Journal of Computer Science 7 (2): 225-233, 2011
ISSN 1549-3636
© 2011 Science Publications

Analysis of the Model Checkers' Input Languages for Modeling Traffic Light Systems

[1]Pathiah Abdul Samat, [2]Abdullah Mohd Zin and [2]Zarina Shukur
[1]Faculty of Computer Science and Information Technology, University Putra Malaysia, 43400 Serdang, Selangor, Malaysia
[2]Faculty of Technology and Information Science, University Kebangsaan Malaysia 43600 Bangi, Selangor, Malaysia

Abstract: Problem statement: Model checking is an automated verification technique that can be used for verifying properties of a system. A number of model checking systems have been developed over the last few years. However, there is no guideline that is available for selecting the most suitable model checker to be used to model a particular system. **Approach:** In this study, we compare the use of four model checkers: SMV, SPIN, UPPAAL and PRISM for modeling a distributed control system. In particular, we are looking at the capabilities of the input languages of these model checkers for modeling this type of system. Limitations and differences of their input language are compared and analyses by using a set of questions. **Results:** The result of the study shows that although the input languages of these model checkers have a lot of similarities, they also have a significant number of differences. The result of the study also shows that one model checker may be more suitable than others for verifying this type of systems **Conclusion:** User need to choose the right model checker for the problem to be verified.

Key words: Model checking, distributed control system, user interface, Linear Temporal Logic (LTL), Computational Tree Logic (CTL), Distributed Control System (DCS), Probabilistic Computation Tree Logic (PCTL), State Transition Diagram (STD)

INTRODUCTION

Model checker (Clarke *et al*., 1999; Berard, 2001; Razali and Garratt, 2010) is a verification tool that is embedded with powerful technique and popularly used in verifying a software or hardware system. There are many model checking systems which are developed; the most popular are SMV (McMillan, 1999; Islam *et al*., 2010), UPPAAL (Bengtsson *et al*., 1995; El Emary and Al Rabia, 2005), SPIN (Holzmann, 2003) and PRISM (Marta, 2003, Chandren *et al*., 2010). Each of the model checkers comes in a package with its own input language which has strict notations and features (Bhaduri and Ramesh, 2004; Djavanroodi *et al*., 2008). The SMV language is used to describe a finite state transition relational model. Properties of the model to be verified are specified in a temporal logic, known as Computational Tree Logic (CTL). SPIN accepts design specifications written in the verification language Promela and it accepts correctness claims specified in the syntax of standard Linear Temporal Logic (LTL). In UPPAAL, systems to be verified have to be represented with a collection of timed automata. PRISM known as probabilistic model checking is an automatic procedure for establishing if a desired property holds in a probabilistic system model. Properties to be checked against the constructed model are specified using temporal logic Probabilistic Computation Tree Logic (PCTL).

By using a model checker, all possible behaviors or properties of the system can be checked to determine whether they satisfy the system's specification. If any of the behaviors is not satisfied, the model checkers will produce a counterexample.

In this study, we would like to compare the input languages of various model checkers in modeling a distributed control system. A Distributed Control System (DCS) (Trentesaux, 2009) refers to a control system in which the controller elements are not centrally located but are distributed throughout the system with each component sub-system controlled by one or more controllers. The entire system of controllers is connected by a network for communication and monitoring. A good example of a simple DCS is a traffic light system.

Corresponding Author: Pathiah Abdul Samat, Faculty of Computer Science and Information Technology, University Putra Malaysia, 43400 Serdang, Selangor, Malaysia

The comparison of the input languages of model checkers will focus into two main aspects: (i) The input languages for modeling systems to be model checked and (ii) The language for formalization of properties of the system. Specifically, we would like to answer the following questions:

- Q1: Is there any difficulty in describing the system to be checked into the input language of the model checkers?
- Q2: Is there any significant difference between the input languages of the model checkers?
- Q3: Is there any part of the system that cannot be modeled by the model checkers?
- Q4: Based on Q3. If there answer is yes, is there any solution for the above problem?
- Q5: Is there any difficulty in representing the properties?
- Q6: Based on Q5. If there answer is yes, what type of assistant needed to represent these properties?
- Q7: What are the similarity exist in all the modeling languages while modeling the system?

Related works: A few comparative reports are available on this issue in different domains. Jeffrey *et al.* (2000), the performance of five different model checking techniques was compared such as SPIN, Fc2Tools, SMV, SMC and IOTA. The tools are used to analyze the deadlock property of the example system. The attribute of performance is taken based on memory and CPU time. The result shows that the IOTA tool is more efficient than the other four tools in the verification of the deadlock property. The performance of SPIN and NuSMV (Choi, 2007) were compared on a model of Flight Guidance System (FGS). The purpose of this study is to investigate whether SPIN more suitable than NuSMV in term of scability and usability. This study claim that SPIN performs poorer than NuSMV on the one-sided synchronous FGS model, but scales better to asynchronous two sided FGSs when they manage to handle the one-sided FGS using SPIN.

There is another study to compare the performance of probabilistic model checkers (Jansen *et al*., 2008). This study investigate the strengths and weaknesses of the various model checking tools such as ETMCC, MRMC, PRISM (sparse and hybrid mode), YMER and VESTA. The result shows that YMER is by far the fastest tool and its memory usage is remarkably constant, hardly varying with the model size. Unfortunately, YMER only supports bounded and interval until formulas. In particular, YMER outperforms the other statistical model checker VESTA: VESTA's memory consumption is also rather constant, but more in the order PRISM's memory usage. ETMCC performs the worst in terms of memory and frequently was unable to check models that were easy for the other tools. MRMC mostly performs better than PRISMs both in time and memory.

MATERIALS AND METHODS

Description of the model checkers: SMV's language for the description of automata is based on declarative approach which clearly oriented towards describing a "possible next state" relation between states seen. SMV input language start with keyword MODULE followed by module's name. A MODULE consists of some definitions (type declarations and assignments) that can be reused.

Promela is a verification modeling language for SPIN model checker. Promela programs consist of processes, message channels and variables. Processes are global objects. Message channels and variables can be declared either globally or locally within a process. Processes specify behavior, channels and global variables define the environment in which the processes run. A process can wait for an event to happen by waiting for a statement to become executable.

PRISM language comprises modules and variables. A system is composed a number of modules which can interact with each other. A module contains a number of local variables. Each variable has its own values which constitute as a state of the module. The global state of the whole system is determined by the local state of all modules. The behavior of each module is described by a set of commands. A transition is specified by giving the new values of the variables in the module, possibly as a function of other variables.

UPPAAL consists of a model-checking engine and a graphical user interface. The user interface consists of three parts: system editor, simulator and verifier. The system editor enables the user to model a real time system as a network of timed finite states automata global or local variables and clocks. The automata templates have to be entered by means of a graphical notation that resembles the standard notation for timed automata. The transitions to be synchronized have to be labeled by ch! and ch?. The abbreviation "ch" refers to the communication channel name which is used in a template. The simulator offers the possibility to the user to interactively run the system and check if he made some trivial mistakes in its modeling or design. The verifier allows the user to enter the properties to be verified.

Traffic light system: The traffic light according to Wikipedia, also known as traffic signal is a signaling device positioned at a road intersection, pedestrian crossing or other location. A traffic signal is typically controlled by a controller which is placed inside a cabinet mounted on a concrete pad.

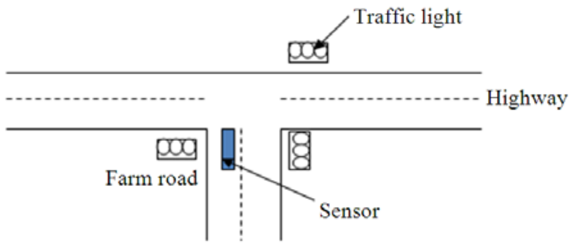


Fig. 1: A 3-way traffic light system

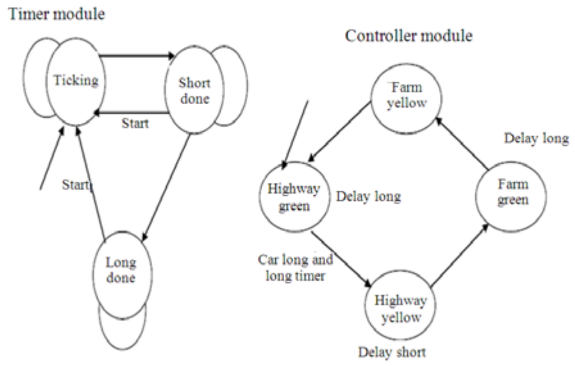


Fig. 2: State transition diagrams for Fig 1

The cabinet typically contains a power panel to distribute electrical power in the cabinet; a detector interface panel to connect to loop detector and other detectors; detector amplifier and other components.

Traffic controllers use the concept of phases. For instance, a simple intersection may have two phases: North/South and East/West. A 4-way intersection with independent control for each direction and each left-turn will have eight phases. Traffic signals must be instructed when to change phase. In some traffic light systems, phase change occur based on timer. Many traffic light systems are now sensor-based system. The sensor is buried in the pavement to detect the presence of traffic light waiting at the light. Thus, it can avoid giving the green light to an empty road while motorists on the different route are stopped. A timer is frequently used as a backup in the case the sensors fail. There are two main components in a sensor-based traffic lights system: controller and timer.

For this particular study, we will use a sensor-based traffic light system which uses sensors for a 3-way intersection, as shown in Fig. 1.

Figure 2 show State Transition Diagram (STD) for timer and controller.

The timer has three sequences of states: ticking, short-done and long-done. Start acts as reset signal. The next state of ticking is either short-done or maintain at the original state. The next state of short-done is either long-done or maintain at original state. The controller has four states: farm-yellow, highway-green, farm-green and highway-yellow. If the state of controller is highway-green and there are many cars and time given is long then the next state is highway-yellow. If the state of controller is highway-yellow and the time given is short then the next state is farm-green. If the state of controller is farm-green and there is no car and time given is long then the next state is farm-yellow. If the state of controller is farm-yellow and time given is short then the next state is highway-green.

Properties of the traffic light system: The above traffic light system should satisfy at least three properties:

- to ensure either the farm road or the highway always has a red light
- if a car appears on the farm road then it will eventually get a green light
- the highway light turns green infinitely often

Modeling in SMV: In SMV, model description and specification of properties are written in the same file. The model of the traffic light system is described by using four modules; main, timer, controller and light.

The module main is used to enable messages to be shared between modules. Based on the code listed below, we can see that controller shares messages with timer and passes a Boolean value.

Module main VAR

```
Farmcars:  boolean;
Cntl:      controller(farmcars, time);
Time:      timer(cntl.start-timer);
Lamp:      light(cntl.state);
```

The controller has two variables: state and start-timer. The evolving state of controller is described by using the next operator. For example, the next state of controller is highway-yellow if the current state is highway-green and there is a car on the farm road and the time taken at that time is long-done.

```
MODULE controller(cars, time)
VAR
state: {highway-yellow, highway-
       green, farm-yellow, farm-
       green};
start-timer : boolean;
```

```
ASSIGN
 init(state) := highway-green;
 next(state) := case
      state = highway-green & cars &
              time=long-done :
              highway-yellow;
      state = highway-yellow &
              (time=short-done) :
              farm-green;
      state = farm-green & (!cars |
              time=long-done) :
              farm-yellow;
      state = farm-yellow & (time=
              short-done) : highway-
              green;
       1: state;
  esac;
```

The timer has three states; ticking, short-done and long-done which are declared under variable state. The start-time which is shared between controller and timer is used to initialize a ticking in timer. If state of time is ticking then the next state is either ticking or short-done.

The light has three states of lamps: green, yellow and red which are declared under variables farm-light and highway-light. The evolving state of farm-light and highway-light are determined by using the next operator. For example, the next state of farm-light is yellow if the state of controller is farm-yellow.

SMV only allows the properties to be written in CTL and must be written in the same file under a keyword SPEC. The three properties of the traffic light system can be described as follows:

- (1)AG (lamp.farm-light = red | lamp.highway-light = red)
- (2) AG (AF(farmcars ->lamp.farm-light = green))
- (3) AG (AF lamp.highway-light = green)

Modeling in SPIN: In SPIN, the traffic light model and the properties must be written in two separate files. The model is described by using one proctype and two inlines. We also use in it to initialize the processes declare in proctype.

We describe our processes in a proctype called controller. Since operation of controller, involves nondeterministic selection, we use guarded expressions:

```
proctype controller(bit crs){
 sst1: if::st_cnl_1(state)->
           car_2(crs);
          get_cr2(crs)->
        if:: st_t_long(time) ->
              state_cnl!hg
            :: state_cnl?state->
              goto sst2
          fi
     fi;
sst2: if::st_cnl_2(state)->
        if::st_t_short(time)->
            state_cnl!fg
          ::state_cnl?state->
            goto sst3
          fi
     fi;
sst3: if::st_cnl_3(state)->crs=0->
        if:: st_t_long(time)->
              state_cnl!fy
            :: state_cnl?state->
              goto sst4
          fi
      fi;
sst4: if::st_cnl_4(state)->
        if:: st_t_short(time)->
              state_cnl!hg
            :: state_cnl?state->
               goto sst1
           fi
     fi;
```

Instructions in proctype controller are executed by calling a number of inline functions.

For example, one of the inline function is timer the main function of this inline is to initialize ticking and to determine the current state of timer. To indicate the next state in Promela, we need to use if.. else statement:

```
inline timer(tt){
if
  ::tt==1->state_time!ticking
fi;
do::state_time?time->
      if
      ::time=ticking->
         state_time!ticking
      ::else->
         state_time!short_done
      fi;
     if
     ::time=short_done->
        state_time!short_done
      ::else->state_time!long_done
      fi
  ::state_time!time->break
od
}
```

The communication between controller and its components is executed via message channels such as state_cnl, state_time, start_time, f_light, h_light and farmsc.

The properties of traffic light are specifies as a never claim. We choose never claim because our aims is to check the behavior that should never occur. The first property can be stated as:

```
never {/*![](lamp 1|| lamp 2)
TO_init:
    if
    ::(!(( lamp 1))&&!(( lamp 1))->goto accept_all
    ::(1) ->goto TO_init
    fi:
 accept_all:
    Skip
}
```

The second property is specified as follows:

```
never {/*![](<>cs2-lamp 4))*/
TO_init:
        If
        ::(!(( lamp 4))&&!((cs2))->goto accept_all
        ::(!(( lamp 4)) ->goto TO_S4
        ::(1) ->goto TO_init
        fi:
TO_S4
   if
   ::(!(( cs2)) ->goto accept_all
   ::(1) ->goto TO_S4
   fi:
accept_all:
    Skip
}
```

The others property we would like to check is the highway turn green infinitely often. The property is specifies as below:

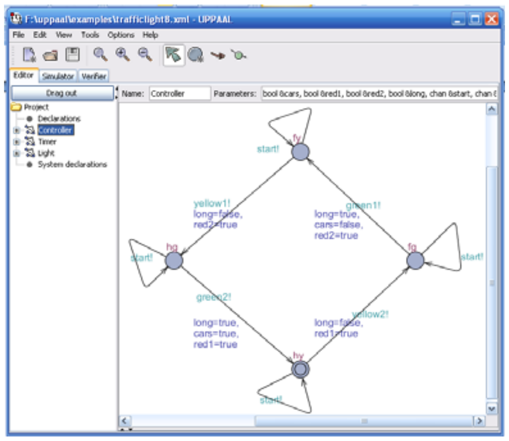
```
never {/*![](lamp3)*/
TO_init:
    If
    ::(!(( lamp3)))->goto accept_S4
    ::(1) ->goto TO_init
    fi:
 accept_S4:
    fi
    ::(!(( lamp3)))->goto accept_S4
    fi:
}
```

Modeling in PRISM: In PRISM, the model and its properties must also be described in two separate files. The model for the traffic light system is described by using three modules; timer, controller and light. The descriptions of module controller is coded as below:

```
module controller
 stateclr:[1..4] init hwayyellow;
 startime:[0..1] init no;
 cars:[0..1] init no;

 [] cars=no -> cars'=yes;
 [] cars=no -> cars'=no;
 [] cars=yes -> cars'=no;
 [] cars=yes -> cars'=yes;
 [b] (stateclr=hwaygreen) &
      (cars=yes) &
      (timestate=longdone)
       -> (stateclr'=hwayyellow);
 [b] (stateclr=hwayyellow) &
      (timestate=shortdone)
      -> (stateclr'=frgreen);
 [c] (stateclr=frgreen) &
       (cars=no)|
      (timestate=longdone)
       -> (stateclr'=fryellow);
 [c] (stateclr=fryellow) &
      (timestate=shortdone)
      -> (stateclr'=hwaygreen);
 [a] (stateclr=hwaygreen) &
      (cars=yes) &
      (timestate=longdone) |
      (stateclr=hwayyellow) &
      (timestate=shortdone) |
      (stateclr=frgreen) &
      (cars=no) &
      (timestate=longdone) |
      (stateclr=fryellow) &
      (timestate=shortdone)
      ->startime'=yes;
endmodule
```

The states of each module are defined as constant integer. In order to strengthen a guard in a command we use the symbols like = and and | which stand for equal and or, respectively. The statement on the right hand side of $\rightarrow$ is executed if the guard return true. The action name is used to force two modules to make transitions. simultaneously. For example, a is placed inside the square bracket in command nine of controller and command one of timer. By default, all modules are combined using the standard CSP parallel composition (i.e., modules synchronize over all their common actions).



Name: Controller Parameters: bool &cars, bool &red1, bool &red2, bool &long, chan &start, chan &

Fig. 3: The template of controller

The traffic light problem which describe in MDP of type model allows the modules themselves to make nondeterministic choice. For example, state 2 until 5 in timer will be nondeterministic choice.

PRISM does not provide any mechanism for parameter passing for its module. The only way to link one module to another module is by adding states to the related modules. For example, the transition of first command in light only occurs if the current state of controller is fryellow.

The properties are specified in PRISM language by using PCTL, which is an extension of classical temporal logic CTL. The properties of the traffic light system are specified in form of p>=1 [true U p] stand for "with the probability of 1 eventually p is satisfied for all states". For example, if we want to check either the farm road or the highway road has a red light, the property is specified as below:

P>=1[true U(farmlight = red | hwaylight = red)]

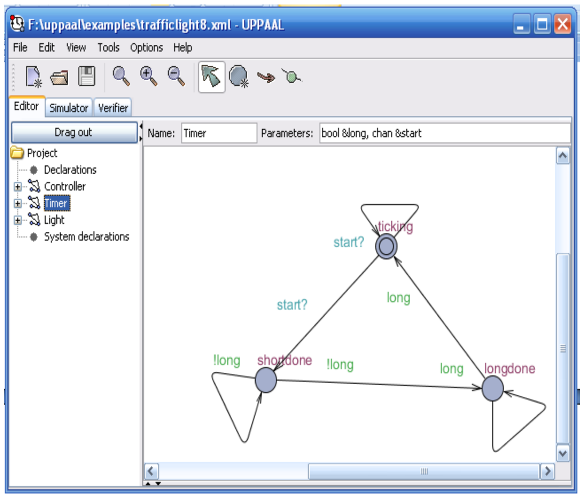
In this model, we also interested to check if a car appears on the farm road, it will eventually get a green light and it could be specified as follow:

P>=1[true U cars = yes = > (farmlight = green)]

The others property that we would like to check is the highway light turn green infinitely often. The property is specified in this form:

P>=1[true U (hwaylight = green)]

For MDP model, properties using the P operator actually reason about the minimum or maximum probability, over all possible resolutions of non-determinism, that a certain type of behavior is observed.



Name: Timer Parameters: bool &long, chan &start

Fig. 4: The template of timer

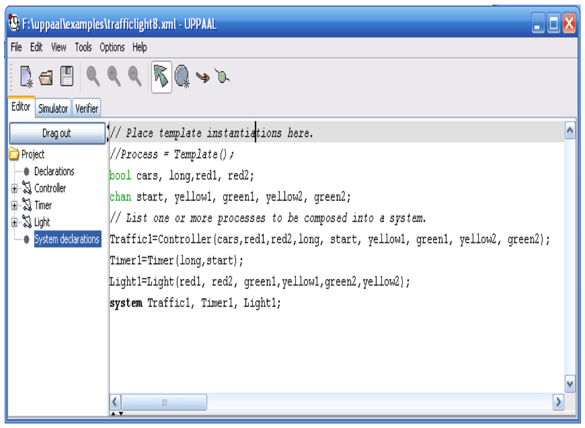
Modeling in UPPAAL: The first step for modeling in UPPAAL is to insert a template in the editor pane. For our case, we have defined three templates; Controller, Timer and Light. Figure 3 shows the controller's template which contains nine parameter such as bool &cars, bool and red1, bool and red2, bool &long, chan &start, chan and yellow1, chan and green1, chan and yellow2 and chan and green2. bool and cars, bool &red1, bool and red2, bool and long, chan and start, chan and yellow1, chan and green1, chan and yellow2 and chan and green2.

Controller has four states or locations stated as fg, hg, hy and fy stand for farm green, highway green, highway yellow and farm yellow, respectively. The location hg synchronizes with others automaton via start! and green2!.

The timer's template contains two parameters such as bool and long and chan &start. Figure 4 shows the template of timer.

The timer has three locations which are labeled as ticking, shortdone and longdone. Ticking is self-loop which synchronizes with controller via start? The others location is executed based on the update stated on edge corresponding with edge on controller.

In the system declaration, templates are instantiated into process. The system declaration in Fig. 5 shows that the template controller is instantiated to process Traffic1, template Timer is instantiated to the processes Timer1. Lastly, the template Light is instantiated to process Light. All of the processes are executed in parallel and declared active as one system.


F:\uppaal\examples\trafficlight8.xml - UPPAAL
File Edit View Tools Options Help
Editor Simulator Verifier
Drag out
Project
Declarations
Controller
Timer
Light
System declarations

```
// Place template instantiations here.
//Process = Template();
bool cars, long,red1, red2;
chan start, yellow1, green1, yellow2, green2;
// List one or more processes to be composed into a system.
Traffic1=Controller(cars,red1,red2,long, start, yellow1, green1, yellow2, green2);
Timer1=Timer(long,start);
Light1=Light(red1, red2, green1,yellow1,green2,yellow2);
system Traffic1, Timer1, Light1;
```

Fig. 5: System declaration for a traffic light

In UPPAAL, the properties are specified in CTL. Properties are written in a separate file with file name extension .q. We use symbols like | and ==, stand for or and equality, respectively. We also use =>, mean that implication if the expression before => true for the following properties. The properties of interest are written as below:

//either the farm road or the highway has a red light
A[] (red 1 == true | red 2 == true)

//if car appears on the farm road, it will eventually get a green light
A[] (cars == true implu light 1. yw1)

//the highway light rurns green infinitely often
E<> (Light 1. gr2)

RESULTS

We discuss the answer for each of the questions listed earlier based on our experience in model checking the traffic light system.

Q1 is regarding the difficulty in describing the system by using the input language of the model checkers. The input language of SMV allows the description of the model to be directly translated from the state diagram. Each state diagram can be translated into one module. Interaction between these modules can be done through shared variables. However it is difficult to translate the state diagram to input languages of UPPAAL, PRISM and SPIN. UPAAL does not support the concept of modules. Each state diagram is represented as a template. Furthermore, UPPAAL does not support non-deterministic behavior. PRISM language does not support representation of enumerated data type, so data in this form have to be converted into numerical representation. Message sharing mechanism between modules in PRISM is different from other model checkers. In SPIN, each state diagram can be representation as one proctype. However, there is a difficulty in passing parameters between proctypes.

Q2 is regarding significant differences between the input languages of the model checkers. It is obvious that there are differences between the languages. For example, in SMV, the synchronization is described by parallel assignment. This is done by copying array or data to a module. In UPPAAL, the synchronization can be described by instantiating the template to a process, followed by identifying processes to be synchronized by using operators !(send) and ?(receive). In PRISM, the synchronization is described by using the action name. The modules which have the same action name will be activated in parallel. In SPIN, the synchronization is described by using message channel through the method ! For sending and ? For receiving.

Q3 is to answer if any behavior of the system that cannot be modeled by using the input languages of model checkers. In this case study, the system is successfully modeled in SMV and PRISM. However we have problems to model the start time and to synchronize the start time with timer in UPPAAL. In SPIN, we will have a problem to pass parameter from controller to timer and light if all of them are declared as proctype.

Q4, is about methods used to solve problems arise in Q3. In UPPAAL, the problem is solved by placing start time as send message to each location in controller and at one particular time controller can only receive one start time message. We also add four locations in the light automaton. In SPIN, we decided to have only one proctype, that is controller and the others are described as inlines. The disadvantage for using this approach is that the codes for describing the model become longer.

Q5 is about the difficulty to represent the properties. We have no problem in specifying properties in SMV. UPPAAL does not allow nested path quantifier while SPIN requires identifiers to be declared as global. Since, PRISM is designed as a probabilistic model checker; it can return not only true or false values but also numerical values.

In order to solve problems stated in Q5, we suggest that all four model checkers provide type checking assistant to assist users to formulize properties.

For Q7, we found that all model checkers need procedures to model the component of the system. Each of procedure requires a special declaration and description of state, initialization and transition. On the other hand, each component requires synchronizing to achieve one of state-machine system objectives.

DISCUSSION

Based on the listed questions above, we classified our result into two groups.

Firstly, the input language of all four model checkers allows us to model traffic light system from state diagrams, although, there are significant differences in modeling synchronization of various components of the traffic light system. We have also shown that all of the states, transitions and initializations can be successfully modeled in all of the input languages. All of model checkers' input languages have their own notations and symbols. For example, in SMV and PRISM, each STD is represented as a module. However PRISM's module not like as a function construct. SPIN use proctype and #inline constructs whereas UPPAAL prefer to use template.

Secondly, is about formulization of properties. SMV provides a lot of temporary logic operators for formulizing the properties. It has a variety of nesting path of quantifier and allows almost all of logical symbols. The other model checkers which accepts nested path of quantifier are SPIN and PRISM. UPPAAL is less elegant to formulizing properties because it does not allow nesting path quantifier.

CONCLUSION

From the study that has been carried out, it seems that all of the model checkers share a lot of similarities. All of them require us to model the state of the system as well as the state transition. All of them provide mechanism for selection, synchronization, message passing and message sharing. Users have to provide a list of properties to be checked against the model.

However, there are some significant differences between these model checkers that may cause some problems for users to move from one model checker to another. In particular, there is a significant different in term of the input language of these model checkers. Each of the input languages uses different notations and symbols. Each of them also uses different means to provide the listed mechanisms. Since each of the model checkers are based on different temporal logic, the specification of the properties must also be stated by using different formulae. The output from the model checkers are also given in different forms.

Since model checkers are developed for different purposes, one model checker may be better than others for model checking a specific system. In this particular example, we found that Promela is better for describing a traffic light system, although it has a limitation in term of the proctype function and our codes is too length. Thus, it is important for users to choose the right model checker for modeling and verifying a system.

REFERENCES

Bengtsson, J. *et al*., 1995. UPPAAL-a tool suite for automatic verification of real-time systems. Proceeding of the 4th DIMACS Workshop on verification and control of Hybrid System, Oct. 22-24, Springer-Verlag New York, Inc. Secaucus, NJ, USA, pp: 232-243.

Berard, B., 2001. Systems and Software Verification: Model-checking Techniques and Tools. 1st Edn., Springer-Verlag, India, ISBN: 3540415238, pp: 190.

Bhaduri, P. and S. Ramesh, 2004. Model Checking of Statechart Models: Survey and Research Directions, http://arxiv.org/abs/cs.SE/0407038

Chandren, R.M., A.M. Zin and Z. Shukor, 2010. Model checking the biological model of membrane computing with probabilistic symbolic model checker by using two biological systems. J. Comput. Sci., 6: 669-678. DOI: 10.3844/jcssp.2010.669.678

Choi, Y., 2007. From NuSMV to SPIN: Experiences with model checking flight guidance systems. J. Formal Methods Syst. Design, 30: 199-216. DOI: 10.1007/s10703-006-0027-9.

Clarke, E.M., O. Grumberg and D. Peled, 1999. Model Checking, 1st Edn., The Mit Press, United States, ISBN-10: 0262032708, pp: 314.

Djavanroodi, F., M. Sabeghi and K. Abrinia, 2008. Analysis of the parameters affecting warping in radial forging process. Am. J. Applied Sci., 5: 1013-1018. DOI: 10.3844/ajassp.2008.1013.1018

El Emary, I.M.M. and A.I. Al Rabia, 2005. Fault detection of computer communication networks using an expert system. Am. J. Applied Sci., 2: 1407-1411. DOI: 10.3844/ajassp.2005.1407.1411

Holzmann, G.J., 2003. The SPIN Model Checker: Primer and Reference Manual. 1st Edn., Addison-Wesley, Germany, ISBN-10: 0321228626, pp: 608.

Islam, M.R., Z.E. Elshaikh, O.O. Khalifa, A.H.M.Z. Alam and S. Khan, 2010. Fade margin analysis due to duststorm based on visibility data measured in a desert. Am. J. Applied Sci., 7: 551-555. DOI: 10.3844/ajassp.2010.551.555

Jansen, D.N., J.P. Katoen, M. Oldenkamp, M.I.A. Stoelinga and I.S. Zapreev. 2008. How Fast and Fat Is Your Probabilistic Model Checker? An experimental performance comparison. Hardware Software: Verification Test., 4899: 69-85. DOI: 10.1007/978-3-540-77966-7_9

Jeffrey, J., P. Tsai and K. Xu. 2000. A comparative study of formal verification techniques for software architecture specifications. Annals Software Eng., 10: 207-223. DOI: 10.1023/A:1018960305057

Marta, K., 2003. Model Checking for probability and Time: from theory to practice. Proceeding 18th IEEE Symposium on Logic in Computer Science, June 22-25, Ottawa, Canada, pp: 351-360.

McMillan, K.L., 1999. Getting started with SMV, Cadence Berkeley Labs. http://www2.tcs.ifi.lmu.de/lehre/SS08/Automat/smv/doc/smv/tutorial/

Razali, R. and P. Garratt, 2010. Usability requirements of formal verification tools: A survey. J. Comput. Sci., 6: 1189-1198. DOI: 10.3844/jcssp.2010.1189.1198.

Trentesaux, D., 2009. Distributed control of production systems. Eng. Appl. Artificial Intell., 22: 971-978. DOI: 10.1016/j.engappai.2009.05.001