to save memory. Suppose that we have a parallel process P = P1 || P2 || P3. One by one each process P1, P2, P3 in P changing creates a new state. For example, state P1||P2||P3 and state P1||P2||P3' are some states of P and only their last sub process is different. The first two sub processes are unchanged.

Tree will be used to store the visited states as following. The tree consists of one root node. This root node contains the link to other nodes. Each node includes the sub process's name and links to other below nodes. Each visited state is a branch of the tree. Checking whether a state was visited before is equivalent to check whether its corresponding branch belongs to the tree.

To make the algorithm clear, we consider an example. Suppose we have traversed 3 states (1, 2, 4), (1, 2, 3), (1, 2) respectively. The growth of the tree is as below

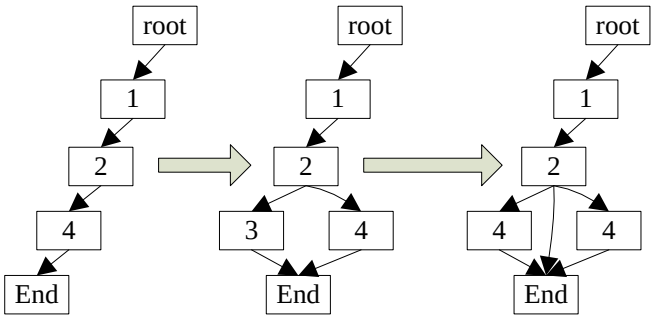

Figure 1. The growth of the tree when added 3 states (1, 2, 4), (1, 2, 3) and (1, 2)

Now suppose we want to check whether the state (1, 2, 4) is visited before. We will go across the tree and the branch (1, 2, 4) is found as no new node is created. The tree guarantees that checking whether a state is traversed for the first time or not is correct. The decision is based on if any new node is added to the tree while traversing the tree.

IV. Heuristic to order MDD

Suppose the parallel process P = P1 || P2 || P3. The domain of P1 is {1, 2, 3, 4}, the domain of P2 is {1, 2} and the domain of P3 is {1}. Then the tree for all states of the process P is as the Figure 2.

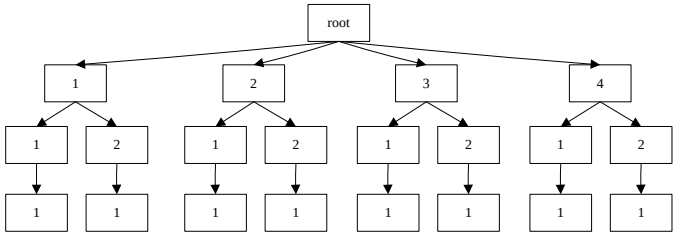

Figure 2. The state space of the process P = P1 || P2 || P3

The number of node is 21. However if we change the order P = P3 || P2 || P1 which does not change P, then the tree will be like the Figure 3.

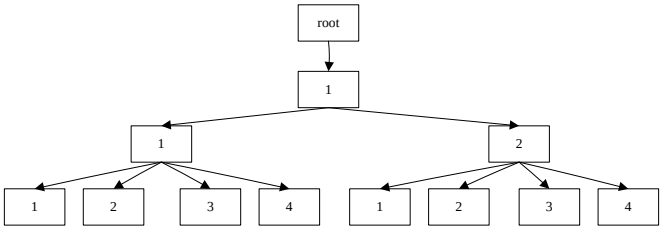

Figure 3. The state space of the process P = P3 || P2 || P1

The number of the node is now only 12 nodes. This example shows that the shape and size of the tree depends on the order of sub processes. We will apply the heuristic to determine the complexity of a process, in other word, to find the cardinal of the domain of each process. And then, the sequence of sub processes in the MDD will be changed in the increasing-complex order. In this order, the size of the tree is smaller.

The complexity of a process will be calculated by a function whose input is the function and output is a positive integer number. The function is defined by recursive as below:

f(Stop) = 1
f(Skip) = 2
f(e $\rightarrow$ P) = 1 + f(P)
f(P1 ; P2) = f(P1) + f(P2)
f(P1 [] P2) = max (f(P1), f(P2))
f(P1 <b> P2) = max(f(P1), f(P2))
f(P1 || P2) = f(P1) . f(P2)

where P, P1, P2 are processes, b is a Boolean expression. This function object is to calculate the number of states in

the graph describing the process although it varies at the runtime. We cannot know the value of the expression b and similarly, it is difficult to care the common events of P1 and P2 in P1 || P2. Therefore, it will take a lot of time to know exactly how many states for a process. In the current implementation of the tree, we only need the relative order of the complexity of the processes, and the exactly number is no need. So, the definition of the function is acceptable and it helps in most case.

V. Experiments

To know the performance of the MDD, we made some experiments and compared the memory used by the MDD with the Dictionary.

Milner	Number of states	Time (seconds)		Memory (bytes)	
		Dictionary	MDD	Dictionary	MDD
n = 9	65,796	5	15	8,003,964	4,691,892
n = 10	161,540	76	212	23,407,516	11,394,768
n = 11	389,124	49	144	56,671,420	27,206,508
n = 12	922,628	189	512	136,984,356	64,072,488

Table I
Comparison between Dictionary and MDD in time execution and memory using with Milner problem

From the Table I, the more states the problem has, the more memory the MDD saves. When the number of processes increases, the representation of each state is bigger. That leads the average memory for each state of Dictionary increases a lot; however, with the MDD, it increases a little. It saves about 50% memory as compared with the Dictionary. This rate still keeps stable when the size of the problem becomes bigger.

Leader Election	Number of states	Time (seconds)		Memory (bytes)	
		Dictionary	MDD	Dictionary	MDD
n = 5	2,587	6	10	1,225,458	544,712
n = 6	7,831	37	61	5,075,626	1,601,496
n = 7	22,058	197	327	20,728,314	4,432,772
n = 8	58,946	949	1,571	75,004,134	11,747,508

Table II
Comparison between Dictionary and MDD in time execution and memory using with Leader Election problem

The result in "Leader Election in Complete Graph" is even better than the previous. According the Table II, the MDD saves about 80% memory as compared with the Dictionary. Again, the rate still keeps stable with bigger problem. The reason for this better result depends on which problem or the added branches in the MDD. Moreover the MDD works more efficiently in Interleave process (the last experiment) than in Parallel process (the first experiment). In Parallel process, each element process will be synchronized together, so there are some states that never happen, but can happen in the Interleave process. For example, we have process P1 whose domain is 1, 2 and process P2 whose domain is 3, 4. If P1 is synchronized with P2, sometimes when P1 changes from 1 to 2, P2 will change from 3 to 4 simultaneously. So with P = P1 || P2, process P's states will never be (2, 3). On the other hand, when P = P1 ||| P2, process P's state can be (2, 3). Therefore, in the Interleave process, the MDD will have a lot of nodes to share. It's why the MDD saves a lot in the "Leader Election in Complete Graph".

VI. Conclusion and future work

In summary, there were many studies about storing the visited states while traversing in model checking. Most of them concentrated on how to hash the state representation into a lower-cost representation such as: bit-state hashing and hash compaction [9]. On the other hand, in this paper, the approach works as a later stage which concentrating on how to store the state representation effectively. This approach can be applied with the former approaches to have a better performance. When each state is hashed as a sequence of bits 0 and 1, the tree will become Binary Decision Diagram (BDD) [10] which is more saving and faster.

References

[1] Clarke E M, Grumberg O, Peled D A, *Model Checking*. The Massachusetts Institute of Technology(MIT) Press, 2000.

[2] Sun J, Liu Y, Dong J S. PAT, *PAT: process analysis toolkit*. http://www.comp.nus.edu.sg/ pat/

[3] J. Sun, Y. Liu, J. S. Dong and J. Pang, *PAT: Towards Flexible Verification under Fairness*. 21th International Conference on Computer Aided Verification (CAV 2009), Grenoble, France, June 2009.

[4] J. Sun, Y. Liu, J. S. Dong and H. Wang, *Specifying and Verifying Event-based Fairness Enhanced Systems*. 10th International Conference on Formal Engineering Methods (ICFEM 2008). Japan, Oct 2008.

[5] Hoare C A R, *Communicating sequential processes*. International Series in Computer Science, Prentice-Hall, 1985.

[6] B. Mahony and J.S. Dong, *Timed Communicating Object Z*. IEEE Transactions on Software Engineering, 26(2):150-177, Feb 2000.

[7] B. Mahony and J.S. Dong, *Blending Object-Z and Timed CSP: An introduction to TCOZ*. In Proceedings of the 20th International Conference on Software Engineering (ICSE'98), IEEE Press, pages 95-104, Kyoto, Japan, 1998.

[8] D. Lehmann and M. Rabin, *On the Advantage of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem (Extended Abstract)*. In Proc. 8th Annual ACM Symposium on Principles of Programming Languages (POPL'81), pages 133–138. 1981.

[9] Gerard J. Holzmann, *An Analysis of Bitstate Hashing*. Formal Methods in System Design archive.

[10] Burch J R, Clarke E M, McMillan K L, et al, *Symbolic model checking: 10^{20} states and beyond*. Information and Computation, 1992, 98(2): 142170.

Listing 1. Implementation of MDD

```
public struct Node
  public int value;
  public Tail tail;

public bool Visit(List<int> value)
  Node temp = root;
  bool result = false;
  VisitTemp(value, ref temp, ref result);
  if (result || (!(temp.tail[0].value == int.MinValue && temp.tail[0].tail == null)))
    count++;
    Node leafNode = new Node();
    leafNode.value = int.MinValue;
    leafNode.tail = null;
    temp.tail.Insert(0, leafNode);
    return true;
  else
    return false;

private void VisitTemp(List<int> value, ref Node temp, ref bool result)
  for (int i = 0; i < value.Count; i++)
    if (temp.tail.Count == 0)
      result = true;
      Node childNode = new Node();
      childNode.value = value[i];
      childNode.tail = new Tail();
      temp.tail.Add(childNode);
      temp = temp.tail[0];
    else
      int pos = BinarySearch(temp.tail, value[i]);
      if (pos >= 0)
        temp = temp.tail[pos];
      else
        result = true;
        Node childNode = new Node();
        childNode.value = value[i];
        childNode.tail = new Tail();

        temp.tail.Insert(~pos, childNode);
        temp = temp.tail[~pos];
```