Journal of Computer Science 8 (1): 99-106, 2012
ISSN 1549-3636

Common Modeling Language for Model Checkers

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

Abstract: Problem statement: There are many different model checkers that have been developed. Each of the model checkers is based on different input languages and they are suitable for model checking different types of systems. Thus it is important for us to choose the right model checker or modeling and verifying a given system. However, moving from one model checker to another is not an easy task since we have to deal with different input languages. **Approach:** In order to solve the problem we propose a common modeling language that is based on UML state chart. Some translation rules for translating the model described in the common modeling language into the input languages of model checkers are also presented. **Results:** The result of the case study shows that our approach has been successfully applied in modeling the control system through the process of transformation and translation. **Conclusion:** Common modeling language can be used as a front end to help users to properly model a system before it is translated into input language of model checkers.

Key words: Model checking, UML state chart, Computational Tree Logic (CTL), Linear Temporal Logic (LTL), Probabilistic Computation Tree Logic (PCTL)

INTRODUCTION

Model checking is an automatic technique for checking properties of software and hardware systems (Clarke, 1997; Berard *et al*., 2010). There are several steps in using model checker. The first step is to specify the properties of the system to be checked. These properties are written in the form of temporal logic statements. The second step is to construct a formal model by using the input language of the model checker. The verification process is then carried out by the model checker. Once verification process is completed, the system will produce either true if model satisfied the property, or false if it does not. Most model checkers will also produce a counterexample if the property is not satisfied by the model. This counterexample is a state sequence that violates the model of the system. This implies that, a model checker will check whether a model satisfies a given property by exploring all possible behaviors of the system.

There are many model checking systems that have been developed, for SMV (McMillan, 1999), UPPAAL (Bengtsson *et al*., 1996), SPIN (Holzmann, 2004) and PRISM (Marta, 2003). All model checkers have their own input languages for modeling the system and for describing properties of system. The SMV language, for example, is based on 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). UPPAAL provides system validation and verification of real-time system. Systems to be verified can 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).

Model checkers are developed for modeling different types of systems. Therefore, one model checker may be more suitable for modeling a certain type of system compared to other model checkers. For example, SPIN is more suitable for modeling and verifying distributed systems, while PRISM is

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

specifically designed for probabilistic systems. Thus, it is important for users to choose the right model checker for modeling and verifying a specific system.

However, moving from one model checker to another is not an easy task. We need to translate the formal model of a system into the language of the model checker that we are using. Our previous study (Pathiah and Zin, 2010; Pathiah and Zin, 2011) shows that translating a formal model into the language of a model checker can be very challenging. First and foremost, we need to understand different types of notation and symbols of the input languages. Formalizing the properties is also difficult since different model checkers use different types of temporal logic.

In order to solve the problem, we propose a Common Modeling language (CM) for model checkers. A method for translating a model described in CM into a specific model checker is also proposed. Therefore, a user can describe the model of the system by using CM. He can then use the proposed translation method to translate the CM model into the language of the selected model checker. Thus, by using this approach a user can use the most suitable model checker for modeling and verifying a system. Since statechart diagram is considered to be one of the most popular graphical representations of a system, our proposed CM is based on the statechart diagram.

Related works: A few studies related to model checking and statechart are available. One of the study focuses on translating UML statechart to Extended Hierarchical Automata or EHA, (Dong *et al*., 2001; Sara *et al*., 2007) EHA which was proposed by Holzmann (2004) is actively used as an intermediate format to map the UML statechart to finite state machine system such as Labeled Transition System (LTS), Kripke Structure and many others. The main theoretical work behind the EHA is its formal operational semantics which precisely describe the semantics of UML statechart. The advantage of EHA is that its operational semantics is formally described to avoid misinterpretation.

Another related study is about translating π-calculus to the input languages of model checkers (Latella and Massink, 2001; Lam and Padget, 2004). π-calculus plays an important role as an intermediate representation between UML statechart and the input languages of model checkers. One of the advantages of this representation is that its formulizations are capable of providing various types of well-defined behavioral equivalences of statechart diagram.

MATERIALS AND METHODS

Description of statechart: A statechart diagram is a graphical state based notation for representing a system. Using a statechart for modeling a system can reduce the number of states needed to represent a system and therefore make it easier to be understood.

Many variations of statecharts are available and one of the most popular is the UML statechart. A UML statechart (Von der Beeck, 2001; Harel, 1987) is a complete graphical characterization of all the desired behaviors of an object during its lifecycle. To be more precise, a statechart conveys how an object behaves through time as a result of its reaction to events from the rest of the universe.

A UML statechart consists of states and transitions. A state describes a situation where an object satisfies some condition, performs some activities, or waits for some events. States can be classified as follows:

- A simple state is a state not composed of any sub states
- An OR state is composed to AND/OR states. If OR state is active, only one of its sub states is active
- An AND state is composed of several concurrent regions such as OR states graphically separated by dotted lines. If an AND state is active, all its regions are active
- The root state is state at the outermost level of the statecraft diagram, but is always drawn explicitly

An active configuration is a maximal collection of active states. A transition is purposely to specify when and to which states the object can change. A simple transition indicates that the system may change its state and perform a sequence of actions when a specified event occurs and a specified guard condition is satisfied. Such transition represents a direct relationship between a source state vertex and a target state vertex.

Proposed method: We propose a common modeling language that is based on basic features of UML statechart. Since a system can normally be decomposed into a number of hierarchical sub-systems, our proposed common modeling language is obtained by extending the statechart to include hierarchical and inter-relation between state hierarchies.

Definition 1: Formally, common modeling language is defined as:

$$CM = <S, S_0, Sc, G, T, L, R, Root>$$

Where:

S = finite set of states, where each state, s is declared as one of the two state types: {AND, OR}

S_0 = Set of initial states ($S_0 \subseteq S$). S_0 forms a valid initial transition relation
S_c = Set of states that forms a valid state configuration
G = Finite set of triggers
T = Finite set of transition relation, T = S×G×S'
L = S→S is the state level function. If s'⊆ L(s), then s` is an immediate descendant of s. The function of L describes a hierarchical state of the model
R = Relation between one levels to another

There are two types of CM states: AND and OR. AND state is a state that is used for modeling the concurrency by composing several simultaneously active sub CM. The AND state is a parent state to sub CM state machine that are concurrently active. The sub CM may interact with each other via triggers which are generated by other active components of the CM. The descendants of an AND state must always be OR states. An OR state is a state that supports one of state inside another state to provide hierarchy in the model. An OR state has sub-states that are related to each other by an exclusive-OR relationship. The leaf states of a CM must always be OR states. A CM, at any time, may have multiple active states which are known as a state configuration. A state configuration always contains one sub-state for each OR state and all sub-states for each AND state.

Consider the statechart model in Fig. 1. The CM hierarchical structure graph of statechart model in Fig. 1 is shown in Fig. 2.

The operation of a CM is described by using step semantics. The state configuration of CM always starts with an AND state. If the current state configuration contains more than one AND state, all of sub CM of the AND states will take place at the same time. For example, the state configuration for the Fig. 2 might be (B, F, H) or (B, G, I). In CM, a transition will always occur at each step in each active state configuration. If no explicitly modeled transitions are enabled, then an implicit transition will be fired. The synchronization of CMs allows it to be "flattened" into sequential automata preserving the model semantics. Each of single flatten CM is equivalent to a finite automata.

The single flatten component of the hierarchy structure graph is defined as CMs = <S, S_o,G,T> where S is a finite set of state, S_o is initial state, G is a finite set of triggers and T is the transitions. For example, CMs for Fig. 2 are:

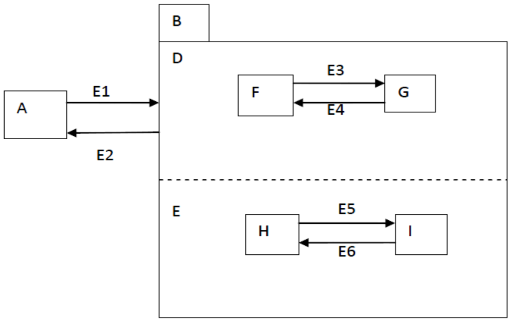

Fig. 1: Statechart model

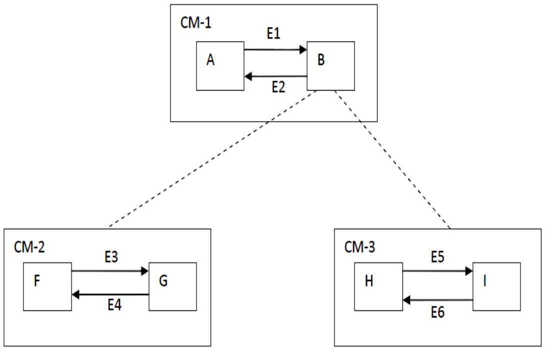

Fig. 2: Example of common modeling language hierarchy structure graph

A(CM-1) = ({A,B},B,{E1,E2},{(A,E1,B),(B,E2,A)})
A(CM-2) = ({F,G},F,{E3,E4},{(F,E3,G),(G,E4,F)})
A(CM-3) = ({H,I},H,{E5,E6},{(H, E5,I),(I,E6,H)}).

An active CM interacts through events. An event may trigger a transition to occur in synchronous components of the system in the following step. If an event trigger a transition from a state, s and the result of the transition is sub CM then the state, s is called as super state. As an example, B is super state of CM-2 and CM-3. These situations create inter-level transition. Inter-level transition cross state hierarchy boundaries. If a transition is leaving a super state, s, then the firing of all transitions contained the sub hierarchy of s is suppressed which is represented as dotted line in Fig. 2. This creates the relation, R between super state s and sub hierarchy and vice versa depending on the message received. At the same time, the level, L is created. The number given to the level is based on the priority leaving the super state. The example of level, L are CM-1, CM-2 and CM-3 where CM-1 is top level and CM-2 and CM-3 is sub level of CM-1. The Relation, R between levels is said as follows:

CM-1: receive message from CM-2 and CM-3
CM-2: receive message from CM-1
CM-3: receive message from CM-1

Translation from CM to I-SMV: To describe the translation of CM to input language of SMV, we first need to define I-SMV, the SMV input language.

Definition 2: Let I-SMV be the input language of SMV that consists of four tuples:

<M, V, N, Y>

Where:
M = Set of finite modules
V = Set of finite state variables
N = Set of next states
Y = Relation between one module to another

As the input language for SMV, I-SMV, is modular. The high level module is called as a module main. The other modules are called sub modules. In a module, M, there are state variables, V to describe the module. A state evolves from one state to another through a next operator, N. The relation, Y, between one module to another is described by using a set of parameter. The translation from CM to I-SMV is represented by a set of rules.

Rules from CM to I-SMV: In I-SMV, the level, L of CM corresponds to the module, M. The set of states, S and triggers, G correspond to the state variables, V. The transition, T corresponds to the next state, N. Lastly, the relation between levels, R corresponds to the relation between modules, Y. In this study, there are four rules for mapping CM to I-SMV. The rules are defined as follows:

Rule 1(module): Let Lev is the set of levels in CM. Each $Lev_i \in$ Lev is modeled as module declaration in I-SMV as follows:

Module $Lev_i(arg_i,\dots,arg_{i+1})$

If $Lev_i \in$ Lev does not exist, then the execution must be terminated. In I-SMV, arg_i is reference to the actual parameter of a module in the main module.

Rule 2 (Variable): Let St be the set of states and Gr is the set of triggers in CM. $St_i \in$ St is declared inside a module as follows:

$St_i : s_1..s_{n+1}$; if St is integer type
St_i: $\{s_1,..,s_n\}$; if St is enumerated type
St_i: {}; if St is boolean type

$Gr_i \in$ Gr is declared inside a module as follows:

$Gr_i : g_1..g_{n+1}$; if Gr is integer type
Gr_i: $\{g_1,..,g_n\}$; if Gr is enumerated type
Gr_i: {}; if Gr is boolean type

Rule 2 is used, if and only if the represented module exists and either St≠{} or Gr≠{}.

Rule 3 (state change): Let Tr be the set of transitions. In CM, the state changes might occur with or without a trigger, Gr. This implies that the state changes is between the source state, Ss and target state, St with or without trigger. The state changes in I-SMV are defined as follows:

```
next (St):=
  case{
     Tr_i: St; if gr ∈ Gr, Gr ≠ {}
     Tr_j: Ss; if gr ∈ Gr, Gr = {}
    default: St;
  };
```

The first statement defines the state changes caused by triggered transitions while the second statement defines the state changes caused by null-triggered transitions.

Rule 4 (Relation between modules): Let R_a, $R_{b,}$ be state variables for Lev_a and Lev_b. Let R_c and R_d be state variables for Lev_{cg}. The relation between those levels is defined as follows:

```
Module main()
St-Lev_c: Lev_c(St-Lev_a .R_a, St-Lev_b .R_b);
St-Lev_a: Lev_a(St-Lev_c.R_c);
St-Lev_b: Lev_b(St-Lev_c.R_d);
```

St-Lev_c , St-Lev_a , St-Lev_b are state variables in the main module. In I-SMV the arguments to a module is defined by state variable of destination message followed by state variable of source destination message.

Translation from CM to I-PRISM: The translation process follows the same approach as SMV. First, we need to define I-PRISM, the input language for PRISM.

Definition 3: I-PRISM is the input language of PRISM, which consists of four tuples:

<P, Q, H, C>

Where:
P = Set of finite modules,
Q = Set of finite state variables
H = Set of commands
C = Relation between one module to another

I-PRISM is also modular. However, a module, P is not allowed to produce a sub-module. Therefore, all of the modules are in the same level of hierarchy. In a module, there are state variables, Q which is used to model a module of a system. The state changes are described by a set of commands, H. The relation, C from one module to another is stated by synchronizer called as system .. endsystem construct.

Rules from CM to I-PRISM: In I-PRISM, the level, L of CM corresponds to the module, Lv. The set of states, S and triggers, G corresponds to the state variables, Vr and Er. The transition, T corresponds to the command, Sr. Lastly, the relation between levels, R corresponds to the relation between modules, Lv_i. The proposed rules for the mapping are as follows:

Rule 5 (module..endmodule): Let Lv be the set of level, L in CM. Each $Lv_i \in Lv$ is modeled as module declaration as follows:

```
module Lv_i
….
endmodule
```

If $Lv_i \in Lv$ does not exist, the translation process is terminated.

Rule 6 (state variable): Let Vr be the set of states, S and Er is the set of triggers G in CM. Both of $Vr_i \in Vr$ and $Er_i \in Er$ are declared within a module as variable declaration as follows:

Vr_i : [0..n+1] init 0; Vr is integer type
Er_i : [0..n+1] init 0; Er is integer type

In our case, n is an integer value starting from 0 to a large prime number. In I-PRISM, init is the initialization which automatically assigns the value of variables to 0.

Rule 7 (commands): Let Sr be the set of transitions T. In CM, the state changes might occur with or without a trigger ir. This implies that, the state change is between source state, ss and target state, st with or without trigger. In I-PRISM, the state changes are represented as a command, Sr which is defined as follows:

Sr is either:
  [] Vr_i = ss and Er_i=ir : Vr_i`=st; if $Er_i \neq 0$
or:
  [] Vr_i = ss: Vr_i`=st; if Er_i =0

The first statement defines the state changes caused by triggered transitions while the second statement defines the state changes caused by null-triggered transitions.

Rule 8 (synchronization): Let Lv_a, Lv_b, Lv_c are levels in CM.. Suppose Lv_b and Lv_c have relation, R with Lv_a, then we said both of Lv_b and Lv_c can be synchronized with Lv_a. In I-PRISM, the synchronization of Lv_a, Lv_b, Lv_c are defined as follows:

```
system
    ((Lv_b ||| Lv_c) || Lv_a)
endsystem
```

The symbol ||| implies that Lv_b and Lv_c are alternately synchronized with Lv_a. We assume synchronization is the same as relation between modules in SMV.

RESULTS

The translation from CM to input language of model checkers starts by modeling the behavior of a system into a statechart diagram. In this example, an elevator system which was previously modeled and verified using four model checkers is chosen for modeling and translation process. Fig. 3 shows the UML statechart diagram for an elevator system.

As shown in Fig. 3, there is only one AND state (On1) and two OR state (Open/Close and Level1/Level2). There are only two basic states (On2 and free) and this model has only two regions, the first one is shown as dotted line in On1 state and the other is the outermost. The flatten CM of the elevator system is shown in Fig. 4.

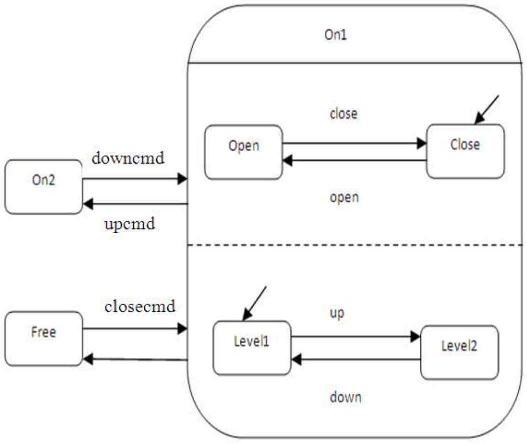

Fig. 3: Statechart model of elevator system

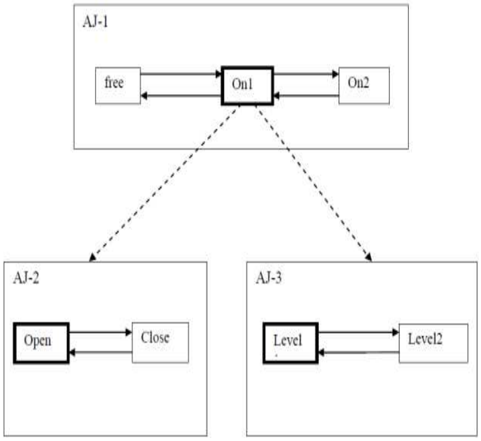

Fig. 4: The flatten CM of the elevator system

Based on the Fig. 4, each flatten CMs=(S, S_0, G, T) is obtained as follows:

```
CMs(AJ-1) = ({free, On1,On2},On1,
   {opencmd,closecmd,upcmd,downcmd},
   {(On1,opencmd,free),(free,closecmd,On1),
   (On1,upcmd,On2), (On2,downcmd,On1)})

CMs(AJ-2)=({Open,Close},Open,{open,close},
   {(Open,close,Close),(Close,open,Open)})

CMs(AJ-3) = ({Level1,Level2},Level1,{up,down},
   {(Level1,up,Level1),(Level2,down,Level1)})
```

The relation, R between CMs is obtained as follows: For:

AJ-1: receive message from AJ-2 and AJ-3
AJ-2: receive message from AJ-1
AJ-3: receive message from AJ-1

We use the rules for translating the CM of the elevator system into the input language of SMV and PRISM.

From CM to SMV: By using Rule 1, each level in CM is translated into a module in SMV input language. Each module name is followed by a list of arguments, arg_i in I-SMV. So in this example, there are three modules which are stated as AJ-1, AJ-2 and AJ-3. The arguments are created based on relation, R in Rule 4 of the modules.

```
Module AJ-2(aj-1)
Module AJ-3(aj-1)
Module AJ-1(aj-2, aj-3)
```

Rule 2 is regarding the variables declared in each module. All the states, S and triggers, Gr in CM are translated as follow:

```
Module AJ-2(aj-1)
VAR
   state: {Open, Close};
   trigger2: {close, open};

Module AJ-3(aj-1)
VAR
   state: {Level1, Level2};
   trigger3: {up, down};

Module AJ-1(aj-2, aj-3)
VAR
    state: {On1, free, On2};
    trigger5: {closecmd, opencmd};
    trigger6: {downcmd, upcmd};
```

The transition, T which is stated in Rule 3 is translated to next state in SMV. In SMV, transition for each level is written as follow:

```
Module AJ-3(aj-1)
…..
next (state) := case
  ((state = level 2) and (aj_1.trigger6 = downcmd)):
        level1;
  ((state = level1) and (aj_1.trigger6 = upcmd)):
        level2;
  1: state;
esac;

Module AJ-2(aj-1)
……
next (state) := case
   ((state = open) and (aj_1.trigger5 = closecmd)):
          Close;
    ((state = Close) and (aj_1.trigger5 = opencmd)):
          Open;
   1: state;
esac;

Module AJ_1(aj_2, aj_3):
…….
next (state) := case
   (((state = free) and (trigger5 = closecmd)) |
   ((state = On2) and (trigger6 = downcmd))): On1;
   ((state = On1) and (trigger5 = opencmd)) : free;
    ((state = On1) and (trigger6 = upcmd)) : On2;
  1: state;
esac;
```

Relation between levels which is stated in Rule 4 is translated into module main. Based on Rule 4, the state variables of Module main are defined as a call module for corresponding level. The argument in each call module is state variable of source state followed by its message passing. The following codes describe relation between levels:

```
Module main:
VAR
   Aj-2: AJ-2 (aj-1)
   Aj-3: AJ-3 (aj-1)
   Aj-1: AJ-1 (aj-2, aj-3)
```

Based on the above codes, aj-1, aj-2 and aj-3 are state variables and used as arguments modules AJ-1, AJ-2 and AJ-3.

From CM to PRISM: We use Rule 5 to Rule 8 to translate CM to the input language of PRISM. By using Rule 5, level, L in CM is mapped to modules in PRISM. Modules in PRISM are coded below:

```
module AJ-2
…..
endmodule

module AJ-3
….
endmodule

module AJ-1
………….
endmodule
```

Rule 6 is about state variable. Similar to SMV, states and triggers are mapped to state variables in PRISM. The state variable for each module is coded below:

```
Module AJ_2
AJ_2_state:[0..1] init Open;
AJ_2_trigger3:[0..1] init close;
….
endmodule

Module AJ_3
AJ_3_state:[0..1] init level1;
AJ_3_trigger3:[0..1] init up;
….
endmodule

Module AJ_1:
AJ_1_state:[0..2] init on1;
AJ_1_trigger5:[0..1] init closecmd;
AJ_1_trigger6:[0..1] init downcmd;
………….
endmodule
```

Rule 7 is about a command. Each transition is mapped to a command in the PRISM input language. The commands in each module are coded below:

```
Module AJ_2:
….
[] (AJ_2_state = Open ) -> (AJ_2_state' = Close );
[] (AJ_2_trigger3=close )->(AJ_2_state' =Close );
[] (AJ_2_state = Close ) ->(AJ_2_state' = Open );
[] (AJ_2_trigger3=open )->(AJ_2_state' = Open );
endmodule

Module AJ-3:
….
[](AJ_3_state= Level1 ) -> (AJ_3_state' =Level2 );
[](AJ_3_trigger3= up ) -> (AJ_3_state' = Level2 );
[]( AJ_3_state = Level2)-> (AJ_3_state' =Level1 );
[](AJ_3_trigger3=down )->(AJ_3_state' = Level1 );
endmodule

Module AJ_1:
………
[](AJ_1_state = free)->(AJ_1_state' = On1 );
[](AJ_1_trigger5=closecmd)->(AJ_1_state'=On1 );
[](AJ_1_state = On1) -> (AJ_1_state' = free );
[](AJ_1_trigger5=opencmd)->(AJ_1_state' =free );
[](AJ_1_state = On2)->(AJ_1_state' = On1);
[](AJ_1_trigger6=downcmd)->(AJ_1_state'=On1);
[](AJ_1_state =On1)->(AJ_1_state' = On2);
[](AJ_1_trigger6 = upcmd )->(AJ_1_state' = On2);
endmodule
```

Based on the above codes, the commands are executed with or without triggers.

Lastly is the synchronization between modules. Based on Rule 8, relation between levels of hierarchy is used to map the synchronization in the PRISM input language. Since AJ-2 and AJ-3 is alternately synchronized with AJ-1, the synchronization of modules is coded below:

```
system
   ((AJ_2 ||| AJ_3) || AJ_1)
endsystem
```

DISCUSSION

The study describes the proposed common

modeling language. CM for model checkers together with a set of rules for translating from the language into the input language of two model checkers SMV and PRISM. The result of of the case study shows that the proposed method for translating the model described in CM into the input languages of SMV and PRISM is feasible.

One of the property of the system can be described as follows: When the cabin arrives at the requested floor, the door is opened.

We have model checked the SMV codes obtained from the translation process together with the above property by using SMV model checker. The verification result shows that the above property is satisfied TRUE. The number of BDD nodes is 74 and model checking time is 0.015 second whereas verification time by the system is 0.01 second. The number of transition relation is 12. A similar verification process was done for PRISM.

CONCLUSION

This study describes a common modeling language for model checkers. This language relies on UML statechart features with some extension such as the state hierarchy and inter-relation transition. We have also proposed translation rules for mapping the model described in the common modeling language into the input language of SMV and PRISM model checkers. The feasibility of the proposed method has been demonstrated by using a case study.

Currently, we are in the process of carrying out two more activities in order to enhance our proposed method. The first activity is to expand the proposed method to include two more model checkers: SPIN and UPPAAL. The second activity is to develop a software tool that can help users to translate the model described in the common modeling language into the input languages of model checkers.

REFERENCES

Bengtsson, J., K. Larsen, F. Larsson, P. Pettersson and W. Yi, 1996. UPPAAL- A tool suite for automatic verification of real-time systems. Hybrid Syst. III, 1066: 232-243. DOI: 10.1007/BFb0020949

Berard, B., M. Bidoit, A. Finkel and F. Laroussinie *et al*., 2010. Systems and Software Verification: Model-checking Techniques and Tools. 1st Edn., Springer Publishing Company, Incorporated, ISBN: 3642074782, pp: 196.

Clarke, E.M., 1997. Model Checking. Lec. Notes Comput. Sci., 1346: 54-56. DOI: 10.1007/BFb0058022

Dong, W., J. Wang, X. Qi and Z. Qi, 2001. Model Checking UML Statechart. IEEE Proceeding of the 8th Asia-Pacific Software Engineering Conference, Dec. 4-7, IEEE Computre Society, China, pp: 363. ISBN: 0-7695-1408-1

Harel, D., 1987. Statecharts: A visual formalism for complex system. J. Sci. Compt. Program., 8: 231-273. DOI:10.1016/0167-6423(87)90035-9

Holzmann, G.J., 2004. The SPIN Model Checker: Primer and Reference Manual. 1st Edn., Addison-Wesley, Boston, ISBN: 0321228626, pp: 596.

Lam, V.S.W. and J. Padget, 2004. Symbolic model checking of UML statechart diagrams with an integrated approach. Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based System, May 24-27, IEEE Xploor, USA., pp: 337-346. DOI: 10.1109/ECBS.2004.1316717

Latella, D. and M. Massink, 2001. A formal testing framework for UML statechart diagrams behaviours: From theory to automatic verification. Proceedings of the 6th IEEE International Symposium on High-Assurance System Engineering Symposium, (HISES' 01), IEEE Xploor, Boco Raton, pp: 11-12. DOI: 10.1109/HASE.2001.966803

Marta, K., 2003. Model checking for probability and Time: From theory to practice. Proceedings 18th IEEE Symposium on Logic in Computer Science, June 22-25, IEEE Computer Society, Ottawa, Canada, pp: 351. ISBN: 0-7695-1884-2

McMillan, K.L., 1999. Getting Started with SMV. 1st Edn. Cadence Berkeley Labs, USA., pp: 1-90.

Sara, V.L. and H. Albert, 2007. SV$_t$ L: System verification through logic tool support for verifying sliced hierarchical statecharts. LNCS, 4409: 142-155. DOI: 10.1007/978-3-540-71998^{-}4-9

Von der Beeck, M., 2001. Formalization of UML-statecharts. Lec. Notes Comput. Sci., 2185: 406-421. DOI: 10.1007/3-540-45441-1_30