CMGT: Support Tool for Using Model Checking

Pathiah Abdul Samat and Abdullah Mohd Zin

Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, 43300 Serdang, Selangor, Malaysia.
Faculty of Technology and Information Science, Universiti Kebangsaan Malaysia 43600 Bangi, Selangor, Malaysia.

Abstract: Model checking is now considered to be a popular approach for verifying software and hardware systems. Since different model checkers are designed to support different types of systems, they and their input languages support different features. So the problem arises; how to choose a suitable model checker for a certain system? One approach is to provide a tool to guide the use of model checkers. In this study, we propose an interface to enable the user to model a system. Then its output is translated and input directly to a model checker. The correctness of the common model is checked with two case studies: the tool is applied successfully to SMV and PRISM. It is found that the targeted case studies are implemented successfully without any barrier especially in the modeling steps; transformation and translation. Our approach can also be used to other model checkers such as SPIN and UPPAAL as long as the system to be modeled is restricted to statechart diagram.

Key words: Model checking, UML statechart, input language, translation, transformation.

INTRODUCTION

Model checking (Clarke, E.M., 1999; Berard, B., 1999) is now considered to be a popular approach for verifying software and hardware systems. A model checker accepts system requirements or designs (called models) and a property (called a specification) that the final system is expected to satisfy. The tool then produces an output “yes” if the given model satisfies given specifications; otherwise, it generates a counter-example with error traces. A large variety of model checkers are currently available; including the popular SMV (McMillan, K.L., 1999), SPIN (Bengtsson, J., et al., 1995), PRISM (Holzmann, G.J., 2004) and UPPAAL (Marta, K., 2003). Since model checkers are developed for different purposes, one model checker may be more suitable for modeling a certain type of system than another. For example, our previous study (Pathiah, A.S, 2011) shows that SPIN is the best of four common model checkers for modeling a traffic light system. Thus, it is important for users to choose a suitable model checker for modeling and verifying a specific system. Model checkers come in packages with their own notations and facilities. Our previous study (Pathiah, A.S, 2011) also shows that although there are a number of similarities between the notations used; there exist some significant differences that may cause some difficulties when a user moves from one model checker to another. In order to solve that problem, we have proposed the use of a common modeling language (Pathiah, A.S and Abdullah, M.Z. 2012) based on state machines approach. In this paper we describe a software tool that can be used to help a user to describe models in the input notations used by a number of model checkers. The main purpose of this tool is to help users to express a model in the input language of the selected model checker. Several studies relate tool supports of model checker are available. One toolset (Patrick, O.B., 2003) accepts XMI as intermediate format and methods; parsing and transformation are applied embodying the behavioral semantics and properties of statechart elements. The tool is developed based on combination technologies: UML statecharts, XMI, database, model checking and simulation. This tool supports only input language of SPIN. The same technologies have been applied (Marco, B., 2007) to develop a support tool for transforming web models into formal models. A support tool (Yeung, W.L., 2007) has been developed for mapping statechart diagrams to Communicating Sequential Processes (CSP) together with FDR2 model checking. The third tool supports the verification of sliced hierarchical statecharts (Sara, V.L. and Albert, H. 2007) and is known as SVL. SVL is designed to retrieve Extended Hierarchical Automata (EHA) operational semantics and translates the EHA semantics to CaSMV temporal logic formula. The advantage of this tool is its ability to visualize the error trace produced by CaSMV. Most of the above studies focus on only a single model checking system. This implies that, the tool developed can be used for only a specific model checker. In the work presented in this paper, we describe an approach and tool to support a number of model checkers.

MATERIALS AND METHODS

A state machine can be represented as a state transition system which in turn can be described graphically

Corresponding Author: Pathiah Abdul Samat, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, 43300 Serdang, Selangor, Malaysia.
E-mail: pathiah@fsktm.upm.edu.my;
by a state transition diagram (STD). Each state is represented as a circle and each transition as an arrow. An incoming arrow without origin identifies the initial state. Such graphical representations are particularly useful for automata-based formalisms, and so provide invaluable support for our understanding of a system’s operation. Formally, a finite state machine (FSM) is a 4-tuple \((Q,E,T,q_0)\) in which:

- \(Q\) is a finite set of states;
- \(E\) is the finite set of transition labels;
- \(T \subseteq Q \times E \times Q\) is the set of transitions;
- \(q_0\) is the initial state.

However in a real problem, a system consists of several components configured hierarchically and concurrently, and communicating with each other. As a result, the system operation is hard to capture and certainly difficult to understand using FSMs. In order to solve this problem Harel[13] proposed the concept of statecharts to extend FSMs. Statecharts are visual diagrams rich in graphical state based notations. Using statecharts to model a system can reduce the number of states needed to graphically represent system behavior and make the model much easier to comprehend. Many variations of statecharts exist including UML statechart.

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 component 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.

In this particular statechart, a hierarchical automaton is associated with each object (class instance) to model its behavior. Many software tools have been developed to support UML statecharts. One of them is Altova [14] that support the generation of a XMI format file.

The proposed common modeling language applies UML statecharts and extends the hierarchical and inter-relation between state hierarchies which are not supported by the original statechart. Its formal definition is as follows:

**Definition**: Our common modeling language is defined to be \(CM=(S, S_0, S_c, G, T, L, R)\) where:

- \(S\) is a finite set of states, where each state, \(s\) is declared as one of the two state types: \{AND, OR\}
- \(S_0\) is a set of initial states \((S_0 \subseteq S)\), \(S_0\) forms a valid initial transition relation.
- \(S_c\) is a set of states that forms a valid state configuration.
- \(G\) is a finite set of triggers, \((g \subseteq G)\)
- \(T\) is a finite set of transition relation, \(T = S \times G \times S'\)
- \(L:S \rightarrow S\) is the component. If \(s' \subseteq L(s)\), then \(s'\) is an immediate descendant of \(s\). The component, \(L\) describes a hierarchical state of the model.
- \(R\) is a relation between one component 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 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, and then Fig. 2 is the CM hierarchical structure graph of statechart model in Fig. 1.
The operation of a CM is described by using step semantics. The state configuration of CM always start 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 the single flatten CM is equivalent to a sequential automata.

A sequential automata [15] is defined as $A = <\sigma_A, s_0^A, \lambda_A, \delta_A>$ where $\sigma_A$ is a finite set of states, $s_0^A$ is the initial state, $\lambda_A$ is a finite set of labels, and $\delta_A \subseteq \sigma_A \times \lambda_A \times \sigma_A$ is the transition relation. The single flatten CM is defined as $CMs = <S, S_0, G, T>$ where $S$ is a finite set of state, $S_0$ is an initial state, $G$ is a finite set of triggers and $T$ is the transitions. For example, the sequential automata, $A$ or single flatten, CMs for Fig. 5.2 are,

$A(CM-1) = (\{A,B\},B,\{E1,E2\},\{(A,E1,B),(B,E2,A)\})$
$A(CM-2) = (\{F,G\},F,\{E3,E4\},\{(F,E3,G),(G,E4,F)\})$

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 superstate. As an example, $B$ is superstate of CM-2 and CM-3. These situations create an inter-level transition. Inter-level transition cross the state hierarchy boundaries. If a transition leaving a superstate, $s$, then the firing of all transitions contained the sub hierarchy of $s$ is suppressed which is represented by the dotted line in Fig. 2. This creates the relation, $R$ between superstate $s$ and sub hierarchy and vice versa depending on the message received. At the same time, the component, $L$ is created. The number given to the component is based on the priority leaving the superstate. The examples of component,
L are CM-1, CM-2 and CM-3 where CM-1 is top component and CM-2 and CM-3 is sub component of CM-1. The Relation, R between components is said as follows:

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

Common Modeling and Guided Translation (CMGT) is a support tool based on CM to assist users to use model checkers. The overall concept used in the design and implementation of CMGT is shown in Fig. 3.

![Fig. 3: Implementation of CMGT.](image)

The whole process involves four phases as follows: (1) A user describes the informal model of the system by using UML statecharts; (2) CMGT then transforms the UML statechart into XMI format. This is done by using Altova; (3) CMGT then processes the XMI file by extracting the required tag names of XMI to transform them to a representation in CM; (4) Then, CMGT guides users to translate the model from the CM to the input language of the selected model checker.

**Translating into XMI Format:**

The verification of a state machine starts by expressing its behavior using UML statechart. The UML statechart is translated into XMI format by using Altova. With the UML of version Altova, the diagram is saved in an XMI file.

**Processing XMI File:**

A program module reads the XMI file. A method like the DOM parser [16] is used to parse the required tag names into a set of tables. The tag names are <region>, <subvertex>, <transition> and <trigger> including the corresponding attributes. The program module extracts information such as Type, id, uuid, name, ParentId, component, source and target from the original statechart. The extracted information forms two different tables. Since a statechart may contain different types of composite state either OR or AND state, region and triggers, it is convenient to put all this information into one table named ExtrinsicObject. Fig. 4 shows the fields of the ExtrinsicObject table. We have layered each element of statechart into a number of components based on the CM definition. The element of statecharts can be grouped by their component.

![Fig. 4: Table ExtrinsicObject.](image)

A transition corresponds to a transition edge on the statechart diagram. It may contain items like trigger, source state, target state and etc. We group all this information into a Transition table. Fig. 5 shows the fields of Transition table.

![Fig. 5: Table Transition.](image)
The information about component is obtained by creating a special code based on the method HashMap which is provided in java.util.

Transformation into CM:
The elements from both tables are used for transforming the state chart into CM by using the following steps:
• Every element from the same component of both tables is grouped into hierarchy in CM.
• For every Type uml: state in ExtrinsicObject, a list of states is created in CM under the STATE.
• For every Type uml: trigger in ExtrinsicObject, lists of trigger are created in CM under the TRIGGER $n+1$.
• For every Type uml: transition in Transition with parent of uml:trigger equal to id of uml:transition, a set of transition is created in CM by combining source and trigger name with target. The set of transition is grouped under TRANSITION.
• For every component, if parent of uml: region equal to id of uml:state, a list of links is created in CM. The corresponding component is said to be the successor-component and its parent component is said to be the predecessor-component. The links can be formed into two types:
  o Successor-component: receive messages from the predecessor-component.
  o Predecessor-component: receive messages from successor-component, and successor-component $n+1$.

The component of the system is represented in the CM by extracting the data of elements for corresponding components as shown in Fig. 6.

```
SELECT COMPONENT FROM ExtrinsicObject, Transition
while (COMPONENT not finish) {
    <COMPONENT>:< printout COMPONENT >
    <STATE>: While (STATE of AJ-1 not finish) {
        printout STATE for this COMPONENT
    } //end-while STATE
    <TRIGGER>: While (TRIGGER of AJ-1 not finish) {
        printout TRIGGER for this COMPONENT
    } //end-while TRIGGER
    <TRANSITION>: while (transition of AJ-1 not finish) {
        if (PARENTID of uml:trigger == xml:id of uml:transition)
            then (get SOURCE of uml:transition ;
                        get TRIGGER of uml:trigger
                        get TARGET of uml:transition)
    } //end-while TRANSITION
} //end-while COMPONENT
```

Fig. 6: An algorithm of single common modeling in text form.

The relationship between various components is obtained by extracting elements as shown in Fig. 7. The relationship is important since it can be used to guide users in modeling the synchronization of system components. This relationship can also be used to identify whether the components should interact between one another through message passing or through shared variables.

```
Select FROM ExtrinsicObject, Transition
<REFINEMENT>: get PARENTID of uml Region and get xml:id of uml state:
Case (COMPONENT):
    AJ-3: <This module receive message from its parent>
    while (uml.Region not finish) {
        if (PARENTID of uml.Region == xml:id of uml.state)
            then printout (COMPONENT of uml.state)
    } //end-while
    AJ-2: <This module receive message from its parent>
    while (uml.Region not finish) {
        if (PARENTID of uml.Region == xml:id of uml.state)
            then printout (COMPONENT of uml.state)
    } //end-while
    AJ-1: <This module receive message from its children>
    while (uml:state not finish) {
        if (child of uml:state == xml:id of uml.Region)
            then printout (COMPONENT of uml.Region)
    } //end-while
//end-case
```

Fig. 7: An algorithm of relationship between common modeling.
Translating into the Input Language of Model Checkers:

The description about the system components and the relations between the components in CM are used for guiding users in the process of translating into the input language of a model checker. In the current implementation, we are providing translators for translating CM into two model checkers: SMV and PRISM. The implementation of CMGT translator is shown in Fig. 8.

From the figure above, a description of the system in CM is parsed by the parser to produce a list of objects. Each of the elements in this list is then mapped to SMV or PRISM by the translator.

Results:

In this section, we show the use of the tool on a case study of a CCTV system known as Digital Video Control and Monitoring System (DVCMS). Briefly, DVCMS [17] use sixteen high technology of personal computer under controlling and monitoring of CCTV. In addition, this system can also display and record video images from sixteen channels without missing the original images; each camera represents three processes to handling the transmission display, record and controller. The display is either displaying a static image or motion image or no signal. If the screen displays a static image, then the camera status is green, if the screen displays a motion image, then the camera status turns to red; and if there is no signal, the camera status turns to yellow. Only motion images are recorded. However, if it receives a static image continuously for more than 10 seconds, then the next record will be stored in a new file. Otherwise, the record is stored in the same file. This phenomenon is known as delay time. In this system, the property to be checked is whether the record will save motion images in a new file if the delay time is greater than 10 seconds.

Representation in Statechart:

The model of DVCMS as a statechart is shown in Fig. 9.

Fig. 8: The implementation of CMGT translator.

Fig. 9: The statechart model of DVCMS.
Translation into XMI:
The statechart described in Fig. 9 is then translated into XMI format by Altova.

Transformation into CM:
The elements of XMI are read and transformed into CM. The text form of CM is shown in Appendix A.

Translation into SMV:
Based on the information in CM, components are translated into a module’s name in SMV, coded as follows:

<table>
<thead>
<tr>
<th>Component in CML:</th>
<th>SMV:</th>
</tr>
</thead>
<tbody>
<tr>
<td>AJ-1</td>
<td>MODULE AJ1(AJ2,AJ3)</td>
</tr>
<tr>
<td>AJ-2</td>
<td>MODULE AJ2(AJ1)</td>
</tr>
<tr>
<td>AJ-3</td>
<td>MODULE AJ3(AJ1)</td>
</tr>
</tbody>
</table>

In SMV, the states, triggers and transitions of CM in each component are translated to state variables and next states. For example states, triggers and transitions in AJ-1 are translated as shown in Appendix A.

As coded above, component AJ-1 is automatically translated to MODULE aj1(AJ2,AJ3). The states and triggers of AJ-1 are translated to AJ1_state, AJ1_trg1, and AJ1_trg2 etc. The transitions for each variable in AJ-1 are translated to next states in SMV. However, the transitions appear CM due to the statechart diagram. In SMV, all the state variables must be declared as next relation. Due to that limitation, we provide functions to add and remove the next relation to other state variables in our tool. As a result, we could extend the next relation for all triggers stated in common modeling. In SMV, the extensions of the next relation is represented as next (AJ1_trg1), next (AJ1_trg2), next (AJ1_trg3), next (AJ1_trg4) and next (AJ1_trg5). The same approach is applied to others components such as AJ-2 and AJ-3.

The other important part in common modeling is the refinement component, which represents a hierarchy and acts to synchronize components. In SMV, the refinement component is translated to MODULE main. In order to synchronize the components of the system, we use shared variables between the modules by locating the shared variables in the parameter lists. In our tool, we provide a selection for choosing the shared variable on a pull down menu. The source code of module main is shown below:

Refinement in CML:       smv:
AJ-1:receive message    VAR
    from [AJ2, AJ3]       AJ1:aj1(AJ2,AJ3);
AJ-1:receive message    AJ2:aj2(AJ1);
    from [AJ2, AJ3]       AJ3:aj3(AJ1);
AJ-1:receive message    AJ4:aj4(AJ1);
    from [AJ2, AJ3]

We also provide typing assistance for formalizing the properties. By using the tool provided, the properties of interest are formalized in the same file as follows:

SPEC
AG((ctrl.delay>10)-> AF(rec.rec_state=New))

The result shows that the above property is satisfied. The number of BDD nodes is 4257 and model checking time is 0.015 seconds whereas verification time by the system is 0.03 second. Lastly, the number of reachable states is 17. The full source code of DVCMS in SMV language can be found in Appendix B. Fig. 10 shows a screenshot of the SMV translator.

Translation into PRISM:
Translation into PRISM can be done by using a similar approach. At the beginning, all of states and triggers are translated as constant integer which is defined as global in PRISM language. The source codes below show how states and trigger for each component are automatically translated to constant integer in PRISM language.
Fig. 10: A screenshot of SMV translator.

The components of common modeling are automatically translated to module’s name in PRISM language. The source codes below show how each component is mapped to module’s name including its state variables in PRISM.
The state variables are declared in each module of PRISM language. However, the initialization (init) is made based on the user selection from the pull down menu. In PRISM, each of state variables must be initialized.

The transition in common modeling is automatically translated to command in PRISM. In PRISM, each transition is automatically translated to command which consist of two parts; guard and modifier. Since transition appear in common modeling is limited to statechart diagram, the transition of the other state variables are provided by our tool. In this tool, a user can add new command based on the selection provided in the pull down menu. Based on the codes, a group of command under triggers is made based on user’s selection. The same approach applied for other component such as AJ-2 and AJ-3. The source codes below show how the transition in component AJ-1 corresponds to command in module aj1.

The refinement component in common modeling is automatically translated in PRISM as below:
As coded above, the `endsystems` construct is used to synchronize the components of a system. In this construct, module `aj2` and `aj3` are alternatively synchronized with module `aj1`. This construct also represents the hierarchical structure of modules. In this case, module `aj2` and `aj3` are sub-hierarchy of `aj1`. Similar to SMV, we provide assistance to specify the properties in PRISM. By using a type in assistance, the property of interest is formulated as follow:

\[ P \geq 1 \ [ \text{true} \ U \ \text{delay} > 10 \Rightarrow \text{recState} = \text{New} ] \]

Once model and property of PRISM is completed, the form mode can be changed to text mode by clicking a button on the top of the right hand side. Fig. 10 shows the screenshot of PRISM translator. Our tool will display the complete source codes in text for copy and paste to PRISM model checker. If the source code is free from error, the verification can be performed automatically. The verification result obtained indicates that the property is FALSE. The model checking time is 0.04 second and number of state satisfying is 80 including the initial time. The number of states and transitions are recorded as 88 and 278 respectively. However, the reachability is only 15 iterations. Fig. 11 shows a screenshot of the PRISM translator. Our tool displays the complete source code as the text for PRISM model checker. If the source code is free from error, the verification can be performed automatically.

Fig. 11: A screenshot of PRISM translator.

**Conclusion:**

This study has concentrated on the methods for reading XMI documents, building database tables and translation tools for generating SMV and PRISM languages from common modeling. We have used the integrated development environment (IDE), which comprises of Altova, XMI, Java Dom parser, SQL database, SMV and PRISM. The IDE provided the necessary tools for developing our parser including the SMV and PRISM input languages. Our approach is based on the UML statechart model and transforms the statechart model to a common modeling language which acts as intermediate format towards model checking. The purpose of common modeling is to flatten the statechart model to represent the hierarchical structure of the system.

From the above methods, we have developed a front-end tool called as CMGT for guide translating common modeling to SMV and PRISM. CMGT also supports synchronization, enabling the modeler to model check modules derived from common modeling. Our tool also provides functions such as add or remove next changes for SMV and commands for PRISM. The tool also provides facilities to initialize the state and this function can be made by the selection from a pull down menu. The others facility provided by CMGT is type assistance. This mechanism provides multiple choices of temporal operators, relational operators and other important symbols.

From the case study that has been carried out, we believe that our tools are able to transform UML statechart to a common modeling language. We also believe that the translation from common modeling language to input language of SMV and PRISM has been validated.
REFERENCES

Altova UML., Available at http://www.altova.com