SysML State Machine Diagram to Simple Promela Verification Model Translation Method

Takahiro Ando¹, Yuya Miyamoto², Hirokazu Yatsu¹, Kenji Hisazumi³, Weiqiang Kong⁴, Akira Fukuda¹, Yasutaka Michiura¹, Keita Sakemi⁵, and Michihiro Matsumoto⁵

¹Graduate School of Information Science and Electrical Engineering, Kyushu University, Fukuoka, Japan
²DOCOMO Systems Inc., Tokyo, Japan
³System LSI Research Center, Kyushu University, Fukuoka, Japan
⁴School of Software, Dalian University of Technology, Dalian, China
⁵Japan Manned Space Systems Corporation, Tsukuba, Ibaraki, Japan

Abstract—In this study, we developed a method for converting SysML state machine diagrams into Promela models that can be verified using the SPIN model checking tool. The Promela code generated in our approach is a sequential verification model that simplifies the verification process when used in the early stages, and also prevents state explosion in the verification process. Thus, using the sequential verification model reduces the cost of the overall verification process. In this paper, we describe the rules used to convert the SysML state machine diagrams with parallel processes to a single sequential process in Promela.

Keywords: State Machine Diagram, SPIN, SysML, Model Checking, Formal Method

1. Introduction

Software is embedded in various devices, machines, and equipment, including smartphones, automobiles, space equipment. Because this embedded software needs to be very reliable, at each stage of development, rigorous verifications are required.

Model checking [1] is a well-known verification technique for formally analyzing state transition systems. In model checking, a target system is modeled in a formal description language and the model is exhaustively explored to check whether desired properties of the system are satisfied. SPIN [2], NuSMV [3], and UPPAAL [4] are state-of-the-art model checkers. To use any of these tools, a state transition diagram of the target system is first modeled in the formal description languages corresponding to the desired model checker. Further, the properties to be checked are written in a formal specification language such as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL).

In formal verification such as model checking, factors such as verification costs, time, memory size, human resource, are major issues. Therefore, during the early stage of verification processes, verification with a simple model to reduce the cost is desired. The quality of the verification model is dependent on the skill of the verification engineer, which is often a problem. Therefore, automatic generation of the verification model, i.e., making it independent of the engineer’s skill, is desired. In addition, because analysis of the verification results often tends to be complex, a technique that simplifies this process is also necessary.

In this paper, we discuss the automatic generation of simple verification models from the state machine diagrams in SysML [5]. In particular, we focus on the verification model used in the SPIN model checker, and propose a translation method that converts SysML state machine diagrams into simple verification models for SPIN. Using our translation method, the behavior of a state machine diagram with parallel processes is translated into a simple verification model with a single process in Promela, the input language used by SPIN. We demonstrate the efficacy of our proposed method by applying it to verification of a simple system.

2. Related Work

Much research has been conducted on the application of formal verification techniques to formally analyze state machine diagrams. Bhaduri and Ramesh [6] carried out a comprehensive survey of studies that applied model checking to state machines, in which various model checkers including SPIN [2], SMV [7], and FDR [8] were used.

Latella et al. [9] translated state machines into models written in Promela and then verified them using SPIN. However, they only dealt with the basic components of state machine diagrams. Lilius and Paltor [10] proposed a tool called vUML for verification of UML models using SPIN, but presented no details of the rules used to translate the models into Promela.

3. Proposed Translation Method

In this section, we describe our proposed state machine diagram to verification model translation method. The verification model is written as a simple process in Promela. Each element of the model is lumped using a macro description, in order to easily recognize the correspondence between the elements of the original diagram.
In our verification model, the parallel processes in a state machine are combined into one sequential process in Promela. Although, our verification model cannot represent all the behaviors of the diagram with parallel processes, it is useful in the early verification stage because a simple model prevents state explosion and reduces the cost of verification.

3.1 Overview
The inputs to our translation method are the state machine diagram and information about its variables. Information about the serialized state machine is stored in an XMI file, and information about its variables is stored in a CSV file. The translation rules shown below translate the components of the diagram into their corresponding description in Promela.

1) The state names in the diagram are translated into mtype values in Promela.
2) An mtype variable that represents the current state is provided for each region in the diagram.
3) The event names in the diagram are translated into mtype values.
4) An mtype variable that represents the current event occurrence is provided.
5) On the basis of the input variable information, the corresponding variables in Promela are provided for the variables in the statement or guard condition in the diagram.
6) The initial pseudo state is translated into an inline macro description.
7) The states, including composite states, are translated into three types of inline macros.
8) The outgoing transitions of a state are translated into one inline macro.
9) The regions are translated into two types of inline macros.
10) The final state in each region is translated into an inline macro.
11) The event occurrence model is described as an inline macro.
12) The behavior of the original state machine is described by one process in Promela.

In the following subsections, we explain our translation rules in detail along with translation examples. Each of the following examples are obtained by translating the corresponding element of the state machine diagram shown in Fig. 1.

3.2 Declaration of Values and Variables
Each state name and event name is translated into an mtype value in Promela (lines 1–4, 10). Each variable representing the current state in a region is declared an mtype variable (lines 6–9), and each variable representing current event occurrence is declared an mtype variable (line 11). Each variable in guard conditions and actions in the state

3.3 Translation of Initial Pseudo States
Initial pseudo states are translated into inline macros that each has an inline macro call for the state entry behavior. The state is pointed to by the initial pseudo state.

3.4 Translation of States
Each state is divided and translated into three inline macros: entry part (lines 1–6), do part (lines 8–11), exit part (lines 13–19). The entry part represents the entry behavior of the state, the do part represents the do behavior, and the exit part represents the exit behavior. In addition, when the state has regions, the entry part has inline macro calls for the entry behaviors into the regions (lines 4–5), the do part

3.3 Translation of Initial Pseudo States
Initial pseudo states are translated into inline macros that each has an inline macro call for the state entry behavior. The state is pointed to by the initial pseudo state.

3.4 Translation of States
Each state is divided and translated into three inline macros: entry part (lines 1–6), do part (lines 8–11), exit part (lines 13–19). The entry part represents the entry behavior of the state, the do part represents the do behavior, and the exit part represents the exit behavior. In addition, when the state has regions, the entry part has inline macro calls for the entry behaviors into the regions (lines 4–5), the do part

```
1 mtype = {top_init, top_stop, top_doing, top_final};
2 mtype = {doing_heat_init, doing_heat, doing_wait};
3 mtype = {doing_humidification_init, doing_humidifyon, doing_humidifyoff};
4 mtype topState = top_init;
5 mtype doing_heatState = doing_heat_init;
6 mtype doing_humidificationState = doing_humidification_init;
7 mtype = {NULL, finish, poweron, poweroff, when01, when02, humidifyon, humidifyoff};
8 mtype event = NULL;
9 int nowtemp = 25;
10 int maxtemp = 30;
11 int mintemp = 20;
12 bool heater = false;
13 bool tempmonitor = false;
```
has inline macro calls for the behavior in the regions (lines 9–10), and the exit part has inline macro calls for the exit behavior from the regions (lines 14–15).

These inline macro calls in each part are not parallel but sequential. Consequently, an execution sequence in our verification model is also sequential, and the correspondence between the execution sequence and the lines of the verification model code can be easily observed.

```
inline S_doing_entry() {
    topState = top_doing;
    T_doing_heat_init();
    T_doing_humidification_init();
}

inline S_doing() {
    R_doing_heat();
    R_doing_humidification();
}

inline S_doing_exit() {
    R_doing_heat_exit();
    R_doing_humidification_exit();
    doing_heatState = doing_heat_init;
    doing_humidificationState = doing_humidification_init;
}
```

### 3.5 Translation of Outgoing Transitions

Our transition rules translate all outgoing transitions of a state into an inline macro in a group. The internal transitions are dealt with in a manner similar to the outgoing transitions and they are translated into the same macro as outgoing transitions. In this macro, the transitions are described as conditional branches by the trigger event occurrences. The inline macro call of the exit behavior of the source state is placed before the actions of each transition. The inline macro call of the entry behavior of the target state is placed after the actions. These behaviors are obtained from the semantics of the state machine diagrams.

```
inline R_doing_heat() {
    if :: (doing_heatState == doing_heat) ->
        S_doing_heat();
        T_doing_heat();
    :: (doing_heatState == doing_wait) ->
        S_doing_wait();
        T_doing_wait();
    fi
}

inline R_doing_heat_exit() {
    if :: (doing_heatState == doing_heat) ->
        S_doing_heat_exit();
    :: (doing_heatState == doing_wait) ->
        S_doing_wait_exit();
    fi
}
```

### 3.6 Translation of Regions

Each region of the state machine diagram is translated into an inline macro that combines the inline macro calls of the states and transitions in the region (lines 1–9). The inline macro represents the behavior in the region. The exit behavior from the region is translated into a different inline macro (lines 11–19).

```
inline S_final() {
    topState = top_final;
    break
}
```

### 3.7 Translation of Final State

The final state is translated into an inline macro that terminates the operations of the behavior in the region.

```
inline S_final() {
    topState = top_final;
    break
}
```

### 3.8 The Event Occurrence Model

In our verification model, we adopted the following event occurrence model. In the model, the events that can occur are the events that are triggers of the current states, including inner states.

```
inline eventOccur() {
    event == NULL;
    if :: (topState == top_stop) ->
        event = poweron
    :: (topState == top_stop) ->
        event = finish
    :: (topState == top_doing) ->
        event = poweroff
    :: (topState == top_doing &&
        doing_heatState == doing_heat &&
        nowtemp > maxtemp) ->
        event = when01
```
3.9 Process for State Machine Behavior

Only one process is present in our Promela verification model. This process represents the overall operation of the original state machine diagram. It starts from the inline macro call of the initial pseudo state and alternates the event occurrences and the state transitions for the events.

```
active proctype stm() {
    T_init();
    do
        :: eventOccur();
        R_top()
    od
}
```

4. Case Study

In this section, we confirm the efficacy of our translation method for SPIN model checking.

We applied our translation method to the state machine diagram shown in Fig. 1, and generated the corresponding Promela code. Using the code as the input to the SPIN model checker, we conducted LTL verification. We used the following LTL formula, which states that when the current state is the doing state, the stop state cannot be reached forever, as a verification property.

```
[] (topState == top_doing) ->
    [ ](topState != top_stop)
```

The result of this verification case study is shown in Fig. 2. The figure shows that a transition sequence from the doing state to the stop state has been found. This is appropriate as a result for this verification case study, and shows that the generated model can be used for SPIN model checking.

The counterexample sequence for this case study is shown in Fig. 3. The figure shows that the complexity due to the execution order of the parallel behavior has been rectified—making it easy to observe the correspondence between the steps in counterexample and the code corresponding the verification model.

These results show that our proposed method is useful in the model checking process.

5. Conclusions

In this paper, we described our proposed translation method that converts state machine diagrams with parallel processes into simple SPIN verification models with a single Promela process. In our simple verification model, using inline Promela macros, we can easily recognize the correspondence between Promela codes and each component of the original diagram. Our translation rules convert states that have some regions into inline macros with sequential inline macro calls for the regions. The verification model alternates the event occurrences and state transitions for the event. Thus, a state transition sequence in our model is simple, and we can analyze the verification results more easily.

In future work, we plan to develop a translation method similar to the one proposed here, but which translates state machine diagrams into simple parallel verification models. We believe that using a one-step complex model, we can implement stepwise from simple verification to full verification, and then we can further reduce the verification cost. In addition, we plan to develop an automatic translation system based on our translation method. We are also planning to apply our method to various examples and refine the translation rules using feedback.

References

Fig. 2: Result of verification case study using the SPIN model checker

```
andor-macrosample andor spin -o '!' [] ((topState == top_doing) -> []((topState == top_stop) || humid_heat.pml))
andor-macrosample andor gcc -o pan pan.c -DSAFETY
andor-macrosample andor ./pan
warning: never claim + accept labels requires -a flag to fully verify
warning: for p.a. reduction to be valid the never claim must be stutter-invariant
(never claims generated from Lit. formulae are stutter-invariant)
pan: assertion violated [] !((topState==3)) (at depth 72)
pan: wrote humid_heat.pml.trail

(Spin Version 6.4.5 — 3 January 2006)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
  never claim
  assertion violations
  cycle checks
  disabled by -DSAFETY
  invalid end states
State-vector 36 bits, depth reached 72, errors: 1
  37 states, stored
  0 states, matched
  37 transitions (= stored+matched)
  0 atomic steps
hash conflicts: 0 (resolved)
States on memory usage (in Megabytes):
  0.002 equivalent memory usage for states [stored+State-vector+overhead]
  0.250 actual memory usage for states
  128.000 memory used for hash table (=203)
  0.524 memory used for DFS stack (=1888)
  128.730 total actual memory usage

pan: elapsed time 0.81 seconds
andor-macrosample andor
```

Fig. 3: Counterexample of a verification for the system using the SPIN model checker