Implementation of Symbolic State Space Generator using Reduced Ordered Binary Decision Diagram based on SDES Description in PDETOOL Framework

Reza Fathi[†] **and Mohammad Abdollahi Azgomi** [††],

School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran

Abstract
The main disadvantage of Model Checking is the state-explosion problem, which can occur if the system under verification has many processes or complex data structures. Although the state-explosion problem is inevitable in worst case, over the past 2 decades considerable progress has been made on the problem for certain classes of state-transition systems that occur often in practice. Using of Decision Diagrams to hold and manipulate the state space is an approach for this problem. In this approach the state space of the model is preserved in symbolic way instead of set.
An algorithm is proposed in this paper for symbolic state space generation on SDES description models. Using SDES which is a multi-formalism description makes it possible to transform other stochastic discrete event system formalisms like Petri nets, stochastic activity networks, etc. to SDES formalism and then generate symbolic state space for them. Using symbolic state space generation by reduced ordered decision diagrams makes it possible to produce larger state spaces; which is used to producing state space of models in PDETool in order to alleviate its state space explosion problems.

Key words:
Symbolic state space generation; reduced ordered binary decision diagrams, state space explosion; SDES description; stochastic discrete event systems.

1. Introduction

As systems become more complex, the need for software tools to model and analyze these systems grows. Generating a high-level abstracted model of the system such as Stochastic Petri Nets (SPNs), stochastic reward nets (SRNs), and stochastic activity networks (SANs) and then analyzing the model to get some qualitative and quantitative properties of the system is a common way to analyze it. This model analyzing and checking can be assessed by analyzing low-level state space and state graph of the model. Since the size of the state space grows exponentially with the number of the model's variables, model checking techniques based on explicit state space production methods can only handle relatively small examples. Even relatively simple models can suffer from the commonly called state space explosion problem [1], where the number of states reachable from the initial state becomes too large to store.

In order to cope with increasingly complex models we therefore require advanced techniques for constructing and storing state spaces and state graphs. Generating symbolic state space of model using Decision Diagrams is a way to alleviate the state space explosion problem in tools. PDETool was using a traditional way, linked lists, to construct and preserve state space of the models. So, it was not able to do model checking on complex models. In order to makes it powerful encountering complex models for model checking, in this paper, we introduce a new method for producing symbolic state space of a model defined by SDES description by adding a symbolic state space generator module to the tool. This module generates symbolic state space of the model using ROBDD data structure and then delivers it to a symbolic model checker module to do symbolic model checking on it.
The remainder of this paper is organized as follows. In section 2, ROBDD and SDES description are briefly described. Section 3 reviews related works in this field and describes PDETool briefly. Section 4 describes the motivations of the paper. In section 5, state space generation algorithm including ROBDD algorithm is brought besides an example of ROBDD is described in section 6. After that, in section 7, an evaluation of the implementation is brought. And finally, some concluding remarks and a list of future works are mentioned in section 8.

2. Background

2.1 Reduced Ordered Binary Decision Diagram (ROBDDs)

Binary decision diagrams (BDDs) as a data structure for representation of Boolean functions were first introduced by Lee [2] and further popularized by Akers [3] and Moret [4]. They are rooted, directed, acyclic graphs. A BDD is

Manuscript received July 5, 2013
Manuscript revised July 20, 2013

constructed over a finite, ordered set of Boolean variables that represents a Boolean function. We represent it as fB: $\mathbb{B}\ K \rightarrow \mathbb{B}$ over k boolean variables $x_k > \cdots > x_1$ [5].

Definition: Let B be a $\wp$-OBDD. B is called reduced if for every pair (v,w) of nodes in B: $v \neq w$ implies $fv \neq fw$. Let $\wp$-ROBDD denote a reduced $\wp$-OBDD [6].

In the restricted form of Reduced Ordered BDDs (ROBDDs) they gained widespread application because ROBDDs are a canonical representation and allow efficient manipulations as proved by Bryant [7]. In an ROBDD, every reduced cofactor of a function is shown by exactly one node. This is a key precondition to prove that a ROBDD data structure is universal and canonical. Simply, universal means that every function can be shown by a OBDD. And, canonical means that any OBDD for the same function will be alike if we rename the nodes. On the other hand, for a function always there is a ROBDD [5], [8].

2.2 SDES description

A discrete-event system is a system that is in a state during some time interval, after which an atomic event might happen that changes the state of the system immediately. Several stochastic discrete-event models have been proposed, which all share some common characteristics and many algorithms and methods that have been developed for one model are applicable for many of them. SDES [9], introduced by Zimmermann, is a unified description for stochastic discrete-event systems. Popular model classes like automata, queuing networks, and Petri nets of different kinds with stochastic extensions are subclasses of stochastic discrete-event systems and can be translated into the SDES description.

In [9], a stochastic discrete-event system, SDES, is defined as a tuple SDES = (SV^*, A^*, S^*, RV^*), where SV^* describes a finite set of state variables and actions A^* together with the sort function S^* and the reward variables RV^* corresponds to the quantitative evaluation of the model. With allocating values to the state variables, all the possible states of the model, on the other hand, state space of the given model defined in $\Sigma = \prod_{(sv \in SV^*)} S^*(sv)$ is produced.

Each state variable is defined by a couple sv=($Cond^*$, Val_0^*) where Val_0^* is a function representing the initial value of each estate variable and $Cond^*$ indicates whether or not a state variable is allowed in a specific state of the model. An action $a \in A^*$ of SDES describes possible state changes of the modeled system. It is composed of the attribute functions defined as a=(Pri^*,Deg^*,$Vars^*$,Ena^*,$Delay^*$,$Weight^*$,$Exec^*$).

And each item is defined as follows:

- Pri^* associates a global priority to every action.
- The enabling degree Deg^* of an action specifies the number of activities that are permitted to run concurrently in any state.
- The action variables $Vars^*$ define a model-dependent set of variables $Vars^*(a)$ of an action a with individual sorts.
- The value of the Boolean enabling function Ena^* of an action variant for a state returns if it is enabled or not.
- $Delay^*$ describes the time that must elapse while an action variant is enabled in an activity until it finishes.
- The $Weight^*$ of an action variant is a real number that defines the probability to select it for execution in relation to other weights.
- $Exec^*$ defines the state changes that happens as a result of an action variant execution and is called execution function.

3. Related works

3.1 BuDDy and CUDD pachages

Two well-known packages that are used widely to create and manipulate decision diagrams are BuDDy developed in IT University of Copenhagen and Cudd which is developed in the University of Colorado. BuDDy is a powerful library for Boolean expression manipulation; it is implemented in C but has a wrapping C++ interface. BuDDy combines as easily as a C++ interface and is an efficient implementation based on the novel BDD data structure. A BDD represents a formula as decision graph where the nodes in the graph are vertices and the edges coming out of a vertex represent the two possible Boolean assignments to that variable. Thus, a complete assignment to all variables corresponds to a path in the graph which ends in a value of true or false, which is the value of the formula when given that assignment. Each node requires 20 bytes of memory in the implementation of the BuDDy [10].

The CUDD package provides functions to manipulate Binary Decision Diagrams (BDDs), Algebraic Decision Diagrams (ADDs), and Zero-suppressed Binary Decision Diagrams (ZDDs). BDDs are used to represent switching functions; ADDs are used to represent function from $\{0,1\}^n$ to an arbitrary set. ZDDs represent switching functions like BDDs; however, they are much more efficient than BDDs when the functions to be represented are characteristic functions of cube sets, or in general, when the ON-set of the function to be represented is very sparse. They are inferior to BDDs in other cases. The package provides a large set of operations on BDDs,

ADDs, and ZDDs, functions to convert BDDs into ADDs or ZDDs and vice versa, and a large assortment of variable reordering methods.
A C++ interface is included in the distribution of CUDD. It automatically frees decision diagrams that are no longer used by the application and overloads operators. Almost all the functionality provided by the CUDD exported functions is available through the C++ interface, which is especially recommended for fast prototyping [11].

3.2 The PDETOOL framework

There exist many modeling and simulation tools, which most of them support only a single simulation or modeling language and a few simulation or solution methods. It is interested to develop a multi-formalism modeling framework to support a wide range of models and easily be extensible to support new formalisms. PDETool is a framework that developed for verification of the discrete-event systems. It uses SDES description as a middle language to translate all the input models into it, and then, do all the wanted operation on the unified SDES model. This means that any verification we provide on the SDES model can be done on any model that is translated to this language. The core engine of the tool is SimGine which has been developed to work on SDES models. Every input models like stochastic petri nets, stochastic active nets (SANs), and etc., given to the tool using its interface is translated to the SDES model. The translation is done by Model Translator module as shown in the Figure 1. Then the input model in the SDES language is given to the SimGine engine to do the simulation on the model.

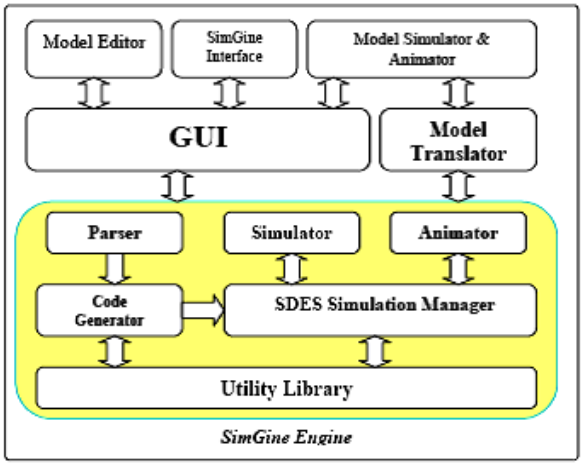

Figure 1. Architecture of the PDETool [12]

When the translated model from input is available, it is given to the state space generator (SSG) to generate reduced state space of the model. SSG and model checker modules are shown in PDETool architecture in Figure 2. State space generation step is a prerequisite step for model checking. Traditional data structures like linked lists, bit or hashed map, and etc. to produce and maintain it is not efficient. The problem is when a model becomes larger, the state space of the model becomes enormously large, and then, the state space explosion occurs. This state space explosion problem was a big obstacle for PDETool in order to do model checking on complicated models because it was using linked lists method for storing state space of the input models. To alleviate this problem, a state space generator module was added to the tools to produce reduced ordered decision diagram to preserve state space of the input model.

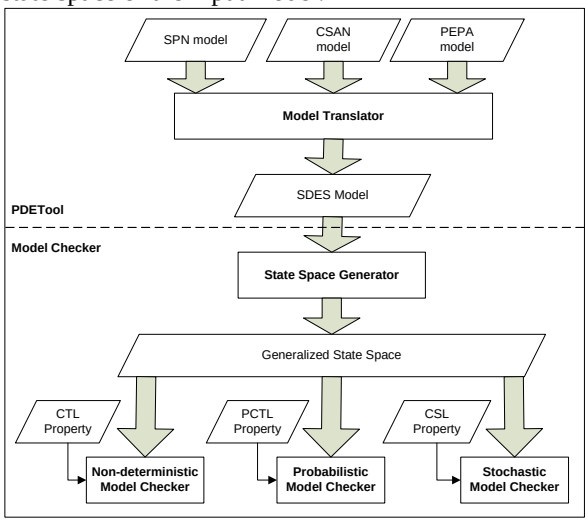

Figure 2 Architecture of state space generator and model checker in PDETool

4. Motivations

PDETool is a multi-formalism modeling and simulation tool for stochastic discrete-event systems which uses SimGine, a simulation engine based on a unified abstract description named SDES. After getting SDES description of the input model in the tools, we have to do some model checking on them. But, with growing the input model, and hence, growing state space of the model, the PDETool was encountering state space explosion problems; because it was using traditional method of linked list for saving state space of the model. In order to alleviate the problem of the tools in this field, we tried to preserve state space of the model in the symbolic form using reduce ordered binary decision diagrams.
Since PDETool has been developed using .net framework with C# programming language, either we had to use tools like Cygwin to use BUDDY or CUDDD libraries which

caused speed inefficiency or develop ROBDD in the tool. By implementing ROBDD in PDETool with C# programming language, we could implement other forms of symbolic state space generating methods and then do model checking on them all integrated in the tool. So, we chose to implement a naval development of ROBDD in our tool which is described in the following sections.

5. State Space Generation Algorithm

For producing state space of an SDES model, we should fire all events of a state and then add new generated states to the state space. Two data structures as shown in Figure 3 are used for state space generation. One is used Utable that used to store main BDD and the other one is Htable which is used to store implemented functions. When we are looking for a function in Utable, if the function is not in it, we look for its remaining part of the function in the Htable. If the function have been implemented before, we can find its index in Utable find in Htable and use it to complete the new function implementation, or else, we implement the needed function in Utable and index it in Htable in reverse mode so that we can find it next time if necessary and use it again. This Htable helps us to find implemented functions in Utable and avoid redundant function implementation.

Figure 4 shows the general algorithm. Firing all enabled events in a state, makes the state space generation (SSG) algorithm a BFS search algorithm. After firing an event, we add the new generated state to the state space of model and also to the newGens Diagram to preserve new generated states from current states in out. At the end of inner loop, we move existing states in newGens to the out and continue main loop. If there was no new generated states in out and it was empty, the algorithm will end.

```
NODE{
    int var;
    long low;
    long high;
    long parent;
    long refs;
}
```

(A) Utable node staructure

```
HNODE{
    int var;
    long low;
    long high;
    long donotcare;
    long ulow;
    long uhigh;
    long refs;
}
```

(B) Htable node structure

Figure 3 Utable and Htable data structure

```
Algorithm #1: state space generation
SSG(SDES model) returns ROBDD including state space of model
begin
ROBDD ss,out,newGens;
Initialize ss,out,newGens; //initialize the OROBDD Structure
Ss=initial states;
Out=initial states;
newGens=null;
While out isn't empty do
Begin
        For each state s in out do
        Begin
                For each action a in model do
                Begin
                    If a is enabled in s then
                  Begin
                        NewState=s.execute(a);
                        newGens.findOrAdd(newState);
                        ss.findOrAdd(newState);
                  End if
                End for
        End for
        Out=newGens; newGens=null;
End while
Return ss;
End algorithm #1;
```

Figure 4 general state space generation algorithm

Function findOrAdd showed in Figure 5, represent the algorithm of add a given state to a ROBDD. It starts from row 2 of the table. This point is the entry point of function containing of state space of the model. Going forward for searching in Utable, if state variable is zero, it goes column low of the table and if it is one, it goes forward high column index of the Utable.

It goes forward until reaches to row 1 representing that state belong to the state space of the model. In this case it returns true resembling that state belonged to the state space. If it reaches to a row containing zero in its related column in low or high depending on the value of state variable, it starts to add the this state as a new state to the ROBDD by calling addhuTable function shown in Figure 6. After adding new state to the state space of the model, it returns false, representing that this was a new state and added to the space of the model.

```
Algorithm #2: findOrAdd a State to ROBDD
findOrAdd(state s) returns Boolean;
Begin
Int varnumber=s.length;
```

```
Booleab find=true;
Index=2;
While varnumer<model.varnumbers+1 AND index>1 do
Begin
        If s[varnumber-1]==0 then direction=low else direction=high end if
        If uTable[index].direction>0 then
                Begin
                    Index=uTable[index].direction;
                    Varnumber=uTable[index].var;
                End
        Else
                Begin
                    find=false;
                    addhuTable(s,varnumber,index);
                   break;
                End if
End while
Return find;
End function
```

Figure 5 algorithm of find or add a state to a ROBDD

When we are searching a state in ROBDD at Utable, if we reach a row containing zero in its low or high column depending on the state variable value, we find this state is not in ROBDD and should create a new path in Utable showing its existence. So we call addhuTable function to build reverse route of semi state in Htable that is a BDD and at constructing reverse route in Htable, we make the rout at Utable too. At the end we connect the created route in Utable to the zero branch of row where we had to call this function.

Htable is a table that is used to save reverse of the state to use in order to prevent finding same functions that exist in ROBDD and prevent building redundant routes. It is called semi because we only add the part of state to Htable that there is not in Utable. And it has link to the related row of the Utable to be used when we need to create new route in Utable that in its route to the leaf, it countered to zero and the remaining part of state shows function that Htable has implemented it before and gives us its index in the Utable. Implementing Htable as a semi BDD makes it faster in finding functions that is needed to adding new states.

fnCopy function is called when it is needed to add new route to a node and its reference is greater than one. It means that more than one function is using this route and making any change to this row will affect other functions. For preventing this, we have to copy existing route for this function form a point where it has the last reference of one in its route form root to the point it is needed to add new route.

Shrink function is called when a node has the same low and high index branch. It means this state variable is a Don't care for this route and can be shrunk and moved from this route. It is done by linking the indexing row to this row to the branch of this row and then frees this row. After shrinking a route, its affect be applied to the Htable to show the created don't care.

```
Algorithm #3: add new state to Htable and Utable to make reverse table.
addhuTable(state s,int varnumber,long index)
begin
  hindex=2;
  var=1;
  while  var<varnumber do
  begin
    if state[var-1]==0 then
       direction=low;
    else
       direction=high;

    if htable[hindex].direction==0 then
       createNewuhNodes(hindex,state,var);

    hindex=htable[hindex].direction;
    var=htable[hindex].var;
  end while
  if utable[index].refs>1 then
    index= fnCopy(state,index);

  if state[varnumber-1]==0 then
    utable[index].direction=htable[hindex].ulow;
  else
    utable[index].direction=htable[hindex].uhigh;

  if utable[uloc].low==utable[uloc].high!=0 AND uloc>2 then
    shrink(uloc,hindex,state);
 end if
End function
```

Figure 6 algorithm addhuTable

The algorithm in Figure 4 is a breadth first search (BFS) one that fires all enabled actions in every state and then will add any generated states in the state space of the model to the ROBDD only one time. The complexity of adding one state to the SS is $O(\Sigma_{(sv \epsilon SV^{\wedge *})}[\log_2(count(S^*$

(sv)))]) which the height of the decision diagram is. Because every states is added to the state space one time, then the algorithm's order is liner regard to the state space of the model, and hence, is added to the N_{ss}; and because every event is considered for every state added to the SS, then the total multiplied to the number of events defined in the SDES model of the input model (N_{events}). In the end, the order of generating state space of a SDES model is the below formula.

$$O(N_{ss} \times (N_{events} \times \Sigma_{(sv \epsilon SV^*)}[\log_2(count(S^*(sv)))]))$$

6. Example

In order to clarify the algorithm, an example is bringing in this section. This example contains five states that is shown in the form of "$X_4X_3X_2X_1$" which contains four variable. For variable orders, the highest number variable is the first and stays at the highest level of the diagram and so all. The states are "0000", "1111", "1101", "1100" and "1110". Table 1 shows the resulting Utable that impalement ROBDD that is used to keep states in the form of symbolic.

Table 1 resulting Utable of the example

index	var	low	high	parent	refs
0	5	0	0	0	0
1	5	1	1	0	0
2	4	5	8	0	0
3	1	1	0	4	1
4	2	3	0	5	1
5	3	4	0	2	1
6	0	0	0	9	0
7	0	0	0	6	0
8	3	0	1	2	1
9	0	0	0	0	0
10	0	0	0	7	0

Table 2 shows Htable that is a reverse table for preserving reverse of the states in the form of BDD. Entry point to both of them is index 2. Column index in both are brought for better understanding and is unnecessary. Column var contains the variable number of the state variable. Row zero and one's var number is variable numbers plus one to distinguish them from ordinary rows and shows that they are reserved rows for false and true. The low column is used for False variable value and high for the True variable value. Parent column in Utable is used to point the father of this node. Column refs is used to maintain number of references to this node and there for to this function. It is used to prevent changing this function if more than one function is using this and instead copy it before changing it. In Htable column, Don'tcare column is used to maintain dont care courses. Column ulow is used to preserve entry point of this function in Utable for low route from route until this row and column uhigh is used for the same purpose for high route.

Table 2 Htable of the example

index	var	low	high	dontc	ulow	uhig
0	5	0	0	0	0	0
1	5	1	1	0	0	0
2	1	3	0	6	3	0
3	2	4	0	0	4	0
4	3	5	0	0	5	0
5	4	0	0	0	0	0
6	2	0	0	7	0	0
7	3	0	8	0	0	8
8	4	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0

7. Evaluation and experimental results

Type St For observing algorithm's behavior, the state-time diagram is shown in Figure 7. All states were produced randomly. The horizontal axis shows the number of states added to the ROBDD and the vertical axis shows the time it takes to do. To find out the time needed to add a specific random state, the algorithm is executed 500 times and the times in the diagram is average time of the 500 execution. The variance between different execution times for a point were always lower than 0.01. For example, in order to obtain the time for adding 1000 random states to the implemented ROBDD, 1000 states produced randomly and then added to the structure, then this process iterated 500 times. And finally, the result time was calculated the average time of all this 500 execution times. The data is generated by executing the process on a regular notebook computer with a core i5-480M 2.66 processor and 4G ram under the Windows 7 operating system.

The symbolic approach is attractive because it allows decision diagram nodes to share not only state encodings but also intermediate results, during symbolic state-space generation. The more state encoding and intermediate results are shared, the greater efficiency symbolic approaches exhibit with respect to explicit ones and this is shown in Figure 7. It is shown in the diagram that when the states are going to be added are few, the needed time to build the state space is higher per state. This is because the time to initialize the data structure considerable when the states are few. But the high number of states shows that the needed time per states becomes little. It can be seen from the diagram that when the number of states go higher, the times increases with lower steep; and hence, the algorithm is more effective when it is used for complicated models which have huge state spaces.

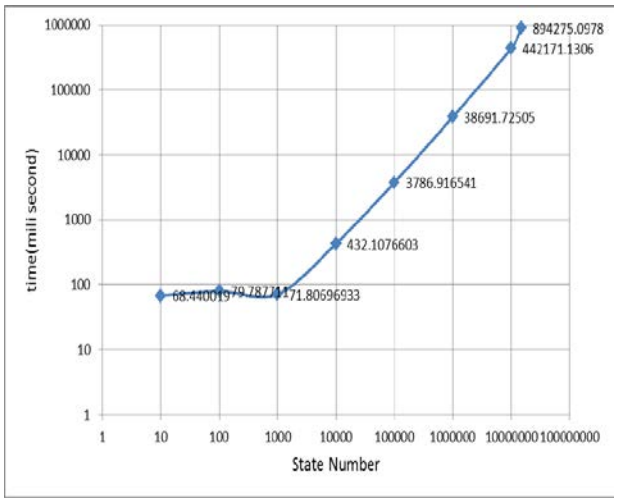

Figure 7 Stat-Time diagram for adding random states to 24 variable ROBDD

In order to evaluate our implementation and compare its efficiency with existing and implemented packages, we chose BuDDy one of the popular packages for this purpose. So, we compare our implementation and BuDDy 2.2 performance on solving N-Queen problem. The same hardware as mentioned above used to execute both of them. The result of the execution is shown in the Figure 8. The horizontal axis shows the queen number of the problem and the vertical axis shows the time need to solve the problem which is base 10 logarithmic scale. Although the half-logarithmic diagram shows that BuDDy solves the problem faster, our solution 'speed has same slope and tend to reach the BuDDy's speed in complicated problems with very large state spaces. This was our first attempt to implement ROBDD for state space generator of the PDETool framework and we hope we enhance it at our next attempts.

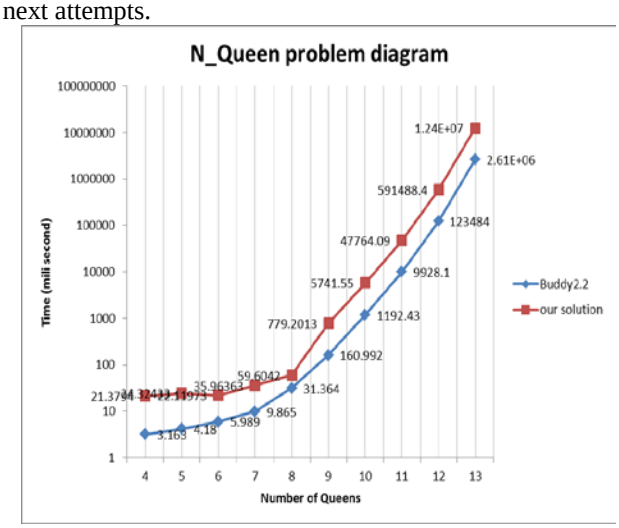

Figure 8 Solving -Queen Problem Diagram in PDEToll and BuDDy 2.2

8. Summery and future works

This paper presents a naval implementation of symbolic state space generation using ROBDD data structure. Implementing ROBDD data structure makes the PDETool, a multi-formalism framework, able to produce symbolic state space of the input models. Since PDETool uses SEDS as a uniform language and translates all input models to this language, it can produce symbolic state space to the all input models like GSPN, SAN, and etc. and then do symbolic model checking on them. In brief, now PDETool is a tool that can model and analyze more complex systems with very large state spaces.

Future work will include implementation of other forms of decision diagrams like MDDs and ZDDs and other forms of the decision diagrams besides implementing the functions for transforming between them. Also, working on producing distributed state space of the model using ROBDD and doing distributed model checking on it is a good field to continue to work on it. In addition we are going to work on variable reordering algorithms and try to make the ROBDD's size smaller by selecting better variable orders.

References

[1] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the state explosion problem in model checking," in Springer, 2001, pp. 176--194.

[2] C. Y. Lee, "Representation of switching circuits by binary decision dia grams," Bell Syst.Tech. J., vol. 38, p. 985–999, 1959.

[3] S. Akers., "Binary Decision Diagrams," in IEEE Trans Comp, vol. 27, 1978, pp. 509-516.

[4] B. M. E. Moret, "Decision trees and diagrams," ACM Computing Surveys (CSUR), vol. 14, no. 4, pp. 593--623, 1982.

[5] D. Parker and A. Miner, "Symbolic representations and analysis of large state spaces," Citeseer, pp. 296-338, 2004.

[6] C. Baier and J. P. Katoen, Principles of Model Checking(Representation and Mind Series). TheMIT Press, 2008.

[7] R. E. Bryant, "Graph-Based algorithms for Boolean function," IEEE Trans. Computers, vol. 35, no. 8, 1986.

[8] G. Ciardo, G. Lttgen, and A. S. Miner, "Exploiting interleaving semantics in symbolic state-space generation," Formal Methods in System Design, vol. 31, no. 1, pp. 63-100, 2007.

[9] A. Zimmermann, Stochastic discrete event systems: Modeling, evaluation, applications. Springer-Verlag New York Inc, 2008.

[10] J. Lind-Nielsen. BuDDy - A Binary Decision Diagram Package. [Online]. http://sourceforge.net/projects/BuDDy

[11] [Online]. ttp://vlsi.colorado.edu/~fabio/CUDD/node1.html

[12] A. Khalili, A. Jalaly Bidgoly, and M. Abdollahi Azgomi, "PDETool: A Multi-formalism Modeling Tool for Discrete-Event Systems Based on SDES Description," Lecture Notes in Computer Science, vol. 5606, pp. 343-352, Jun. 2009.

[13] A. Jalaly Bidgoly, A. Khalili, and M. Abdollahi Azgomi, "Implementation of Coloured Stochastic Activity Networks within the PDETool Framework," in Proc. of 3rd Asia Int'l Conf. on Modelling & Simulation (AMS09), 2009, pp. 710-715.

[14] PDETool-Homepage. [Online]. http://pdel.iust.ac.ir/pdetool/

[15] HomePage. [Online]. http://pdel.iust.ac.ir/Projects/SimGine.html

[16] S. Akers, "Binary decision diagrams," IEEE TRANSACTIONS ON COMPUTERS, 1978.

[17] R. Bryant, "Graph-based algorithms for boolean function manipulation," IEEE Transactions on Computers, vol. 35, no. 8, pp. 677-691, 1986.

[18] C. Lee, "Representation of switching circuits by binary-decision programs," vol. 38, 1959.

[19] D. a. M. A. Parker, "Symbolic representations and analysis of large state spaces," in Validation of Stochastic Systems, 2004, pp. 296-338.

[20] L. Ghomri and H. Alla, "Modeling and analysis using hybrid Petri nets," Nonlinear Analysis: Hybrid Systems, vol. 1, no. 2, pp. 141-153, 2007.

[21] D. Lime and O. H. Roux, "Model checking of time Petri nets using the state class timed automaton," Discrete Event Dynamic Systems, vol. 19, no. 2, pp. 179--205, 2006.

Reza Fathi received the B.S. degree from Birjand University in 2007 and M.S. degree from Iran University of Science and Tachnolgy in 2011 both in Computer Engineering . He has been a researcher member of Performance and Dependability Engineering (PDE) Research Lab since 2008. In addition, he works as developer and manager in Raydana softwar company which develpes Enterprise Resource Planing (ERP) systems based on J2EE Technology.

Mohammad Abdollahi Azgomi received the B.S. , M.S. ,and Phd. degree in Computer Engineering from Sharif University of Technology in years respectively 1991, 1996, and 2005. He has been an Assitant professor in Iran University of Science and Technology at Computer Engineerig Departmant since 2005. In addition he has been the Director of Information Technology Group, E-Learning Center in IUST since 2006.