Electronics and Computer Science

Faculty of Engineering and Physical Sciences

University of Southampton

Zhou Jie

September 2020

Improving the model checking function of PRISM model checker by using symbolic approach

Project supervisor:

Dr Corina Cirstea

Second examiner:

Dr Enrico Marchioni

A project report submitted for the award of

MSc Artificial Intelligence

Abstract

We will first introduce the cause of this project, which is the problem of state space explosion during model checking. The following section II is the background research about this project. In this section, we will briefly go through the related basic concepts such as model, property, and model checking. The algorithm for symbolic RBM model checking will be explained in detail. And the PRISM model checker tool will be introduced briefly. In section III, the problem to be solved in this project will be defined concretely. Section IV and V are the implementation and testing. The implementation includes 2 parts: model building and model checking. The conduction of unit and functional tests are in section V. The remaining sections are the Analysis and Conclusions.

Acknowledgements

I would like to thank my supervisor, Dr Corina Cirstea, for giving me support through this project. It is her patient guidance that makes me overcome difficulties and make progress.

Statement of originality

I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.

I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.

I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

• I have acknowledged all sources, and identified any content taken from elsewhere.

• I have not used any resources produced by anyone else, with the exception of the PRISM model checker for which I have designed and developed an implementation, as well as several open-source libraries, the details of which can be found in the project archive.

• I did all the work myself, or with my allocated group, and have not helped anyone else.

• The material in the report is genuine, and I have included all my data/code/designs.

• I have not submitted any part of this work for another assessment.

• My work did not involve human participants, their cells or data, or animals.

Contents

I. INTRODUCTION

There are many complex computerized systems in our daily life. Some of them have high requirements for security. The correctness of them is becoming more and more important. The conventional methods are often inadequate to test those systems. Formal verification is a technology that aims to resolve this problem. In this project, it is done by the approach of model checking.

In the process of model checking or synthesis. The number of states often increases exponentially. If you have 2 Boolean variables there are 4 states. If you have n Boolean variables there are 2^n states and so on. This phenomenon is called the State space explosion. We need an efficient and succinct way to represent the state space. Binary Decision Diagram (BDD) is a solution to it. It reduces the number of states by removing duplicate and redundant states by a series of algorithms and operations between BDDs. This project is to make a further extension of the symbolic representation function of PRISM to improve the scalability of the process.

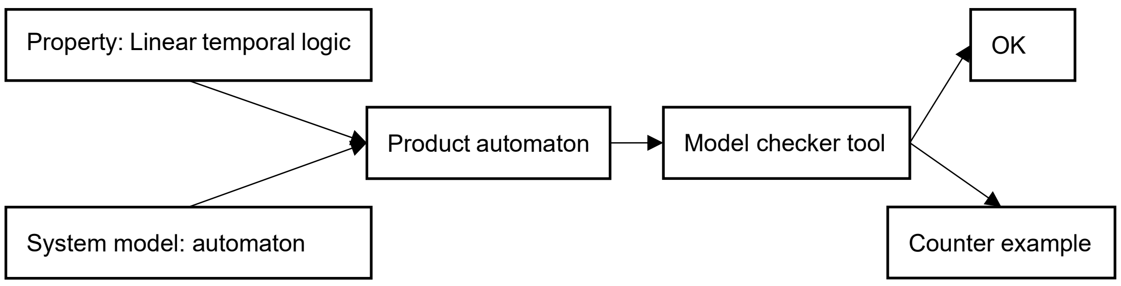
The goal of this project is to deliver an extension program of PRISM model checker. The extension program should reduce the complexity when traversing the state space during the process of the model checker. This should be demonstrated by testing different scenarios after implementation.

II. BACKGROUND RESEARCH

Safety and reliability are very important for some systems, like traffic lights controllers and pacemakers implanted in people’s hearts. Conventional testing sometimes can’t meet their safety requirements. A new verification technology called model checking was developed to solve this problem.

In general, model checking performs the verification by modelling the behavior of the system into an automaton i.e., the abstraction of the model, and verify it on a specification or property which specify the requirement expressed in a formal language called linear temporal logic (LTL). There are two results from model checking: the model satisfies the property or the model doesn’t satisfy the property and a counter example will be given. There are many kinds of model checking tools. In this project, we will use the PRISM model checker [1].

Below is the workflow of the model checking described above:



The questions that follow are: what is a model in an automaton? what is the property expressed in linear temporal logic? How to perform the model checking? All these questions will be explained in the following sections.

A. What is model?

Automaton is the abstraction of system model. The automaton mentioned here refers to a finite state machine (FSM) or finite state automaton (FSA) [2]. An FSM with labels is known as a Kripke structure [3]. The Kripke structure is defined as a 4-tuple (S, S0, T, L), where S is a finite set of states, S0 is the initial states, T is the transitions between states, and L labels each state. Specifically, L are the labels of states expressed as a set of atomic propositions (AP) and every state is represented as an element of a powerset of atomic propositions. In this project, two specific model types: DTMC and RBM will be discussed and they will be introduced in the following sections.

B. What is property?

In order to know what property is, we should know what is an execution over the model. It is an infinite word over the powerset and the property is a subset of the set of infinite words. The system is said to be satisfied with the property if all its executions are belonging to the property.

As stated above, the property or specification is expressed in a formal language called linear temporal logic (LTL), which is a fragment of first order logic. Specifically, it's a kind of modal logic and the modal logic is extended from the first order logic. It can express the sense of the future path that represents state transition. This explains the word ‘temporal’ in the name.

C. What is model checking?

Before introducing the model checking in detail, we should know what Buchi automaton is. Buchi is the person who proposed that automaton. A Buchi automaton [4] is a theoretical machine with states and transitions. It either accepts or rejects infinite inputs. A Buchi automaton is called non-deterministic Buchi automaton (NBA) if there exists a state with multiple outgoing transitions lead to different successive states.

As the property is specified in LTL formula, the model checking problem [5] becomes checking whether the model satisfies the LTL formula. It begins by obtaining the negation of LTL formula φ, i.e., $\neg\varphi$, then converting both the model and $\neg\varphi$ to the non-deterministic Buchi automata (NBA) and taking the product of them to check it is empty or not.

D. Symbolic model checking

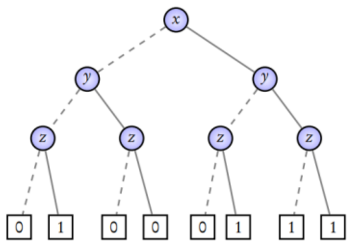
As mentioned in the introduction section, BDDs are introduced to solve the problem of states explosion problem during model checking. So, what is a BDD? BDD is reduced from a binary decision tree and binary decision tree is the representation of a Boolean function and its corresponding truth table.

Taking the example [6] below: there are 3 variables in the Boolean function, i.e., x, y, z. So, there are 8 rows in the truth table representing 8 possible assignments of them. The last column of the truth table is the value of the Boolean function. In the corresponding binary decision tree, the dotted line represents the value 0 and the solid line is 1. Every leave node is the function value of one group of assignments of the variables.

$$f(x,y,z) = x \cdot y + \bar{y} \cdot z$$

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Truth table



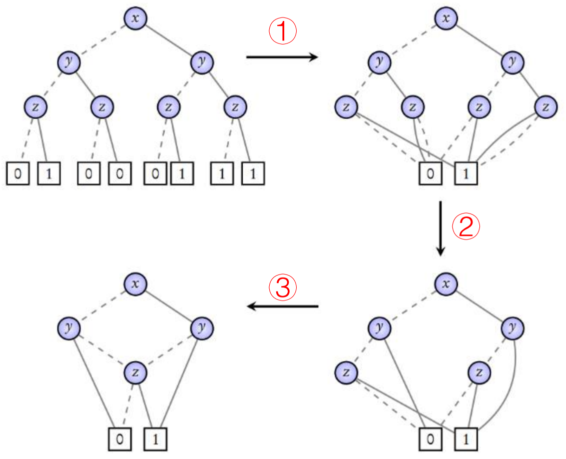
Binary Decision Tree

The process of obtaining the BDD from a binary decision tree takes 3 steps:

①Remove duplicate leaves: since all the leaf nodes have only 2 values of 0 and 1, multiple leaf nodes of 0s or 1s are not needed.

②Remove redundant tests: there are 2 z nodes, no matter the assigned value is 0 or 1, they point to the same leaf nodes.

③Remove duplicate subtrees: the remaining sub trees with the root node of z have the same structure, only one of them will be kept.



The complexity of the BDD is sensitive to the ordering of the variables. When applying the BDDs in implementation, it is important to care about the ordering of variables. The problem of finding the best variable ordering is NP-hard [7].

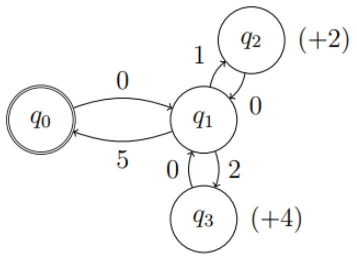
We can also do some logic operations between BDDs [8], such as: Negation, Apply (Or, And, Imply, Equivalence), Existential quantifier, and Universal quantifier. These operations will be frequently used in this project.

The practical use of BDDs is to represent the state space and transitions of the model. In short, the states are first encoded into the Boolean variables so that we can get the corresponding truth table and binary decision tree. And the BDD can be derived from them according to the rules described above. The encoding scheme of the states in the model will be discussed in detail in the following implementation section.

E. Resource based model checking

So far, the model checking is just a qualitative task to check whether a model satisfies the property. In the resource based model [9], there is resource usage on the transitions and resource gain on the states.''' Those resource or energy changes make another quantitative task possible in the resource based model checking: computing the minimal initial energy of each state to make the energy always positive during traversing the entire model.

To explain the meaning of minimal initial energy, I will illustrate it in the following example:



The state with double lines is called an accepting state and the states with a single line are called non-accepting states. The minimal energy for state q0 is 1. With that initial energy, the accumulated energy is always positive from step 0 to 15. After step 15, it comes back to q0, the accumulated energy become 1 again and it can run forever with the positive energy along the same path.

Steps	Transitions /states	Resource use/gain	Accumulated energy
0	q0	0	1
1	q0→q1	0	1
2	q1	0	1
3	q1→q2	-1	0
4	q2	2	2
5	q2→q1	0	2
6	q1	0	2
7	q1→q3	-2	0
8	q3	4	4
9	q3→q1	0	4
10	q1	0	4
11	q1→q3	-2	2
12	q3	4	6
13	q3→q1	0	6
14	q1	0	6
15	q1→q0	-5	1
…	…	…	…

The minimal initial energy for each state is also called extent [9], the algorithm for computing those extents in explicit approach can be found in [9].

F. Symbolic resource based model checking

The algorithm of computing extents in RBM described in the previous section is performed in the explicit-state model. It can also apply the symbolic approach. The algorithm used in this project is adapted from [10] (the pseudocode in Appendix A).

The computing of extents can be seen as solving an infinite energy game: $\Gamma =< G, V_0, V_1 >$ [10], where $G =< V, E, w >$ is a finite directed graph with vertices V, edges E, and a weight function w representing resource use. V_0, V_1 are the vertices belong to player 0 and 1 respectively. The resource based model is a simplified version of the game with only one player 0, so there is no partition of the vertices. Player 0 can win the game if it can build an infinite play i.e., an infinite sequence of vertices along the edges or transitions, such that the accumulated energy is always positive during the play. Computing the extents can be seen as finding the minimal required energy in every vertex for player 0 to win.

The states space and transitions need to be represented as BDDs before performing the symbolic algorithm. The method of representation will be discussed in detail in the Implementation section.

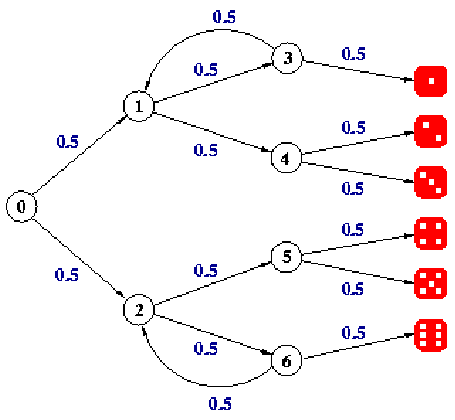
The algorithm begins by defining the variable 'minEngStates' storing pairs of the set of states and their minimal required initial energy and 'minEngPred' as the predecessor states in 'minEngStates' and the corresponding energy. Then the initial energy of each state is initialized to 0 before the start of the Do while loop.

The main part of the algorithm is a Do while loop (line 3 to 18). In general, it calculates the 'minEngPred' iteratively until minEngStates=minEngPred, namely the greatest fixed point [11] is reached. Intuitively, 'minEngPred' forces an energy level in 'minEngPred' in one step, hence the 'minEngStates' and 'minEngPred' contain energy for player 0 to win in k and k+1 steps. We can know that the energy 0 initialized before the Do while loop is sufficient for player 0 to win in 0 step.

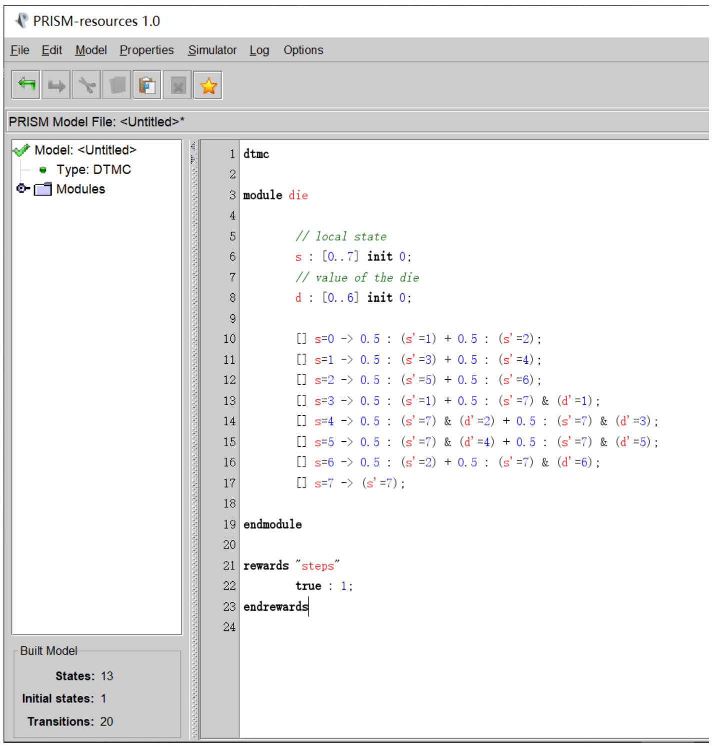
G. Model checker tool: PRISM

This project is to make an extension on PRISM model checker [1]. It is a free probabilistic model checking tools with both GUI and command line. One of the most important features of PRISM is to support probabilistic model checking. It also performs some quantitative tasks in the model checking and some quantitative results are returned such as the values of probabilities. It can be used to analyze the systems with random or probabilistic behaviors. The corresponding property language for probabilistic model is probabilistic computation tree logic (PCTL) and it can be adapted from LTL.

There are several probabilistic model types in PRISM model checker. The model Discrete-time Markov chains (DTMC) [12] is one of them. Taking the 'die' example [13] from the tutorial of PRISM official site:



It starts from circle 0, a coin will be tossed at each step, and there is a 50% chance to choose two possible methods. When you reach one of the dice values, the algorithm terminates. The snapshot below shows the results of model building: 1 initial state, 13 states, and 20 transitions are detected. They can be verified easily from the model graph.


PRISM-resources 1.0
File Edit Model Properties Simulator Log Options
PRISM Model File: <Untitled>*
Model: <Untitled>
Type: DTMC
Modules
Built Model

```
dtmc

module die

        // local state
        s : [0..7] init 0;
        // value of the die
        d : [0..6] init 0;

        [] s=0 -> 0.5 : (s'=1) + 0.5 : (s'=2);
        [] s=1 -> 0.5 : (s'=3) + 0.5 : (s'=4);
        [] s=2 -> 0.5 : (s'=5) + 0.5 : (s'=6);
        [] s=3 -> 0.5 : (s'=1) + 0.5 : (s'=7) & (d'=1);
        [] s=4 -> 0.5 : (s'=7) & (d'=2) + 0.5 : (s'=7) & (d'=3);
        [] s=5 -> 0.5 : (s'=7) & (d'=4) + 0.5 : (s'=7) & (d'=5);
        [] s=6 -> 0.5 : (s'=2) + 0.5 : (s'=7) & (d'=6);
        [] s=7 -> (s'=7);

endmodule

rewards "steps"
        true : 1;
endrewards
```

States: 13
Initial states: 1
Transitions: 20

After the model building, we can do model checking in the 'Properties' tab. A PCTL property:

P=? [F s=7 & d=5]

was added to be verified. The meaning of the property is to calculate the probability of reaching the value 5 of the die. From the snapshot below, we can know the probability is '0.16666…' i.e., '1/6'. This can be verified by hand:

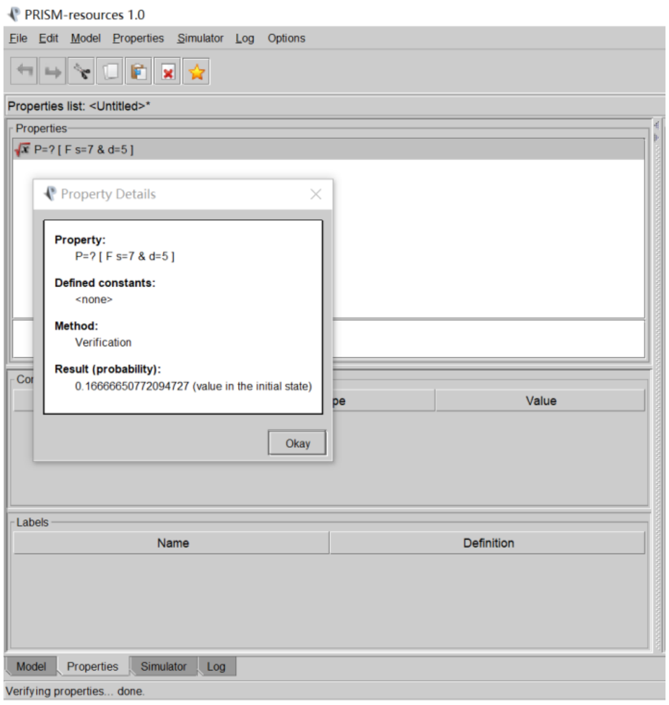
Define an event A as 'from the initial state to a state where s=7 & d=5', then

P (A)

=P(s0→s2→s5→d5) + P(s0→s2→s6→s2→s5→d5) + P(s0→s2→s6→s2→s6→s2→s5→d5) +…

$=0.5^3+0.5^5+0.5^7+\ldots$

=1/6


PRISM-resources 1.0
File Edit Model Properties Simulator Log Options
Properties list: <Untitled>*
Properties
Property Details
Okay
Value
Labels
Name
Definition
Model Properties Simulator Log

P=? [F s=7 & d=5]

Property:
P=? [F s=7 & d=5]

Defined constants:
<none>

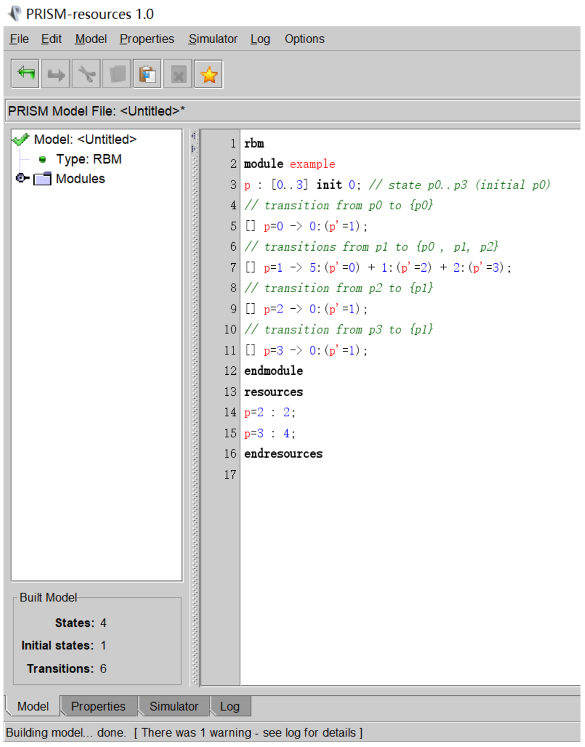
Method:
Verification

Result (probability):
0.16666650772094727 (value in the initial state)

Verifying properties... done.

III. DEFINING THE PROBLEM

The resource based model checking can be performed in an extension of the PRISM model checker developed by Aymen Qader [9]. Taking the example described in section II.E, from the snapshots below, the model was built successfully for RBM and the extent of q0 was showed correctly after the model checking.

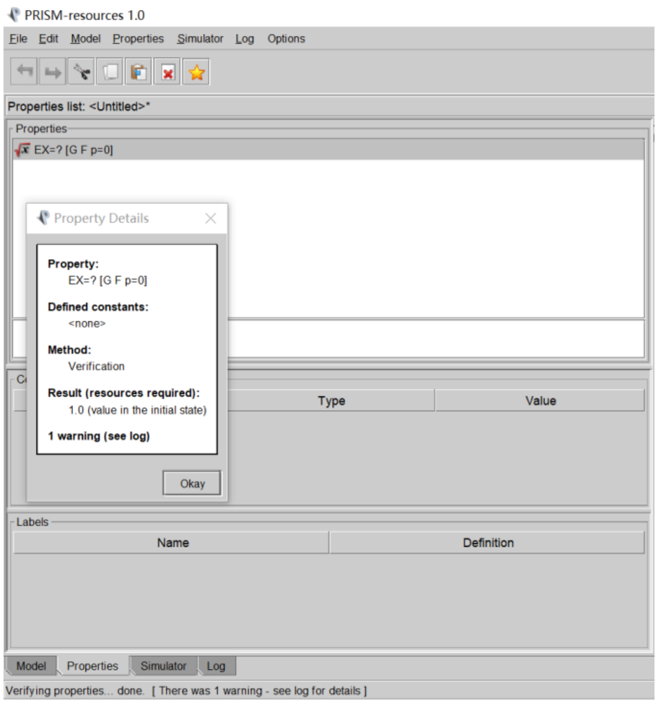

PRISM-resources 1.0
File Edit Model Properties Simulator Log Options
PRISM Model File: <Untitled>*
Model: <Untitled>
Modules
Built Model
Model Properties Simulator Log

Type: RBM

States: 4
Initial states: 1
Transitions: 6

```
rbm
module example
p : [0..3] init 0; // state p0..p3 (initial p0)
// transition from p0 to {p0}
[] p=0 -> 0:(p'=1);
// transitions from p1 to {p0 , p1, p2}
[] p=1 -> 5:(p'=0) + 1:(p'=2) + 2:(p'=3);
// transition from p2 to {p1}
[] p=2 -> 0:(p'=1);
// transition from p3 to {p1}
[] p=3 -> 0:(p'=1);
endmodule
resources
p=2 : 2;
p=3 : 4;
endresources
```

Building model... done. [There was 1 warning - see log for details]


PRISM-resources 1.0
File Edit Model Properties Simulator Log Options
Properties list: <Untitled>*
Properties
Property Details
Okay
Type
Value
Labels
Name
Definition
Model Properties Simulator Log

EX=? [G F p=0]

Property:
EX=? [G F p=0]

Defined constants:
<none>

Method:
Verification

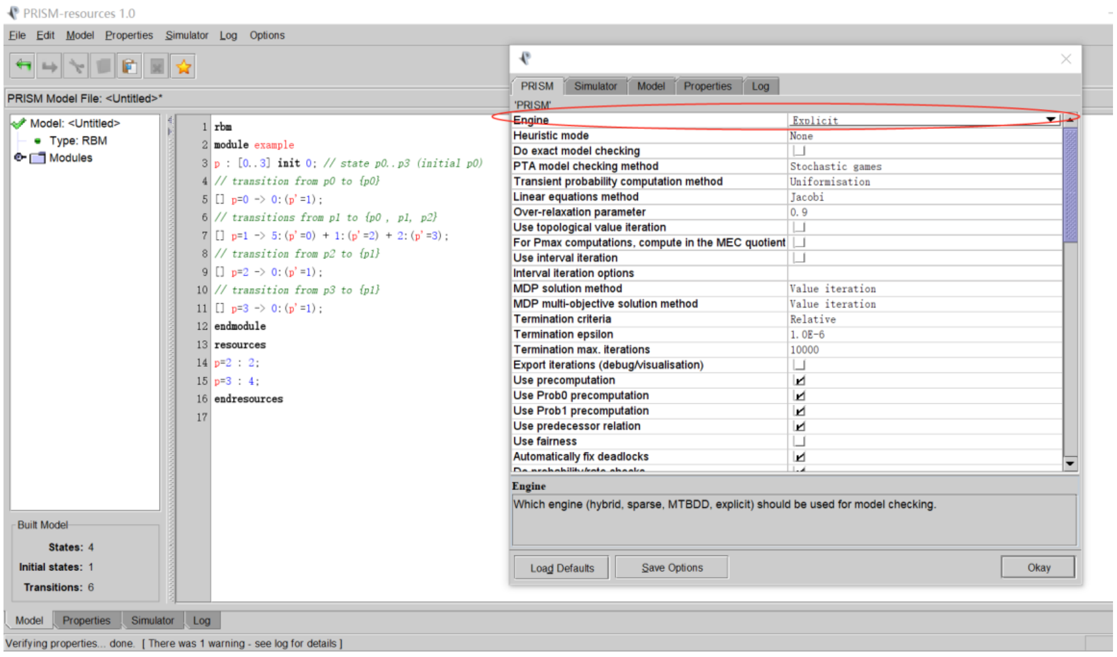
Result (resources required):
1.0 (value in the initial state)

1 warning (see log)

Verifying properties... done. [There was 1 warning - see log for details]

Both the model building and checking are performed under a specific computation engine [14]. Different computation engine implements PRISM's model checking functionality in different approach. There are four of them: "MTBDD", "sparse", "hybrid", and "explicit" in the 'Options' menu.

However, the model building and checking of RBM can only be done with the 'explicit' computation engine in his extension. The task of this project is to make a further extension on his PRISM to enable switching computation engine from 'Explicit' to 'MTBDD' to perform symbolic resource based model checking.


PRISM-resources 1.0
File Edit Model Properties Simulator Log Options
PRISM Model File: <Untitled>*
PRISM Simulator Model Properties Log
'PRISM'
Load Defaults
Save Options
Okay
Model Properties Simulator Log

Model: <Untitled>
Type: RBM
Modules

Built Model
States: 4
Initial states: 1
Transitions: 6

```
rbm
module example
p : [0..3] init 0; // state p0..p3 (initial p0)
// transition from p0 to {p0}
[] p=0 -> 0:(p'=1);
// transitions from p1 to {p0 , p1, p2}
[] p=1 -> 5:(p'=0) + 1:(p'=2) + 2:(p'=3);
// transition from p2 to {p1}
[] p=2 -> 0:(p'=1);
// transition from p3 to {p1}
[] p=3 -> 0:(p'=1);
endmodule
resources
p=2 : 2;
p=3 : 4;
endresources
```

Engine	Explicit
Heuristic mode	None
Do exact model checking	☐
PTA model checking method	Stochastic games
Transient probability computation method	Uniformisation
Linear equations method	Jacobi
Over-relaxation parameter	0.9
Use topological value iteration	☐
For Pmax computations, compute in the MEC quotient	☐
Use interval iteration	☐
Interval iteration options	
MDP solution method	Value iteration
MDP multi-objective solution method	Value iteration
Termination criteria	Relative
Termination epsilon	1.0E-6
Termination max. iterations	10000
Export iterations (debug/visualisation)	☐
Use precomputation	✓
Use Prob0 precomputation	✓
Use Prob1 precomputation	✓
Use predecessor relation	✓
Use fairness	☐
Automatically fix deadlocks	✓

Engine
Which engine (hybrid, sparse, MTBDD, explicit) should be used for model checking.

Verifying properties... done. [There was 1 warning - see log for details]

IV. Implementation

A. *Overview of implementation*

There are several packages in the code base of PRISM. The main class GUIPrism.java is in the package ‘userinterface’. The program can be built by compiling and running it.

The package ‘explicit’ is the explicit computation engine implemented in Java. The class RBMSimple.java and RBMModelChecker.java are in this package. These 2 classes are included in the extension of PRISM developed by Aymen Qader. The implementation of this project is based on it. In his extension, he made the PRISM support the model type of RBM, which was not before. Specifically, he made an extension of the model and property language to describe RBM and the LTL expression respectively. Then, an algorithm was developed to perform the RBM model checking in the class RBMModelChecker.java. However, it’s performed in an explicit approach. This project is to make it possible for the RBM model checking performs in a symbolic way.

PRISM is written in Java and C/C++. Although a very important package ‘CUDD’ storing and manipulating BDDs is written in C/C++, the implementation in this project will only focus on the Java part because the package CUDD can be manipulated by Java native interface (JNI). The package ‘jdd’ is a library for accessing CUDD via JNI.

Almost all the important functionalities of PRISM are in the package ‘prism’ and all the code developed in this project is in it. In short, the goal of the implementation is to make the computation engine of MTBDD in the option menu available for RBM. The general idea is to implement in two steps: symbolic model building and symbolic model checking.

In the model building phase, a class called SymbRBM.java was created to instantiate the model built by MTBDD computation engine. Since the MTBDD computation engine is already applicable to some model types in PRISM such as DTMC and the model languages to describe DTMC and RBM are very similar, the SymbRBM.java can be created by imitating ProbModel.java which is the class to store symbolic model of DTMC.

The model checking phase is in the class SymbRBMChecker.java. In this class, a method called ‘computeExtents’ was created to perform the symbolic RBM model checking. The method name is the same as the corresponding method in RBMModelChecker.java which is performed in an explicit way.

B. *Model Representation with MTBDDs*

There are 2 parts in the implementation of constructing MTBDDs for RBM: creating SymbRBM.java and the modification of Modules2MTBDD.translate() method.

As described before the symbolic model class to store RBM is SymbRBM.java. In fact, because the languages describing RBM and DTMC are so similar that the SymbRBM.java was created by extending the ProbModel.java directly.

The main method for translating a model into BDDs is Modules2MTBDD.translate(). And the classes to practice BDDs are JDDNode.java, JDD.java, and JDDVars.java. The class to initiate the basic data structure (a BDD node) is JDDNode.java. The manipulation of BDDs is done in JDD.java, it defines many logic operations on BDDs as mentioned in section II.D. Additionally, the JDDVars.java is a particular type of JDDNode.java, where the variable is at the root and the leaves are 0 and 1.

As mentioned before, PRISM uses the CUDD library for all of its (MT)BDD implementation at the bottom. That's written in C/C++. Some existing symbolic model checking code are therefore written in

C++, and called from the Java level using JNI. However, we also have a Java wrapper around the key data structures and operations.

Before diving into the Modules2MTBDD.translate() method, a change was made in the class prism.Prism.java: in the method buildModel(), the statement of the mandatory use of explicit computation engine for RBM was made invalid. That made the MTBDD computation engine available for the RBM instead of using explicit computation engine regardless of the options as before.

There are 6 steps in the process of obtaining the MTBDDs for the RBM in the Modules2MTBDD.translate() method:

①Getting information from the parsed model

The method begins with getting information from the parsed model file i.e., modulesFile such as PRISM variables, model type, module names, etc. Taking the example described in section II.E, there is only one PRISM variable to represent the states which is ‘p’, the model type is recognized as ‘RBM’, and the module name is ‘example’.

②Encoding

After obtaining the information of the model, some MTBDD variables will be given to describe the PRISM variables in the Modules2MTBDD.allocateDDVars() method. This process is called encoding [15].

Taking the example in section II.E, there is 1 PRISM variable: p, the range is [0..3]. In the Modules2MTBDD.allocateDDVars() method, the number of MTBDD variables to encode the PRISM variables is returned by calculating the ceiling of log2 of the range of variable. As a result, 2 MTBDD variables, say x1, x2 will be given. For instance, the state p2 will be encoded as (1,0).

As introduced in section II.D, the complexity of MTBDD is sensitive to the order of MTBDD variables. The Modules2MTBDD.sortDDVars () method help to sort out DD variables.

③Translating modules

In the previous steps, all the processing on RBM is the same as other model types, that is to say, no modification was made in those steps. In this step, changes will be made on the method Modules2MTBDD.translateModule() and the methods it calls to adapt to RBM.

The ‘module … endmodule’ clause is translated here. The method Modules2MTBDD.translateSystemDefn() was called in the method Modules2MTBDD.translateModule() to return a MTBDD for the system. A system can be made of multiple modules. The translation of system is achieved by combining the translations of modules, and the translation of module is by combining each command in that module. The whole process is carried out in a recursive manner. An MTBDD ‘sysDDs’ is returned for the following process. And the resource use on the transitions is stored in the ‘transRewards[0]’ during translation as the MTBDD for the transitions.

A method Modules2MTBDD.computeResource() was developed for translating the ‘resources … endresources’ clause, the variable ‘stateRewards[0]’ got resource gain of each state from this method and it will be used in the model checking algorithm.

Besides, a warning for the probabilistic model like DTMC was made invalid for RBM. It enforces that the sum of the probabilities of every transition must be equal to 1. It is meaningless for RBM since the ‘probabilities’ in RBM are resource use of the transitions.

The translation of an RBM is similar to a probabilistic model like DTMC. The transition MTBDD was generated as for probabilistic model. The state MTBDD was got after the method Modules2MTBDD.computeResource(). In fact, this method was written according to the method Modules2MTBDD.computeReward().

④Obtaining ‘start’

The following method Modules2MTBDD.buildinitialStates() was called to return the MTBDD for the initial states, i.e., ‘start’. No changes were made here neither, the original method was used.

⑤Instantiation of SymbRBM.java

After the previous steps, all the arguments for the constructor of SymbRBM.java are ready. A translated model was instantiated to prepare for the final return model of the method Modules2MTBDD.translate().

⑥Model.doReachability()

So far, we only got the initial states JDDNode ‘start’ and the transitions JDDNode ‘trans’, the rest of the states can be reached by the method Model.doReachability().

One modification was made here to avoid the problem that cannot reach the successive states in RBM. The reason for it is that the original code treats the resource use in RBM as probabilities. Taking the example in section II.E, the 'probability' from state q0 to q1 is defined as 0, namely, it’s impossible to transit from q0 to q1. As a result, it can't reach any successive transitions and states except the initial state q0 in the model i.e., model.numStates=1, model.numTransitions=0.

The solution for this problem is to store the resource use in another variable for the translation. On the other hand, set all the resource use of transitions as 1 when they are treated as ‘probabilities’.

The model building is done after these six steps, a resource based model represented by MTBDDs is returned at the end of the Modules2MTBDD.translate() method.

C. Model checking with MTBDDs

①Preparation for the symbolic RBM model checking:

In preparation for performing the RBM model checking in the MTBDD computation engine, one change is required on the class StateModelChecker.java. This class is created for expression to MTBDD conversions. In the class StateModelChecker.java, I made a modification in its createModelChecker() method to add a case for the RBM so that the MTBDD computation engine is available for the model checking of RBM.

Another preparation is to split the MTBDDs which represent the model into a collection of BDDs. The rules for splitting the MTBDDs are:

# The MTBDD of state space should be split into several BDDs according to the resource gain, i.e., 1 BDD for each value of resource gain.

# The MTBDD of transitions should be split into several BDDs according to the resource use, i.e., 1 BDD for each value of resource use.

This was done in the method SymbRBMChecker.computeExtents() and the method JDD.Equals() was used to do this.

②The implementation of the symbolic RBM model checking

The algorithm is implemented following the pseudocode described in section II.F. The variables ‘minEngStates’ and ‘minEngPred’ defined in line 1 of the pseudocode were created as Hashmap with key-value pairs to reflect the relations of required energy and states.

The implementation of this algorithm features the use of BDD operations. Those operations were used in line 6,11,12,14,16 of the pseudocode.

Besides, a private method SymbRBMChecker.equalJDDNodes() was created to test the stop Boolean condition of the Do while loop.

There are also some adaptations were made on the original algorithm to fit the RBM:

#The algorithm was adapted to perform a 1 player game instead of 2 players. Specifically, the BDD operation: $(\rho^{e}\rightarrow(\rho^{s}\cap bestT)_{\exists sys'})_{\forall env'}$ in line 14 of the pseudocode was simplified to ‘bestT’.

#The element ‘e-w’ of the set ‘best’ in line 7 was updated to ‘e+w-r’, where:

e= the initial energy of every successor of ‘S’ (‘S’ is one set of states in ‘minEngStates’)

w=the resource use on every outgoing transition of that 'S'

r=the resource gain of that 'S'

It is worth noting that the sign of weights ‘w’ in the implementation is positive, on the contrary, it is negative in the pseudocode. That’s why the element is ‘e+w-r’ rather than ‘e-w-r’.

D. *Classes/methods created and modified*

The table below is a summary of the key classes and methods created or modified during the implementation:

Phases	Packages	Classes	Methods	Modified/Created	Purposes
Model building	prism	prism.java	buildModel()	Modified	Avoid forcing RBM to explicit computation engine
	prism	Modules2MTBDD.java	translateModule(int, Module, String, int)	Modified	1, Avoid the warning that the probability sum must be 1 2, Storing the MTBDD represent transitions in 'transReward[0]'
	prism	Modules2MTBDD.java	computeResource()	Created	Translating the 'resources...endresources' clause
	prism	SymbRBM.java		Created	Storing the symbolic model of RBM
Model checking	prism	SymbRBMChecker.java		Created	Performing the symbolic RBM model checking

V. TESTING

A. *Unit test*

Unit test is to test a single unit while simulating all dependencies, such as methods (functions) in a class. All the unit tests of the implementation in this project are in the test.prism package.

①Modules2MTBDDTest.java

This class is to test Modules2MTBDD.translate() method. Specifically, It tests whether the MTBDD computation engine can translate the RBM. By directly reading the test model file (PRISM language txt file), using the Modules2MTBDD.translate() method, we can know that the translated result is consistent with the expected result.

②SymbRBMTest.java

This class performs the test by customizing a model of class SymbRBM.java, and then the method getModuleNames() is called to judge whether the returned results are consistent with expectations.

B. Functional test

Functional test is carried out in a black box fashion. I used two cases to test whether the model can be successfully built and the model checking is performed correctly.

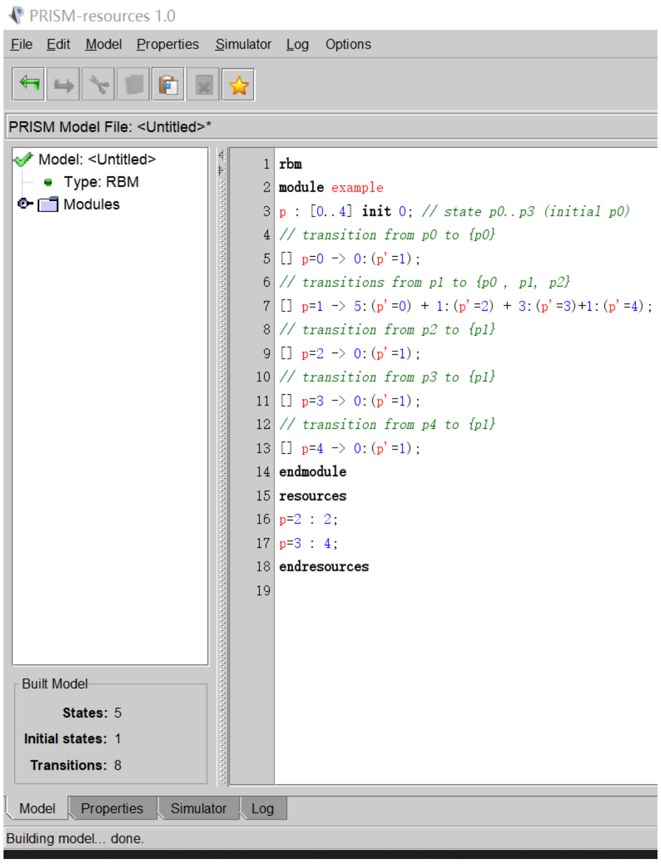
① Adding a state to the model

One line of command was added to the RBM model mentioned in section II.E:

$$[]\ p=4 \rightarrow 0:(p'=1);$$

That command defines the transitions of a new state p4. And the command defining the transitions of p1 are also modified accordingly so that both the incoming and outgoing transitions of p4 are linked to state p1.

We can see from the snapshot below, there are 5 states and 8 transitions were reached in the new model after model building. The results are consistent with our expectation.


PRISM-resources 1.0
File Edit Model Properties Simulator Log Options
PRISM Model File: <Untitled>*
Model: <Untitled>
Type: RBM
Modules
Model Properties Simulator Log

```
rbm
module example
p : [0..4] init 0; // state p0..p3 (initial p0)
// transition from p0 to {p0}
[] p=0 -> 0:(p'=1);
// transitions from p1 to {p0 , p1, p2}
[] p=1 -> 5:(p'=0) + 1:(p'=2) + 3:(p'=3)+1:(p'=4);
// transition from p2 to {p1}
[] p=2 -> 0:(p'=1);
// transition from p3 to {p1}
[] p=3 -> 0:(p'=1);
// transition from p4 to {p1}
[] p=4 -> 0:(p'=1);
endmodule
resources
p=2 : 2;
p=3 : 4;
endresources
```

Built Model

States: 5
Initial states: 1
Transitions: 8

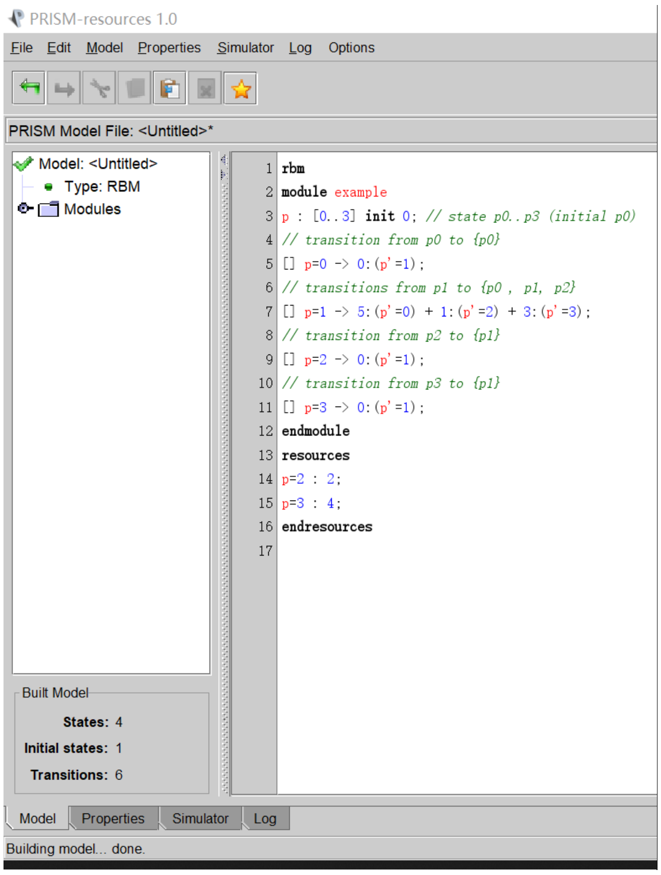
Building model... done.

②Changing the resource use on transitions

I tested the model checking function by changing the resource use in the model: the resource use of transition from p1 to p3 is modified from 2 to 3, we can calculate the minimal required energy of p0 by hand with the table below, and the result is still 1:

Steps	Transitions /states	Resource use/gain	Accumulated energy
0	q0	0	1
1	q0→q1	0	1
2	q1	0	1
3	q1→q2	-1	0
4	q2	2	2
5	q2→q1	0	2
6	q1	0	2
7	q1→q2	-1	1
8	q2	2	3
9	q2→q1	0	3
10	q1	0	3
11	q1→q3	-3	0
12	q3	4	4
13	q3→q1	0	4
14	q1	0	4
15	q1→q3	-3	1
16	q3	4	5
17	q3→q1	0	5
18	q1	0	5
19	q1→q3	-3	2
20	q3	4	6
21	q3→q1	0	6
22	q1	0	6
23	q1→q0	-5	1
24	q0	0	1
…	…	…	…

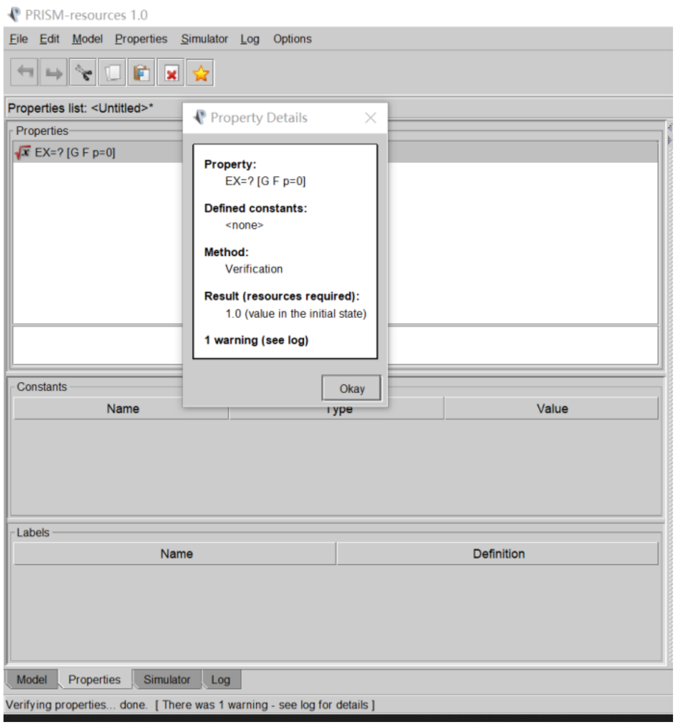
The test result in the PRISM shows the same extent.


PRISM-resources 1.0
File Edit Model Properties Simulator Log Options
PRISM Model File: <Untitled>*
Model: <Untitled>
Type: RBM
Modules
Model Properties Simulator Log

```
rbm
module example
p : [0..3] init 0; // state p0..p3 (initial p0)
// transition from p0 to {p0}
[] p=0 -> 0:(p'=1);
// transitions from p1 to {p0 , p1, p2}
[] p=1 -> 5:(p'=0) + 1:(p'=2) + 3:(p'=3);
// transition from p2 to {p1}
[] p=2 -> 0:(p'=1);
// transition from p3 to {p1}
[] p=3 -> 0:(p'=1);
endmodule
resources
p=2 : 2;
p=3 : 4;
endresources
```

Built Model
States: 4
Initial states: 1
Transitions: 6

Building model... done.



EX=? [G F p=0]

Property:
EX=? [G F p=0]

Defined constants:
<none>

Method:
Verification

Result (resources required):
1.0 (value in the initial state)

1 warning (see log)

Verifying properties... done. [There was 1 warning - see log for details]

VI. Analysis

Compare to the explicit approach, the symbolic approach is more complicated in both model building and checking. Unlike the explicit approach to store the states and transitions in a straightforward way, the symbolic approach employs an abstract and succinct data structure BDD to represent the state space and transitions. Besides, there are many abstract logic operations on BDDs in the algorithm of model checking to make the algorithm more efficient.

However, the symbolic approach also has some simplifications. There is no distinction between accepting and non-accepting states in the symbolic approach, all the states are non-accepting. And all the required energies of states are initialized as 0, not just accepting states in the explicit approach.

There is also something in common between the two approaches. The calculating of the elements of the set best in the symbolic algorithm is the same as the RBMModelChecker.minimise() method in the explicit approach. They both update the extents of states by considering the extents of the successors and weights on the outgoing transitions and the gains of themselves.

VII. Conclusions

In this project, I developed an extension based on the PRISM developed by Aymen Qader. The MTBDD computation engine becomes available for the RBM model building and checking. Tests have been conducted to verify the correctness of the implementation.

For future work, the scalability analysis of symbolic RBM model checking needs to be done to verify the efficiency of the data structure of BDD. The synthesis or strategy generation functionality of PRISM can continue to be developed after the symbolic RBM model checking. In addition, the development of resource based game (RBG) model checking under the MTBDD computation engine is also worth studying.

References

[1] PRISM model checker, https://www.prismmodelchecker.org/

[2] "Finite-state machine", Wikipedia, https://en.wikipedia.org/wiki/Finite-state_machine

[3] "Kripke structure", Wikipedia, https://en.wikipedia.org/wiki/Kripke_structure_(model_checking)

[4] "Büchi automaton", Wikipedia, https://en.wikipedia.org/wiki/B%C3%BCchi_automaton

[5] B. Srivathsan, Chennai Mathematical Institute, "Algorithms for LTL", https://www.cmi.ac.in/~sri/Courses/NPTEL/ModelChecking/Slides/Unit8-Module1.pdf

[6] B. Srivathsan, Chennai Mathematical Institute, "Binary Decision Diagrams (BDDs)", https://www.cmi.ac.in/~sri/Courses/NPTEL/ModelChecking/Slides/Unit11-Module1.pdf

[7] "Binary decision diagram", Wikipedia, https://en.wikipedia.org/wiki/Binary_decision_diagram#Variable_ordering

[8] Ehab Al-Shaer, Cyber Defense and Network Assurability (CyberDNA) Center School of Computing & Informatics University of North Carolina, "Network Configuration Verification and Analysis Using BDDs", https://www.cs.princeton.edu/courses/archive/spring10/cos598D/lecture-17-BDD-Ehab.pdf

[9] Aymen Qader, Electronics and Computer Science Faculty of Engineering and Physical Sciences University of Southampton, "A Tool for Model Checking and Optimal Synthesis using Resource-Aware Automata"

[10] Shahar Maoz Or Pistiner Jan Oliver Ringert, School of Computer Science Tel Aviv University, "Symbolic BDD and ADD Algorithms for Energy Games"

[11] Mickael Randour, Université de Mons, Jean-Francois Raskin, Université Libre de Bruxelles, "Strategy Synthesis for Multi-dimensional Quantitative Objectives"

[12] PRISM lectures discrete-time Markov chains, https://www.prismmodelchecker.org/lectures/pmc/02-dtmcs.pdf

[13] PRISM tutorial example die, https://www.prismmodelchecker.org/tutorial/die.php

[14] PRISM Manual Computation Engines, https://www.prismmodelchecker.org/manual/ConfiguringPRISM/ComputationEngines

[15] David Anthony Parker, School of Computer Science Faculty of Science University of Birmingham, Implementation of Symbolic Model Checking for Probabilistic Systems

APPENDICES

A. *Algorithm pseudocode*

```
define minEngStates, minEngPred as ($\mathbb{N}\times$ Set of States)
add (0, TRUE) to minEngPred
while minEngStates $\neq$ minEngPred do
    minEngStates = minEngPred
    empty minEngPred
    remaining = $\{s \in S \mid (e,S) \in$ minEngStates$\}$
    for increasing best $\in \{0\} \cup \{0 \leq e - w \leq$ maxEng $\mid (e,S) \in$ minEngStates $\wedge (w,T) \in$ weights$\}$ do
        define bestT as Transitions
        bestT = $\emptyset$
        for $(v,T) \in$ weights do
            S = $\{s \in S_{e_v} \mid (e_v, S_{e_v}) \in$ minEngStates $\wedge\, e_v - v \leq best\}$
            add T transition to S to bestT
        end for
        B = (forceEnvTo bestT transitions) $\cap$ remaining
        add (best, B) to minEngPred
        remove B from remaining
    end for
end while
return minEngStates
```

B. *Project management*

The original project plan:

	Jun			Jul				Aug			
	week38	week39	week40	week41	week42	week43	week44	week45	week46	week47	week48
Background research	▨	▨									
Programming preparation	▨	▨	▨								
The design of the extension of the PRISM				▨							
The implementation					▨	▨	▨	▨			
Testing of the extension									▨	▨	
The summary report											▨

The actual project progress:

	Jun			Jul				Aug				Sep		
	week38	week39	week40	week41	week42	week43	week44	week45	week46	week47	week48	week49	week50	week51
Java programming basic learning	▨	▨												
Background research: model checking course learning			▨	▨										
PRISM model checker case study					▨									
Background research: algorithm paper study					▨	▨	▨	▨						
IDE and PRISM code base familiarization and set up									▨	▨				
The design of the extension of the PRISM											▨			
The implementation											▨	▨	▨	
Testing of the extension														▨
The summary report												▨	▨	▨

C. *Risk Assessment*

#	Problem	Loss	Prob	Risk	Countermeasure
1	Lack of java skills	5	3	15	Take the Java crash courses and get started with the PRISM code base as early as possible.
2	Difficulty in setting up the IDE and PRISM code base	5	2	10	Seek help/guide from supervisor.
3	Implementation time period underestimated	3	4	12	Modify the roadmap according to the supervisor's advices.
4	Difficulty in understanding the algorithm before implementation	2	5	10	Get familiar with the structure of the algorithm and verify it in the Java code.
5	Lost in the details of the algorithm paper	1	4	4	Identify the points of the paper by discussing with supervisor in the weekly meeting