Formalization and Model Checking of SysML State Machine Diagrams by CSP#

Takahiro Ando[1], Hirokazu Yatsu[1], Weiqiang Kong[1], Kenji Hisazumi[2], and Akira Fukuda[1]

[1] Graduate School of Information Science and Electrical Engineering, Kyushu University, Japan
{ando.takahiro,hirokazu.yatsu,fukuda}@f.ait.kyusyu-u.ac.jp, weiqiang@qito.kyushu-u.ac.jp

[2] System LSI Research Center, Kyushu University, Japan
nel@slrc.kyushu-u.ac.jp

Abstract. SysML state machine diagrams are used to describe the behavior of blocks in the system under consideration. The work in [1] proposed a formalization of SysML state machine diagrams in which the diagrams were translated into CSP# processes that could be verified by the state-of-the-art model checker PAT. In this paper, we make several modifications and add new rules to the translation described in that work. First, we modify three translation rules, which we think are inappropriately defined according to the SysML definition of state machine diagrams. Next, we add new translation rules for two components of the diagrams – junction and choice pseudostates – which have not been dealt with previously. As the contribution of this work, we can achieve more reasonable verification results for more general SysML state machine diagrams.

Keywords: SysML state machine diagrams, formal semantics, model checking, CSP#.

1 Introduction

The OMG Systems Modeling Language (SysML) [2] is a systems modeling language that supports specification description, design, analysis, and verification of systems.

SysML is a language extension of Unified Modeling Language (UML) [3], the de facto standard software modeling language. SysML has nine types of diagrams and the state machine diagrams are one of them. In such diagrams, the life-cycle behavior of a block in a system is expressed as a state transition system. SysML diagrams, including state machine diagrams, do not have a strict formal semantics. This interferes with checking for correctness of the description and makes it difficult to verify with formal methods especially.

Model checking [4] is a well-known formal verification technique for formally analyzing state transition systems. In model checking, a target system is modeled with a formal description language and the model is then exhaustively

B. Murgante et al. (Eds.): ICCSA 2013, Part III, LNCS 7973, pp. 114–127, 2013.
© Springer-Verlag Berlin Heidelberg 2013

explored to check whether desired properties of the system are satisfied. SPIN [5], NuSMV [6], and UPPAAL [7], etc. are state-of-the-art model checkers. In using these tools, a state transition system is modeled in their respective formal description languages, and properties to be checked are written in formal specification languages such as Linear Temporal Logic (LTL) and Computation Tree Logic (CTL), etc.

PAT [8] is a model checking tool whose performance is comparable to that of SPIN, and it uses CSP# as its model description language. CSP# is a language extension of Hoare's Communicating Sequential Process (CSP) [9] with description supports for conditional choice, shared variable and, sequential programs, etc. CSP is a kind of process algebra and it has been used in the formalization of concurrent systems. CSP and CSP# are suitable to describe processes with interruptions and parallel processes.

In this paper, we investigate formalization and model checking for SysML state machine diagrams. In our formalization, the diagrams are translated into CSP# descriptions, and then, we apply PAT model checker to check the CSP# models against desired properties. Similar translation rules from the UML state machine diagrams into CSP# have been proposed previously in [1]. In this paper, we make modification to some translation rules described in [1], which we think are inappropriate according to the definition of SysML state machine diagrams. In addition, we give new translation rules for two components of the diagrams that have not been dealt with in [1]. Moreover, we demonstrate and evaluate our translation rules with a case study for a simple user certification system.

Organization. In Section 2, we describe related work. Section 3 presents our translation rules for SysML state machine diagram. In Section 4, we demonstrate and evaluation our translation rules in a case study. Section 5 concludes this paper and mentions future work.

2 Related Work

Quite a lot research has been done for applying formal verification technique for formally analyzing state machine diagrams. For example, the work in [10] gives a comprehensive survey on researches that apply model checking to state machines, in which various model checkers including SPIN [5], SMV [11], and FDR [12], etc. are used.

In [13], state machines are translated into models written in the Promela language and then verified using SPIN. This work deals only with basic components of state machine diagrams. In [14], a tool called vUML is proposed for verification of UML models using SPIN, but detailed translation rules into Promela, the input language of SPIN, are not described.

Translation for the state machines into SMV input format has been proposed in [15]. In the work, each state (and each event) is assigned a single variable for symbolic model checking. Therefore, even for small diagrams, the state explosion problem in symbolic model checking can become more obvious due to the increase of variable numbers. In [16], the semantics and a symbolic encoding of UML

state machine diagrams, which has more complex notations, are shown and the methods of verification is based on NuSMV. The symbolic encoding of state machines has also been given in [17]. The work proposes to apply SAT-based bounded model checking for analyzing state machines.

Formalizations in which state machines are translated into process algebra have been proposed in [18], [19], [20], and [1]. These works are closely related to our work. In [18], state machines are formalized with mCRL2 [21], a language based on the Algebra of Communicating Process (ACP) [22], and the model is verified using the mCRL2 toolset. In [19], [20], and [1], CSP is selected as the process algebra to formalize state machines. The formalization in [20] can handle entry and exit behaviors, but cannot handle some pseudostates which represent complex transitions. And, formalized models in [20] are verified by FDR2 [12] model checker.

[1] is a direct predecessor of this paper. The work gives a formalization in which state machines are translated into CSP#, a variant language of CSP. In this paper, we make modifications to some translation rules proposed in [1], and add several rules for components of state machine diagrams that are not dealt with in that work.

3 Formalization of State Machine Diagram

In this section, we briefly explain the elements of SysML state machine diagrams and the formal language CSP#. Then, we give the translation rules from state machine diagrams into CSP.

3.1 State Machine Diagram of SysML

When modeling a system in SysML, a state machine diagram is used for illustrating behavior of a block that is a component of the system. A state machine diagram consists of states and transitions between the states. The complete syntax and notations of state machine diagrams are defined in [2].

A state represents a situation in the life-cycle of the block. In general, the situation is expressed by invariant conditions, etc. Each state is allowed to have *entry*/*do*/*exit* behavior. The *entry* behavior is performed when a block goes into the state, and the *exit* behavior is performed when the block leaves the state. The *do* behavior is performed after the *entry* behavior, and it continues performing until it terminates or the block leaves the state.

There are several types of states such as atomic states, composite states and submachine states. An atomic state has no sub-state. A composition state has one or more orthogonal regions and each region has sub-states. A submachine state expresses a state machine that is to be reused in other diagram. The *entry*/*exit* points of a submachine state correspond to initial/final states in the diagram which the submachine state refers to.

A transition between states is allowed to have three attributes – *trigger*, *guard*, and *effect*. An event acts as a trigger that activates the transition. Only if the

trigger event occurs and the guard condition is satisfied, the transition can be fired. When the transition is fired, its effects are performed and then the machine changes from the current state changes to its target state.

In addition, state machine diagrams have several types of pseudostates. Initial pseudostates, final states, junction pseudostates, and choice pseudostates, etc. are representative pseudostates. Initial pseudostates and final states represent the start and end of state machines, respectively. Junction pseudostates and choice pseudostates are notations that are used to describe common parts of multiple transitions.

3.2 CSP#

CSP [9] is a process algebra and is well-known as a formal language for describing interaction patterns of parallel systems. CSP# is the system description language of the model checking tool PAT, which adds conditional choice and shared variable, etc. to CSP.

In the following, we enumerate the elements of a subset CSP# that are used in this paper by giving their intuitive meanings.

Definition 1. *A process P is defined as follows.*

$$\begin{aligned} P ::= & Stop \mid Skip \mid e\{prog\} \rightarrow P \mid P_1; P_2 \mid P_1 \Box P_2 \\ & \mid P_1 \parallel P_2 \mid P_1 ||| P_2 \mid [b]P \mid P_1 \triangle P_2 \\ & \mid case\{b_1 : P_1;\ b_2 : P_2;\ \cdots;\ default : P\} \end{aligned}$$

where, P_1 and P_2 are processes, e is an event name that may have optional sequential program prog, and b, b_1 and b_2 are boolean expressions.

Stop represents a dead lock process and *Skip* represents a process that terminates successfully. $e \rightarrow P$ represents a process that performs as process P after event e occurs. When e has a sequential program, it performs the program atomically, and afterwards behaves as the process P. $P_1; P_2$ behaves firstly as process $P1$ first and then as process $P2$. $P_1 \Box P_2$ represents a non-deterministic choice, where the first occurred event determines whether it behaves as P_1 or P_2. $P_1 \parallel P_2$ represents a parallel composition process that synchronizes the common events of P_1 and P_2. $P_1 ||| P_2$ represents an interleaved parallel composition. $[b]P$ represents a process with a guard condition and, only if the boolean expression b is satisfied, it behaves as P. $P_1 \triangle P_2$ represents a process having an interrupt process. It behaves as P_1 until the first event of P_2 occurs and, after the event occurs, it behaves as P_2. In *case* process, condition expressions are checked in turn, and it behaves as the process that corresponds to the first found satisfied condition.

3.3 Translation Rules

Next, we give translation rules from state machine diagrams into CSP#. In CSP#, states and transitions as well as the whole state machine diagrams are

Table 1. Transition Rules from State Machine Diagrams to CSP#

Elements	**CSP#**	**Comments**
State Machine *sm*	$tr(sm) = tr(i)$, where i is the top level initial pseudo state of *sm*.	Same for regions in composite state.
Initial pseudostate i	$tr(i) = tr(s)$, where s is the target state of the outgoing transition from i.	Same for initial pseudostates in any region.
Final state f	$tr(f) = Skip;$	This means a state machine terminates successfully.
Atomic State s	$tr(s) = tr(entry); ((tr(do); Stop) \triangle (tr(t_1) \Box tr(t_2) \Box \cdots \Box tr(t_n)))$, where $t_1, t_2, \ldots, t_n$ are outgoing transitions from s.	The process of coming after $tr(do)$ is not *Skip* but *Stop*.
Entry behavior *entry*	$tr(entry) = e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow e_n \rightarrow Skip$, where $e_1, e_2, \ldots, e_n$ are actions in a sequence of entry behavior.	Same for do/exit behavior of a state and effect of a transition.
Transition t	$tr(t) = [guard](event \rightarrow tr(exit); tr(effect); tr(s_t))$, where *exit* is a exit behavior of source state and s_t is a target state.	
Composite state *cs*	$tr(cs) = tr(entry); (((tr(do); Stop) \mid\mid\mid tr(r_1) \mid\mid\mid tr(r_2) \mid\mid\mid \cdots \mid\mid\mid tr(r_m)) \triangle (tr(t_1) \Box tr(t_2) \Box \cdots \Box tr(t_n)))$, where $r_1, r_2, \ldots r_m$ are regions in *cs* and $t_1, t_2, \ldots, t_n$ are outgoing transitions.	

all described as processes of CSP#. Events and statements in state machine diagrams are translated into events of CSP#.

Table 1 summarizes translation rules for the elements of state machine diagrams. In the table, for brevity, translation rules are expressed by using an informal function *tr*, and the translation result for an element e is denoted as $tr(e)$.

Our rules in Table 1 are based on the rules proposed previously in work [1], however there are several differences between them. In the original translation rules of [1], the *entry*/*exit* behavior of a state and the *effect* of a transition are translated into an atomic process in the form of a sequence of events or statements. However, in our rules, atomic notations are not used for sequences of events or statements, because we believe there is a possibility that certain unsafe behavior of the target system could be hidden due to over abstraction when these sequences are treated as atomic processes in CSP#. That is, when the sequences are treated as atomic processes, the number of behavior patterns are reduced due to decrease of possible interleaving composition of the processes. As a result, certain event occurrences order that may lead to a problem/bug might not be checked when verifying the processes.

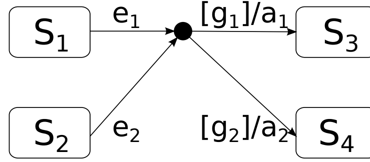

Fig. 1. An Example with Junction Pseudostate

In addition, our translation rules for states have two differences compare to the original rules. The first difference is that the $tr(entry)$ process that represents *entry* behavior is excluded from the region in which an interrupt from an outgoing transition process is caught. This is to reflect that entry behavior must be completed before other behavior and transitions become executable.

Another difference is to concatenate $tr(do)$ process and *Stop* process, which represent *do* behavior and dead lock, respectively, by using operator ";". In the original translation, a translated process is allowed to terminate successfully even if no outgoing transitions are fired after a *do* behavior of some state completes. This is judged from the following two facts.

- When $tr(do)$ process completes, the process behaves as a *Skip* process.
- In CSP#'s semantics, it is allowed that the process "$Skip \triangle P$" terminates successfully without executing process P.

Such a translation is not appropriate because it allows for state machines to terminate successfully in a state that is not a final state. From the CSP# equation, $Stop \triangle P = P$, the behavior that ignores the outgoing transitions can be prohibited by revising "$tr(do)$" as "$tr(do); Stop$".

Next, we give new translation rules for two pseudostates, i.e., junction pseudostates and choice pseudostates, which are used as representations for organizing multiple transitions.

Junction Pseudostates. A junction pseudostate is used to describe common parts of multiple transitions. Therefore, a junction pseudostate can be translated in CSP, by translating the original transitions that are unwound by using information of incoming/outgoing transitions of the pseudostate.

The unwound transition set of a junction pseudostate $Trans_{unwound}$ is given as follows, where $Trans_{in}$ is the set of incoming transitions of the pseudostate and $Trans_{out}$ is the set of outgoing transitions.

$$\begin{aligned} Trans_{unwound} = \{t \mid \exists t' \in Trans_{in}.\ \exists t'' \in Trans_{out}.\\ \big(source(t) = source(t') \land event(t) = event(t')\\ \land\ guard(t) = guard(t'') \land effect(t) = effect(t'')\\ \land\ target(t) = target(t'')\big)\} \end{aligned}$$

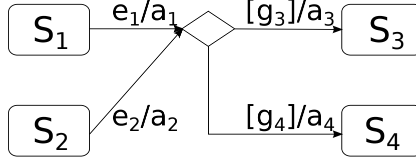

Fig. 2. An Example with Choice Pseudostate

Where the five functions *source*, *event*, *guard*, *effect* and *target* return corresponding source state, event, guard, effects and target state, respectively, of the transition given as an input of the functions. The translation for a junction pseudostate and its incoming and outgoing transitions is the result of applying the translation rule for transitions to each element of $Trans_{unwound}$. For example, the junction pseudostate and its incoming/outgoing transitions on Fig.1 are translated into the following four processes.

$$P_{13} = [g_1]\big(e_1 \rightarrow tr(exit(S_1));\ tr(a_1);\ tr(S_3)\big)$$
$$P_{14} = [g_2]\big(e_1 \rightarrow tr(exit(S_1));\ tr(a_2);\ tr(S_4)\big)$$
$$P_{23} = [g_1]\big(e_2 \rightarrow tr(exit(S_2));\ tr(a_1);\ tr(S_3)\big)$$
$$P_{24} = [g_2]\big(e_2 \rightarrow tr(exit(S_2));\ tr(a_2);\ tr(S_4)\big)$$

where, *exit* is a function that takes a state and returns *exit* behavior. The processes, P_{13}, P_{14}, are referred in $tr(S_1)$ as translation results of outgoing transitions of the state S_1 as follows.

$$tr(S_1) = tr(entry(S_1));\ \big((tr(do(S_1));\ Stop) \bigtriangleup (P_{13} \Box P_{14})\big)$$

P_{23}, P_{24} are referred in $tr(S_2)$ similarly.

Choice Pseudostates. A choice pseudostate has multiple incoming and outgoing transitions as a junction pseudostate does. However, the timing to evaluate the guard conditions of outgoing transitions is different from a junction pseudostate. For a choice pseudostate, the guards of outgoing transitions are not evaluated until reaching the choice pseudostate. Thus, the result of *effects* of the incoming transition influences the evaluations. Based on the above analysis, divided into an incoming part and an outgoing part, a choice pseudostate and its related transitions are translated into CSP#.

An incoming transition of a choice pseudostate is translated into CSP# in almost the same way as a transition between states. However, instead of the process of the target state, the process P_{choice}, the translation result of the outgoing part, is used.

Each outgoing transition is also translated almost like a translation between states. However, the process for the *exit* behavior of the source state is omitted. This is because the choice pseudostate is considered as the source state of these

outgoing transitions. The process P_{choice}, which represents the set of outgoing transitions of the choice pseudostate, is defined as follows.

$$P_{choice} = tr(t_1) \,\Box\, tr(t_2) \,\Box \cdots \Box\, tr(t_n)$$

where, $t_1, t_2, \ldots, t_n$ are outgoing transitions of the choice pseudostate. The choice pseudostate and its related transitions on Fig. 2 are translated as follows.

$$\begin{aligned}
P_1 &= e_1 \rightarrow tr(exit(S_1));\; tr(a_1);\; P_{choice} \\
P_2 &= e_2 \rightarrow tr(exit(S_2));\; tr(a_2);\; P_{choice} \\
P_3 &= [g_3]\big(tr(a_3);\; tr(S_3)\big) \\
P_4 &= [g_4]\big(tr(a_4);\; tr(S_4)\big) \\
P_{choice} &= P_3 \,\Box\, P_4
\end{aligned}$$

where, P_1, P_2 are processes that represent transitions from states S_1 and S_2, respectively, and they are also referred in $tr(S_1)$ and $tr(S_2)$, respectively. In addition, P_3, P_4 are processes that represent transitions toward states S_3 and S_4, respectively.

4 A Case Study

In this section, we demonstrate our translation rules by translating the state machine diagram of a simple user certification system illustrated in Fig. 3 into CSP#. In the following, a process translated from a state, e.g., labeled with "name", is written as "*name()*".

First, let *System()* be the process that represents the whole behavior of the state machine in the diagram. The behavior of the state machine is the behavior that follows the top level initial pseudostate, so the process is written as follows.

$$System() = TopLevelInit();$$

where, *TopLevelInit()* represents the process translated from the top level initial pseudostate. According to Table 1, the initial pseudostate and the final state at the top level are translated as follows.

$$\begin{aligned}
TopLevelInit() &= idle(); \\
TopLevelFinal() &= Skip;
\end{aligned}$$

Next, we show translations for the atomic states "idle", "initializing" and "diagnosing". The idle and diagnosing state do not have *entry*/*do*/*exit* behavior. So, the processes that represent these behaviors of the two states are *Skip*. In addition, since "$Skip; P = P$" is defined in CSP#, the translation results of them can be simply expressed as follows.

$$\begin{aligned}
idle() &= (StartUp \rightarrow initializing()) \,\Box\, (TurnOff \rightarrow TopLevelFinal()); \\
diagnosing() &= (SystemOK \rightarrow operating()) \,\Box\, (SystemNG \rightarrow idle());
\end{aligned}$$

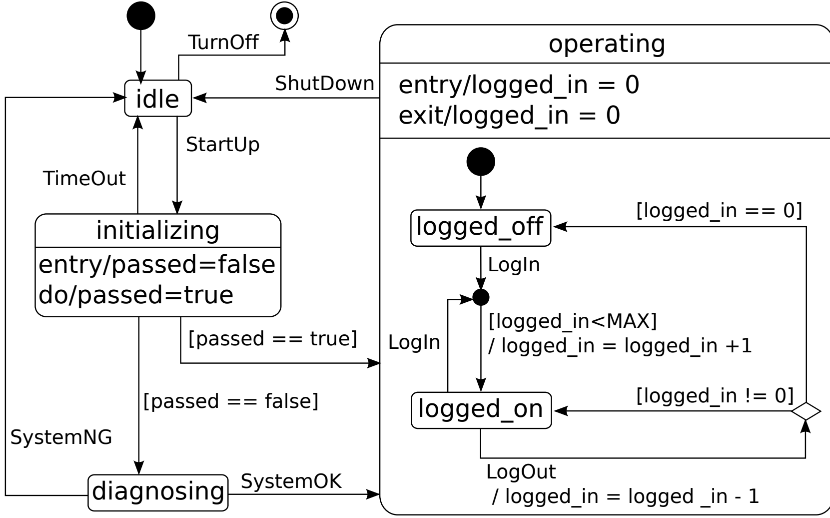

Fig. 3. State Machine Diagram for an Example User-Certification System

On the other hand, the initializing state has *entry/do* behavior that change the value of the variable *passed*. In our translation rules, these behavior is considered as action sequences and is translated to event sequences of CSP#. An event in CSP# is allowed to be attached with a sequential program. For this initializing state, its *entry* behavior is translated into "$\{passed = false\} \rightarrow Skip$", and its *exit* behavior is translated into "$\{passed = true\} \rightarrow Skip$". In addition, the variable *passed* used in these behaviors should also be declared as parts of the translation. Finally, the initializing state (i.e., process) is translated as follows.

$$\begin{array}{l} \textit{// Declaration of Shared Variable} \\ \textit{var passed} = \textit{false;} \\ \\ \textit{initializing()} = \{passed = false\} \rightarrow (\ (\{passed = true\} \rightarrow Stop) \\ \qquad \triangle\ (\ (TimeOut \rightarrow idle())\ \square\ ([passed == true]\ operating()) \\ \qquad\quad \square\ ([passed == false]\ diagnosing())\ )\ ); \end{array}$$

However, when applying previous translation rules proposed in [1], this initializing state is translated as follows.

$$\begin{array}{l} \textit{// When applying previous translation rules} \\ \textit{initializing()} = \underline{(\ \{passed = false\} \rightarrow \{passed = true\} \rightarrow Skip)} \\ \qquad \triangle\ (\ (TimeOut \rightarrow idle())\ \square\ ([passed == true]\ operating()) \\ \qquad\quad \square\ ([passed == false]\ diagnosing())\ ); \end{array}$$

In this translation by previous rules in [1], the entry behavior process of initializing state, $\{passed = true\}$, is included in the region where an interrupt

from an outgoing transition process is caught, and *Stop* in the translation by our rules is replaced with *Skip*. Adopting this previous translation, the process *System()* which represents the whole system's behavior can receive the following event sequences and then terminate successfully.

$$StartUp \rightarrow TimeOut \rightarrow TurnOff \rightarrow Skip$$
$$StartUp \rightarrow \{passed = false\} \rightarrow \{passed = true\} \rightarrow Skip$$

The first sequence means that the user certification system can receive the *TimeOut* event before the entry behavior of the initializing state is executed. On the other hand, the second sequence means that the system can be terminated successfully in the initializing state without firing no outgoing transition. However, we think that these behaviours are inappropriate as the system behaviors according to the definition of the SysML state machine diagrams. That is, we believe that the translated process of the system should not receive the event sequences. Using our translation, there is no possible that they are received.

From now on, we describe the translation of the composite state "operating", which has a region with junction and choice pseudostates. The composite state has an *entry* behavior, and this behavior must be executed before the internal state transition. After the *entry* behavior completes, the state transition started from the initial pseudostate in the internal region is performed. The composite state also has an *exit* behavior, and the behavior is executed when the outgoing transition to the idle state is fired by event *ShutDown*. This composite state is translated as follows, where *operatingSubInit()* is the process translated from the internal initial pseudostate.

```
// Composite State
var logged_in = 0;
operating() = {logged_in = 0}
    → ( (operatingSubInit() ||| Stop)
              △ (ShutDown → {logged_in = 0} → idle()) );
```

The two sub-states "logged_off" and "logged_on" are associated with the transitions with junction and choice pseudostates. No translation rule for states and transitions associated with these pseudostates is proposed in [1]. However, we can translate them by using our translation rules added in this paper.

The junction pseudostate in the composite state groups two transitions, from logged_off to logged_on and from logged_on to logged_on. These two transitions have *LogIn* event as a trigger and the boolean expression "*logged_in* < *MAX*" as a guard condition. Both of the original unwound transitions are translated as follows.

```
// for junction pseudostate
[logged_in < MAX] (LogIn → {logged_in = logged_in + 1} → logged_on())
```

The choice pseudostate has an incoming transition whose effects decrement the value of variable *logged_in* by 1, and two transitions whose guard condition

compares *logged_in* with 0. For a choice pseudostate, as mentioned earlier, the guard condition of an outgoing transition is evaluated after the effect of the incoming transition is performed. Therefore, the incoming and outgoing parts of the choice pseudostate are translated as follows, respectively.

```
// incoming part of choice pseudostate
LogOut → {logged_in = logged_in − 1} → Choice()

// outgoing part of choice pseudostate
Choice() = ([logged_in == 0] logged_off()) □ ([logged_in != 0] logged_on())
```

From the above, the internal state transition in the composite state operating is translated as follows.

```
// for the region in "operating" composite state
operatingSubInit() = logged_off();

logged_off() =
    [logged_in < MAX] (LogIn → {logged_in = logged_in + 1} → logged_on());

logged_on() =
    ( [logged_in < MAX] (LogIn → {logged_in = logged_in + 1} → logged_on()) )
    □ (LogOut → {logged_in = logged_in − 1} → Choice());

// outgoing part of choice pseudostate
Choice() = ([logged_in == 0] logged_off()) □ ([logged_in != 0] logged_on());
```

The CSP# descriptions obtained as above can be used as an input model for the model checking tool PAT [8]. In the rest of this section, we describe about model checking of the translated descriptions (model) with PAT.

We consider the following three properties as requirements that should be satisfied by the state machine of Fig. 3.

1. The state machine is deadlock free.
2. The state machine can reach the state in which the condition, "$logged_in > 0$", is satisfied.
3. The value of the variable *logged_in* always satisfies "$0 \leq logged_in \leq MAX$".

PAT has built-in functions for checking deadlock-freeness and reachability for a CSP# process. For properties 1 and 2, they are written as simple assertions of CSP# as follows, and they can be checked by evaluating these assertions with PAT.

```
#assert System() deadlockfree;                       (for Property 1)

#define logged_on_prop (logged_in > 0);      }
#assert System() reaches logged_on_prop;     }      (for Property 2)
```

Property 3 is expressed in LTL as follows.

$$\Box((logged_in \geq 0) \land (logged_in \leq MAX))$$

In order to check whether the state machine satisfies this LTL property, the following assertion is defined and evaluated.

#define user_num_prop ((logged_in >= 0) && (logged_in <= MAX));
#assert System() | = []user_num_prop;

When these three assertions are evaluated with PAT, all properties are evaluated as valid. Finally, we consider to check whether the system satisfies the following property 4.

4. When the event *LogIn* is received, the value of the variable *passed* is inevitably true.

Property 4 is defined as an assertion as follows and evaluated with PAT.

#define passed_prop (passed == true);
#assert System() | = [](LogIn → passed_prop);

When the assertion is evaluated with PAT, the property is evaluated as "NOT valid", and the following event sequence is given as a counter example.

$$StartUp \rightarrow \tau \rightarrow SystemOK \rightarrow \{logged_in = 0\} \rightarrow LogIn$$

where τ represents an event not labeled. This sequence represents a path to reach the logged_off sub-state through the diagnosing state from the initializing state and to receive the event *LogIn* at the logged_off sub-state. If the system behaves along the path, the value of *passed* is false until *LogIn* event is received. Therefore we can make sure that the system does not satisfy the property 4.

The processes that can be model-checked with PAT are not limited to the top level process *System()* that represents behavior of the whole state machine. For instance, if the process *System()* in the above assertions is replaced with the process *operatingSubInit()*, the internal state transition of the composite state could be model checked. Since the initial pseudostate in each region is translated into a CSP# process in our translation rules, such operations can be performed easier. When invariant properties (i.e., properties that should be satisfied through the whole system) are to be checked, checking them from the internal state transitions could help achieve early detection of defects that may be possibly lurk in deep nests. The translation rules given in this paper have the advantage that such hierarchical checking can be handled flexibly.

5 Conclusions and Future Work

In this paper, we described translation rules from SysML state machine diagrams to CSP# processes. This translation allows formal verification of the correctness of SysML state machine diagrams. We modified some translation rules proposed in [1], which we believe are not appropriate according to the formal definitions of SysML state machines. As a result, model checking of SysML state machine diagrams has become more accurate against the definition of the diagrams.

In addition, we added translation rules for two elements – junction pseudostates and choice pseudostates, which are not handled in [1]. These rules have contributed to increasing component types which are targets of model checking, and then,the coverage of verifiable diagrams has extended. Moreover, we conducted a case study to demonstrate the actual translation, and then, we showed some model checking results with PAT. These model checking results showed that, compared with the case where the previous translation rules in [1], verification of SysML state machine diagrams are more accurate when our translation rules are used.

Future Work. Regarding future work, we consider that it is important to deal with the followings. In order to expand the coverage of our methods, the behavior with message communication across multiple state machine diagrams should be handled. In addition, it is necessary to develop translation rules for other types of SysML diagrams such as block diagrams, parametric diagrams, and sequence diagrams, etc. Moreover, we plan to implement our translation rules and integrate the implementation with a web-based model driven development (MDD) tool Clooca [23], by which SysML diagrams including state machine diagrams can be graphically developed. Clooca is used currently for education purpose mainly. We have a vision that we will be able to perform a series of tasks like the followings on the Web: 1) draw SysML diagrams with Clooca, 2) translate the diagrams and generate processes in CSP#, and 3) apply model checking to check correctness of the diagrams. We expect that we can offer higher quality education for the MDD formalization and model checking by this combination of our transition with Clooca.

Acknowledgment. This research is conducted as a program for the "Regional Innovation Strategy Support Program 2012" by Ministry of Education, Culture, Sports, Science and Technology (MEXT), Japan.

References

1. Zhang, S.J., Liu, Y.: An Automatic Approach to Model Checking UML State Machines. In: IEEE International Conference on Secure Software Integration and Reliability Improvement Companion, pp. 1–6 (2010)
2. OMG: OMG Systems Modeling Language Version 1.3 (June 2012), `http://www.omg.org/spec/SysML/1.3/PDF`
3. OMG: OMG Unified Modeling Language Superstructure Version 2.4.1 (August 2011), `http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF`
4. Edmund, M., Clarke, J., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
5. Holzmann, G.: The Model Checker Spin. IEEE Trans. 23(5), 279–295 (1997)
6. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NUSMV: A New Symbolic Model Verifier. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 495–499. Springer, Heidelberg (1999)
7. Larsen, K.G., Pettersson, P., Yi, W.: Model-Checking for Real-Time Systems. In: Reichel, H. (ed.) FCT 1995. LNCS, vol. 965, pp. 62–88. Springer, Heidelberg (1995)

8. Sun, J., Liu, Y., Dong, J.S.: Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In: Margaria, T., Steffen, B. (eds.) ISoLA 2008. CCIS, vol. 17, pp. 307–322. Springer, Heidelberg (2008)
9. Hoare, C.A.R.: Communicating Sequential Processes. Commun. ACM 21(8), 666–677 (1978)
10. Bhaduri, P., Ramesh, S.: Model Checking of Statechart Models: Survey and Research Directions. CoRR cs.SE/0407038 (2004)
11. McMillan, K.L.: Symbolic Model Checking: An approach to the state explosion problem. PhD thesis, Pittsburgh, PA, USA (1992)
12. The Formal Systems website: FDR2.91 (November 2010), `http://www.fsel.com/`
13. Latella, D., Majzik, I., Massink, M.: Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. Formal Asp. Comput. 11(6), 637–664 (1999)
14. Lilius, J., Paltor, I.P.: vUML: A Tool for Verifying UML Models, 255–258 (1999)
15. Clarke, E.M., Heinle, W.: Modular Translation of Statecharts to SMV. Technical report (2000)
16. Dubrovin, J., Junttila, T., Högskolan, T., Dubrovin, J., Junttila, T., Dubrovin, C.J., Junttila, T.: Symbolic Model Checking of Hierarchical UML State Machines. Technical report, Helsinki University of Technology Laboratory (2007)
17. Niewiadomski, A., Penczek, W., Szreter, M.: A New Approach to Model Checking of UML State Machines. Fundam. Inf. 93(1-3), 289–303 (2009)
18. Hansen, H.H., Ketema, J., Luttik, B., Mousavi, M.R., van de Pol, J.C.: Towards Model Checking Executable UML Specifications In MCRL2. Innovations in Systems and Software Engineering 6, 83–90 (2010)
19. Rasch, H., Wehrheim, H.: Checking Consistency in UML Diagrams: Classes and State Machines. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 229–243. Springer, Heidelberg (2003)
20. Ng, M.Y., Butler, M.: Towards Formalizing UML State Diagrams in CSP. In: 1st International Conference on Software Engineering and Formal Methods, pp. 138–147. IEEE Computer Society (2003)
21. Groote, J.F., Mathijssen, A., Reniers, M., Usenko, Y., Weerdenburg, M.V.: The Formal Specification Language mCRL2. In: Proceedings of the Dagstuhl Seminar. MIT Press (2007)
22. Baeten, J.C.M., Weijland, W.P.: Process Algebra. Cambridge University Press (1990)
23. Technical Rockstars: clooca, `http://www.clooca.com`