

Translating UML State Machine Diagram into Promela

Panisara Damjan and Wiwat Vatanawood

Abstract— In this paper, the translation of UML state machine diagram along with the informative inscriptions into Promela code is considered. The translating rules are proposed to systematically map the elements of the state machine into the corresponding Promela block code. The main state machine notations which are states and pseudo states and their transitions, are focused. In addition, several inscriptions including state's local preconditions and postconditions, transition labels consisting of events, guards, and actions, on the state machine are considered to elaborate the minimal completeness of the resulting Promela code.

Index Terms— UML State Machine Diagrams, PROMELA, SPIN

I. INTRODUCTION

DURING the software design phase, the designers would intend to have their design models verified in order to ensure their correctness beforehand. Recently, a software design model is commonly drawn in terms of UML diagrams to cover both structural and behavioral properties of a software system. There are several researches and development of the formal verification tools for UML diagrams. [1] proposed a method to translate UML sequence diagrams using graph transformation rules into formal model. A tool called AToM was developed. [2] proposed a translation of a subset of UML statechart diagram into formal model and [3] proposed an approach which created a Promela-based model from UML interactions expressed in sequence diagrams. [4] proposed a tool for verifying a collaboration and state machine.

In this paper, we are more specific on the state machine diagram [5], [6] which is one of UML2 diagrams. It has been used for describing the dynamic property of a software system. Typically, it is common and convenient to describe the behavior of a single object or instance of a class, by specifying the sequence of states that an object goes through during its valid lifetime in software system. An object would remain in a certain state until any valid transition triggers so that the object could be changed to move into its next state. The mentioned valid transition is typically inscribed or labelled as events, guard conditions, and actions respectively. In our concern, these inscriptions were

Panisara Damjan is a M.Sc student at department of Computer engineering Faculty of Engineering, Chulalongkorn University, Bangkok, 10330 Thailand (e-mail: Wiparat.Do@student.chula.ac.th).

Wiwat Vatanawood is currently an Associate Professor of department of Computer Engineering, Faculty of Engineering, Chulalongkorn University. His research interests include Formal Specification, Formal Verification, Software Architecture (e-mail: wiwat@chula.ac.th).

frequently left out or covered by the previously proposed researches, however we will mention it later. To verify the UML state machine diagrams in design phase beforehand, helps and guides the designer a lot. There are several formal verification approaches proposed for UML diagrams, the UML state machine diagram in particular. Among these approaches, model checking is an alternative to do exhaustive verification of the software properties - correctness, liveness, and safety, etc. Many related model checking methodologies, modeling languages along with the automatic tools [4], [7], [8] were proposed so that the designer needs no any mathematical background knowledges. Although almost all formalization of the UML state machine diagram were proposed as mentioned earlier, there still are some informative inscriptions in the diagram ignored which include the preconditions, postconditions, invariants, and transition elements - events, guards, and actions, etc. In this paper, we propose an alternative to do the translation of the UML state machine diagram with its informative inscriptions into Promela code. The resulting Promela code would be automatically generated in the well-formed fashion which is reusable and maintainable.

This paper is organized as follows. Section II is the background and section III describes our translation approach. Section IV shows our demonstration and samples. The conclusion is in section V.

II. BACKGROUND

A. UML State Machine Diagram

A UML state machine diagram [5], [9] depicts the behavior of a particular object in the software system, specifying the sequence of events and the responses to the events of the object during its valid lifetime. A typical state machine diagram consists of a finite set of states, a finite set of pseudo states, and a finite set of transitions. However, in this paper, we focus only on five common elements of the diagram - initial state, final state, state, choice, and transition, as shown in Fig. 1 adapted from [5].

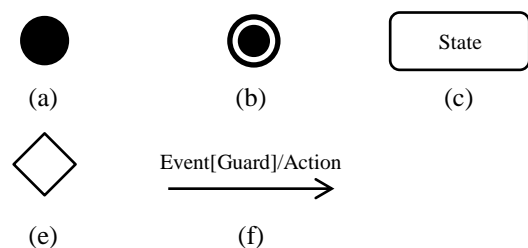


Fig. 1 Symbol of State machine diagram adapted from [5]

- Initial state (a):
Initial state represents the only starting point in the diagram. The outgoing transition from the initial state may have a behavior, but not a guard or trigger.
- Final state (b):
Final state represents the end of a sequence or activity in the diagram. The object finally ends its behaviors and no further responding activities. A diagram may have multiple final states.
- State (c):
State represents a set of object values, a period of time during an object performs local activities, a period of time during an object waits for the expected events to occur.
- Choice (e):
Choice is one of the pseudostates and represents a dynamic conditional branch by evaluating the mutually exclusive guard conditions of the triggers of its outgoing transitions.
- Transition (f):
Transition represents a directed relationship between a source state and a target state. It tells an object how and where the next states are. A transition is commonly inscribed with related event, guard condition, and action responding to the event.

B. Promela

Promela [10], [11] is a verification modeling language. Promela defines of processes and data object. Processes are instantiations of "proctype" which defines a block of behavior. There must be a least one "proctype" declaration in a Promela model. A sample of a common "proctype" delcaration is written as follows.

```
proctype StateNameA() {byte activeflag; activeflag =1}
```

A proctype named "StateNameA" is declared with a local variable named "activeflag" and its value is assigned to 1. To instantiate a process of "StateNameA" proctype, the "init" process is written to do so. The "init" process is considered as the main entry in the model. A sample of a common "init" declaration is written as follows.

```
init { run StateNameA(); }
```

The "init" process instantiate a process to perform the behaviors defined in "StateNameA". In our approach, we intend to translate a state notation of the UML state machine diagram into a corresponding well-formed and encapsulated "proctype" in Promela and the transition flows of the UML state machine diagram into a corresponding "init" in Promela as well.

C. Object constraint language (OCL)

OCL [12] is a formal language which was developed in order for describing the constraints regarding the data objects of the UML diagrams. OCL is used to prescribe the preconditions, postconditions, and invariants, which are expressed in syntactical terms as follows.

- Preconditions and Postconditions

Precondition is a condition that must always be evaluated to be true prior to execute any activity or code section. While postcondition is a condition that should hold prior to the end up of activity or code section. The expression structures of precondition and postcondition are shown as follows.

```
context <TypeName>::<operation> (<parameters>)  
pre[<constraint name>]:<actualpreconditions>  
post[<constraint name>]:<actualpostconditions >
```

- Invariants

Invariants [12] are the conditions which must be true at all time, no matter what and when activity is performed by an object. The label "inv:" is used as shown.

```
context TypeName inv: <actual invariants>
```

In our approach, we assume that the OCL is used to describe the expressions being evaluated within each state. The written OCL compliant expressions would be parsed then the relevant predicated and expressions would be extracted and translated into Promela code as well.

III. METHOLOGY

In this section, the translation scheme of the UML state machine diagram into Promela code is shown in Fig. 2. The given UML state machine diagram would be drawn with the sufficient inscriptions of the preconditions, postconditions, invariants, and transitions' labels. Also, the translating rules are given to map between the common elements of the state machine diagram and the corresponding Promela block codes. The resulting Promela code is expected and executable by SPIN model checker. However, the resulting Promela code would be elaborated later to perform the additional activities as needed.

Definition 1: UML State machine diagram

A state machine diagram is a 5-tuple $SM = (ST, initialst, FINALST, CONTROLNODE, TX)$ where ST is a finite set of states. Each state would be prescribed with preconditions, postconditions, and invariants. These mapping functions, $PRECOND(ST)$, $POSTCOND(ST)$, and $INV(ST)$ provide the corresponding boolean expressions which would be translated into Promela code.

The $initialst$ is the only initial state, $FINALST$ is a finite set of the final states, $CONTROLNODE$ is a finite set of control nodes. $CONTROLNODE = \{choice, fork, terminate, shallowHistory, entryPoint, exitPoint, deepHistory, join, junction\}$. In this paper, we demonstrate only on choice control node. TX is a finite set of the ordered pairs (a,b) where $(a,b) \in (\{initialst\} \times ST) \cup (ST \times FINALST) \cup (CONTROLNODE \times ST)$. Each member of TX is inscribed with the labelling function $LB(TX) \rightarrow EVENT \times GUARD \times ACTION$, which denotes its event, guard condition, and action. Where $EVENT$ is a finite set of possible occurrences that can trigger a state transition. $GUARD$ is a set of

conditions to be evaluated and *ACTION* is a set of atomic activities to be performed when the relevant guards hold.

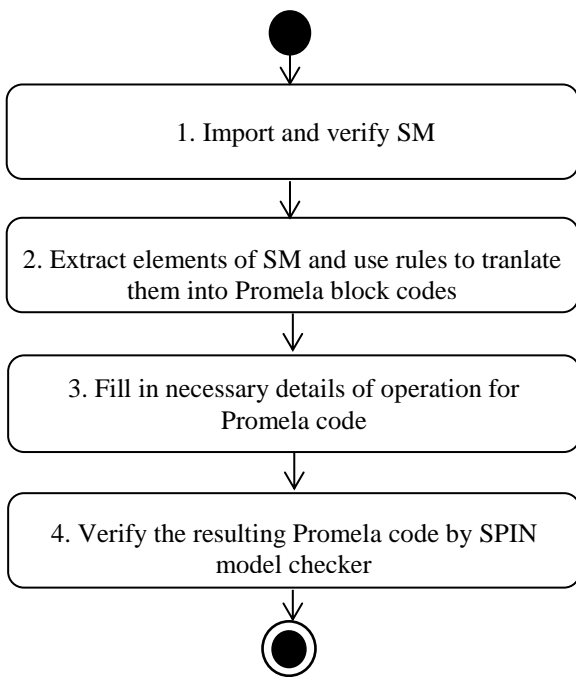


Fig. 2 Our Translation Scheme of The UML State Machine Diagram into Promela Code

A. Import and verify SM.

This step imports the UML state machine diagram *SM* written in XML format. The *SM* is simply verified its well-formedness before proceeding the next steps, otherwise the alert message would be prompted accordingly.

B. Extract elements of SM and use rules to translate them into Promela block codes.

In this step, we propose five translating rules to handle and map the *SM*'s elements into Promela block codes. The given *SM* would be parsed and the *SM*'s elements are extracted - which include initial state *initialst*, a set of states *ST*, a set of final states *FINALST*, a set of control nodes *CONTROLNODE*, a set of transitions *TX* in term of ordered pairs along with their labells from *LB(TX)*. We develop a support parsing tool using ANTLR to extract these *SM*'s elements and generate the target Promela block code according to the translating rules shown in Table I.

From Table I, five translating rules are listed. We intend to use the graphical notations of the *SM*'s elements, instead of the actual XML tags, shown on the left column and the right column shows their corresponding Promela block codes.

Rule1: Translating a State st_i in *ST*

Given a *SM*, each state st_i in *ST*, with its precondition $PRECOND(st_i)$ and postcondition $POSTCOND(st_i)$, found in the given *SM*, would be translated into a Promela proctype named *State st_i* . Several global variables are defined as shown in the block code, line 1-3, *stateStatus st_i* specifies the state status {idle, running, done}. Both preconditions and postconditions are asserted before and

after the performing of the manually inserted operations, line 10-15, if any.

Rule2: Translating the Initial State *initialst*

Typically, only one initial state would be found in the given *SM*. As mentioned in Promela, the process of type *init* always runs as the start process.

In short, we translate the initial state *initialst* into the *init{...}* block code, line 1-4.

Rule3: Translating a Final State *finalst $_i$* in *FINALST*

Given a *SM*, each final state *finalst $_i$* in *FINALST*, would be translated into a Promela proctype named *Finalst $_i$* . When it comes to the end of the *SM*, the variable called *Terminate* would be assigned to 1, in order to stop all active processes if any. This variable *Terminate* would asserted as a invariant of the system by using assertion command in the active process called "*checkInvariant()*", line 6-9.

Rule4: Translating the Fan-out Transitions Between Two States in *TX*

Given a *SM*, each fan-out transition (st_i, st_j) from the state st_i to the next state st_j would labelled with guard condition *Guard $_{ij}$* . We ensure that the guard condition *Guard $_{ij}$* must be held before the jumping from state st_i into state st_j in Promela code, line 3-5, otherwise the system would stay on state st_i .

Rule5: Translating a Choice Control Node in *CONTROLNODE*

Given a *SM*, a fan-out transition ($st_i, choice_i$) between state st_i and choice control node *choice $_i$* , and transition ($st_j, choice_i$) between state st_j and choice control node *choice $_i$* , followed by a transition (*choice $_i$, st $_k$*) between *choice $_i$* and state st_k , would be translating into the Promela block code shown in Table I. The guard conditions found on each transition would be checked to enable the jumping from a state to the next state. The guard condition *Guard $_i$* and *Guard $_k$* would be checked before the jumping from state st_i to the state st_k . While, the guard condition *Guard $_j$* and *Guard $_k$* would be checked alternatively before the jumping from state st_j to the state st_k . In this case we show only the choice with multiple incoming transitions and single outgoing transition.

C. Fill in the necessary details of operations in Promela codes

In this step, the resulting Promela codes from step B would be filled in manually, if any, with the operations to perform the built-in activities in a particular state st_i , called *st $_i$ Operation*.

D. Verify the resulting Promela codes using SPIN model checker checker.

The final resulting Promela codes from step C would be parsed and validated using SPIN model checker.

IV. DEMONSTRATION

This section demonstrates the translation of the UML state machine diagram into Promela codes. The simple

mockup UML state machine is drawn in Fig. 3, with five states and one initial state and one final state. Systematically, we would generate five Promela proctype processes for each state, St_1 , St_2 , St_3 , St_4 and St_5 . These proctype processes, called "stateSt1", "stateSt2", "stateSt3", "stateSt4" and "stateSt5", are generated respectively using Rule1. As shown in Fig. 4, the proctype called "stateSt1" handles its precondition and postcondition, line 7-11 and line 18-22. The state status of St1 has been observed via variable named "stateStatusSt1" showing the current status of idle, running, done. The additional behavior of its local operation would be manually filled at line 15.

The main process called "init" begins at the initial state, using Rule2. The rest of the transitions would be considered using Rule4 - Rule5 to generate the Promela code in this main process. We ensure that the previous Promela proctype for five normal states are remain unchanged no matter how frequently the flows of the transitions in the main process "init" are altered. The sample of the main process "init" is shown in Fig. 5.

At last, all of the final states would be considered. In order to intervene and stop all of the active processes in the system, we generate an active Promela process called "checkInvariant()" to concurrently monitor both invariants and our defined "Terminate" flag. As shown in the translating rules, whenever the irrelevant events occur, the "Terminate" flag would be set to 1. We consider that the final state would enable the "Terminate" flag as well. The Promela code are shown in Fig. 6 to handle the final states and the invariant conditions.

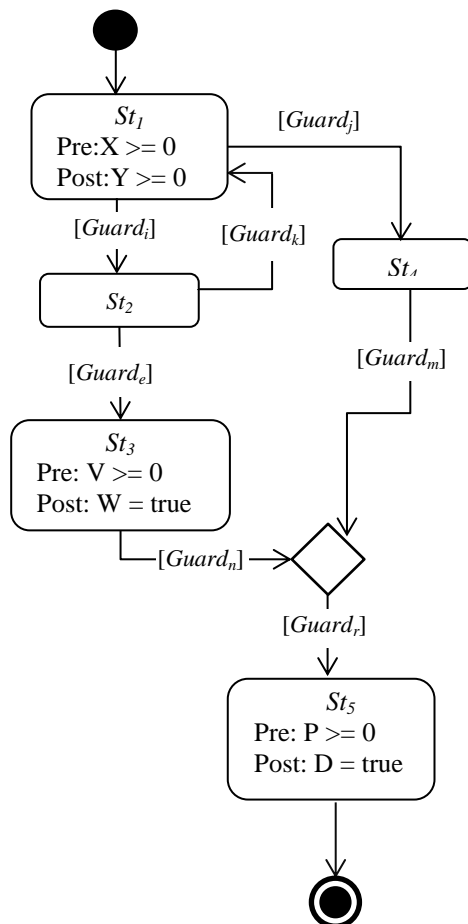


Fig. 3 The example of State machine diagram

```

1: mtype = {idle,runing,done};
2: mtype stateStatusSt1 = idle;
3: bool precFailSt1 = false;
4: bool postcFailSt1= false; int X=0;int Y =0;
5: proctype stateSt1() {
6:   stateStatusSt1= runing;
7:   precon:
8:   if
9:   ::(X >= 0) -> goto St1Operation;
10:  ::(X <=0) -> precFailSt1= true; Terminate =1;
11:  fi;
12:  St1Operation:
13:    atomic {
14:      stateStatusSt1 = runing;
15:      /* ...Fill in details of operation...*/
16:      goto postcon;
17:    }
18:  postcon:
19:  if
20:  ::(Y >= 0) -> stateStatusSt1 = done;
21:  ::(Y <=0) -> postFailSt1= true; Terminate =1;
22:  fi;
23: }
    
```

Fig. 4 Sample of Promela code of State St1

```

1:init{
2: St1 :
3: if
4: ::(stateStatusSt1 == idle)-> run stateSt1();
5: ::(stateStatusSt1 == done && Guardi == true)-> stateStatusSt1 = idle;
6:   goto St2;
7: ::(stateStatusSt1 == done && Guardj == true)-> stateStatusSt1 = idle;
8:   goto St4;
9: fi;
10: goto St1;
11: St2:
12: if
13: ::(stateStatusSt2 == idle) -> run stateSt2();
14: ::(stateStatusSt2 == done && Guarde)->stateStatusSt2 = idle; goto St3;
15: ::(stateStatusSt2 == done && Guardk == true)-> stateStatusSt2 = idle;
16:   goto St1;
17: fi
18: goto St2;
19: St3:
20: if
21: ::(stateStatusSt3 == idle)-> run stateSt3();
22: ::(stateStatusSt3== done&&Guardn == true &&
23:   Guardr == true)-> stateStatusSt3 = idle ; goto St5;
24: fi;
25: goto St3;
26: St4:
27: if
28: ::(stateStatusSt4== idle) -> run stateSt4();
29: ::(stateStatusSt4== done && Guardm == true && Guardr == true)->
30:   stateStatusSt4 = idle; goto St5;
31: fi
32: goto St4;
33: St5:
34: if
35: ::(stateStatusSt5 == idle)-> run stateSt5();
36: ::(stateStatusSt5 == done)-> stateStatusSt5 = idle; run Finalst();
37: fi;
38: goto St5;
39: }
    
```

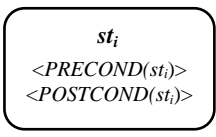
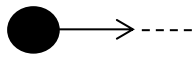
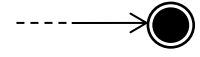
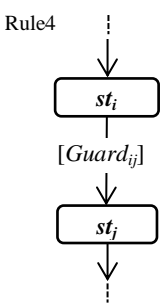
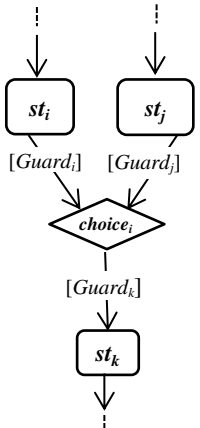
Fig. 5 Sample of the main process "init"

```

1: bit Terminate = 0;
2: proctype Finalst()
3: {
4:   Terminate = 1;
5: }
6: active proctype checkInvariant()
7: {
8:   do
9:   ::assert(Terminate==0);
10:  od
11: }
    
```

Fig. 6 Sample of the Promela code of the final state

TABLE I
OUR TRANSLATION RULES

#	Element of state machine	Promela code skeleton
1		<pre> 1: mtype = {idle,running,done}; 2: mtype stateStatusst_i= idle; 3: bool precFailst_i= false, postFailst_i= false; 4: Proctype Statesst_i (){ 5: stateStatusst_i = running; 6: precon: 7: if PRECOND(st_i) == true 8: then goto st_iOperation; 9: else precFailst_i= true; Terminate =1; 10: st_iOperation: 11: atomic { 12: stateStatusst_i = running; 13: /* ...Fill in details of operation...*/ 14: goto poscon; 15: } 16: postcon: 17: if POSTCOND(st_i)== true 18: then stateStatusst_i = done; 19: else postFail = true; Terminate =1; 20: }</pre>
2		<pre> 1: init{ 2: 3: 4: }</pre>
3		<pre> 1: bit Terminate = 0; 2: proctype Finalst_i() 3: { 4: Terminate = 1; 5: } 6: active proctype checkInvariant() 7: { 8: do 9: ::assert(Terminate==0); 10: od 11: }</pre>
4		<pre> 1: st_i : 2: if stateStatusst_i== idle then run statesst_i(); 3: if stateStatusst_i== done && 4: Guard_ij ==true 5: then stateStatusst_i = idle; goto st_j; 6: goto st_i; 7: st_j : 8: :</pre>
5		<pre> 1: st_i : 2: if stateStatusst_i== idle 3: then run statesst_i(); 4: if stateStatusst_i== done && 5: Guard_i==true &&Guard_k==true; 6: then stateStatusst_i = idle; goto st_k; 7: goto st_i; 8: st_j : 9: if stateStatusst_j==idle 10: then run statesst_j(); 11: if stateStatusst_j==done&& 12: Guard_j==true &&Guard_k==true; 13: then stateStatusst_j = idle; goto st_k; 14: goto st_j; 15: st_k: 16: if stateStatusst_k== idle 17: then run statesst_k(); 18: :</pre>

V. CONCLUSION

This paper proposes an alternative to automatically generate the Promela code from the UML state machine diagram with some informative inscriptions such as preconditions, postconditions, and the transition elements - events, guards, and action, etc. A set of translating rules are proposed to systematically map the elements of the given state machine diagram into the corresponding Promela block codes. The resulting Promela code is well structured so that the Promela code of the states would remain unchanged when the main flow of the transitions are altered as shown in our demonstration section. In this paper, we focus only five main elements of the UML state machine including the initial state, the final state, the normal state, the choice notation, and the transition with its labels, shown in Fig. 1. If needed, the local activities of the states would manually be filled in. For practical purpose, we developed a support tool to do the automatic translation of the UML state machine diagram, written in XML format, into the resulting Promela code.

REFERENCES

- [1] M. Oubelli, N. Younsi, A. Amirat, and A. Menasria. (2011). From UML 2.0 sequence diagrams to PROMELA code by graph transformation using ATOM3. *CIIA*. Volume 825 of CEUR Workshop Proceedings, CEUR-WS.org. [Online]. Available: http://ceur-ws.org/Vol-825/paper_183.pdf
- [2] D. Latella, I. Majzik, and M. Massink, "Automatic verification of a behavioral subset of UML statechart diagrams using the SPIN model-checker," *Formal Aspects of Computing*, Vol. 11, no. 6, pp. 637-664, Dec. 1999. [doi:10.1007/s001659970003].
- [3] V.Lima, C. Talhi, D. Mouheb, M. Debbabi, and L. Wang, "Formal verification and validation of UML 2.0 sequence diagrams using source and destination of messages," in *the 4th International Workshop on Systems Software Verification (SSV 2009)*, Vol. 254, pp. 143-160, Oct. 2009.
- [4] T. Schafer, A. Knapp, and S. Merz. (2001). Model checking uml state machines and collaborations. electronic notes in *theoretical computer science* 47. [Online]. Available: <https://members.loria.fr/SMerz/papers/sw-mc01.pdf>
- [5] Object Management Group (OMG). (2015, March). OMG unified modeling language TM (OMG UML) version 2.5. [Online]. Available: <http://www.omg.org/spec/UML/2.5/PDF/>
- [6] J. Rumbaugh. (2004, July). The unified modeling language reference manual (2nd ed.) [Online]. Available: https://www.utdallas.edu/~chung/Fujitsu/UML_2.0/Rumbaugh--UML_2.0_Reference_CD.pdf
- [7] V. H. Lingegowda, "Building Graphical Promela Models using UPPAAL GUI," M.S. Thesis, Department of Computer Science, Aalborg Univ., Denmark, 2006.
- [8] J. Lilius and I. P. Paltor, "vUML: a tool for verifying UML models," in *Automated Software Engineering, 14th IEEE International Conf.*, 12-15 Oct. 1999.
- [9] refbook.book. (2004, June). State machin view. [Online]. Available: http://www.ibm.com/developerworks/rational/library/content/04August/3153/3153_Rumbaugh_ch07.pdf
- [10] G. J. Holzmann, *Spin Model Checker: The Primer and Reference Manual*. New York: Addison-Wesley Professional, 2003.
- [11] Spin Checked. Verifying multi-threaded software with spin. [Online]. Available: [HTTP://SPINROOT.COM](http://SPINROOT.COM).
- [12] Object Management Group. (2014, February). Object constraint language version 2.4. [Online]. Available: <http://www.omg.org/spec/OCL/2.4>