Formalizing Non-Concurrent UML State Machines Using Colored Petri Nets

Étienne André, Christine Choppy, Kais Klai
LIPN, CNRS UMR 7030, Université Paris 13, France
e-mail: {first.last}@lipn.univ-paris13.fr

Abstract

UML state machines are an interesting graphical language to express dynamic systems behavior. However, using the different features available (hierarchy, internal/external transitions, entry/exit/do activities, history pseudostates, etc.) may yield quite complex behaviors that are difficult to inspect and check visually. We introduce an algorithm to automatically generate a colored Petri net model associated with a state machine description, so as to provide a formal specification. In this proposal, although we do not consider concurrent aspects (such as fork and join), we take into account all the above mentioned features in a thorough and integrated way. This is illustrated on some examples.

1 Introduction

UML behavioral state machines are transition systems used to express the behavior of dynamic systems [10]. They are a variant of Harel's statecharts [4]. Although UML is widely used in the industry, its semantics is not formally expressed, hence not directly suitable for formal methods, or for defining bisimulations with other formalisms. However, a formal semantics can be given, for instance by translation to a formalism, and we chose here to use Colored Petri Nets (CPNs) [6] in order to have a detailed view of the process with a graphical representation, and benefit from powerful tools to test and check our model.

In this paper, we describe a translation into CPNs of UML state machines diagrams (SMDs) where no concurrent behavior occurs. Thus, synchronization of events, fork and join pseudostates, are discarded. However, we do consider history pseudostates, do/entry/exit behaviors, hierarchy of machines with inter-level transitions, and variables appearing in guards and behaviors. In that case, at any time, one and only one simple state is active. Hence, a single token in the CPN can both show the current active state, and contain the value of possible "global" variables and history pseudostates.

Our motivation as regards to non-concurrency is twofold. First, numerous cases of SMDs with no concurrency are often used (see, e.g., [10, 13]), and it is hence interesting to define an approach dealing with non-concurrent SMDs. Second, while concentrating on this and developing a quite thorough model, this gives us the opportunity to present aspects of the translation from SMDs to CPNs in a simpler way than in the concurrent framework. Extending this scheme to the concurrent case will be the next step of this work, and is not trivial for some aspects (see Section 4).

Related Work. Verification of SMDs has been often tackled (see, e.g., [2] for a survey). Some approaches directly give UML a semantics. The closest to the set of syntactic constructs we consider here is [7], where entry/exit behaviors, activities, synchronization and history states are considered; however, global variables are discarded, and no model checking is performed. Many approaches translate UML specification into an intermediate model of some model checker, e.g., SMV [9] or SPIN [5]. Most approaches consider quite restrictive subsets of the UML syntax as defined by the OMG [10]. An automated translation from SMDs into CSP# (an extension of CSP) is proposed in [13]; modeling techniques such as use of data structures, join/fork, history pseudostates, entry/exit behaviors (but with no use of variable inside) are considered, and properties are checked using PAT [12]. Also note that these formalisms do not provide a graphical representation, in contrast to CPNs.

Other approaches use CPNs as an intermediate formal model. In [11, 8], conversions from SMDs to CPNs are considered, using CPN Tools to analyze the generated CPN. The work in [11] aims at providing a systematic and seamless integration of CPNs with object-oriented software architectures. In contrast, our translation technique aims at defining automated and universal conversion rules within a stand-alone analysis approach. [8] uses an intermediate model for composite states (which we do not), and performs the analysis mainly using simulation. Their analysis deals with a query language whose expression power is limited comparing to LTL or CTL logics which we could use (see Section 4).

Our approach is in line with the technique presented in [3] where we propose a formalization of SMDs using CPNs. The SMDs considered in [3] include synchronization, limited aspects of hierarchy, join and fork (with no inter-level transitions) but history pseudostates and variables are not considered. Here, although we discard (in a first step) the concurrent aspects, our aim is to consider many syntactic features of SMDs all together, i.e., state hierarchy with entry/exit/do behaviors involving variables, history pseudostates, and inter-level transitions.

We recall SMDs and CPNs (Section 2), describe our translation together with examples (Section 3), and give hints for extension to the concurrent case (Section 4). Fully detailed definitions, functions and algorithms can be found in [1].

2 Preliminaries

2.1 UML State Machine Diagrams

We introduce the SMD concepts considered here with an example (depicted in Figure 1) used throughout the paper.

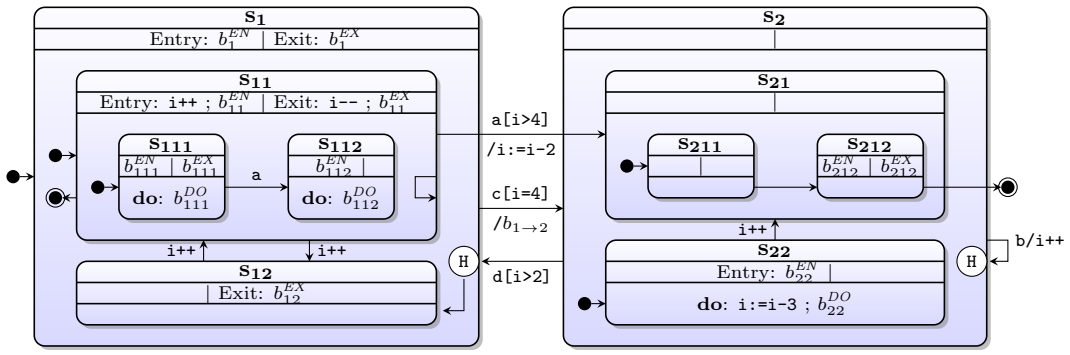

Figure 1: Example of state machine diagram

We devised this example as an SMD composed with several simple and composite states. Some states (e.g., $\mathbf{s_1}$, $\mathbf{s_{11}}$) have entry and/or exit behaviors (or activities), and some ($\mathbf{s_{111}}$, $\mathbf{s_{112}}$, $\mathbf{s_{22}}$) have a do behavior. Recall that entry/exit/do behaviors are behaviors that are performed when entering/exiting/being in a state, respectively. Transitions may involve events, guards and behaviors, where a (global) variable `i` appears. Both $\mathbf{s_1}$ and $\mathbf{s_2}$ have a history pseudostate. Within a state $\mathbf{s_n}$, entry/exit/do behaviors are denoted $b_n^{EN}/b_n^{EX}/b_n^{DO}$ respectively.

We now detail the subset of the SMD syntax considered in our translation.

States. We consider two kinds of states: simple and composite. A composite state is a state that contains other states, allowing to construct hierarchical SMDs. For sake of simplicity, we discard here submachine states, "semantically equivalent to a composite state", [10, Section 15.3.11, p.551]. "Any state enclosed within a region of a composite state is called a *substate* of that composite state. It is called a *direct substate* when it is not contained by any other state; otherwise, it is referred to as an *indirect substate*." [10, Section 15.3.11, p.551]. Given a composite state $\mathbf{s}$, we denote by $SubStates(\mathbf{s})$ the set of direct substates of $\mathbf{s}$ (including final states), and by $SubStates^*(\mathbf{s})$ all substates of $\mathbf{s}$, both direct and indirect. If $\mathbf{s}$ is not contained in any state, we call it a *root* state. If not, $parent(\mathbf{s})$ denotes the state containing $\mathbf{s}$, and $ancestors(\mathbf{s})$ is the transitive closure of $parent$.

Behaviors. Behaviors (or activities) are allowed when entering, exiting a state, as a "do" behavior, or when performing a transition. For sake of a trade-off between conciseness and exhaustiveness, we abstract behaviors using a name b (corresponding to the actual behavior), and a function f expressing the changes induced on the variables of the system. The behavior will be denoted by (b, f). If only the name is defined, we set $f = id$ (with id the identity function), and if only the function is defined, we set $b = none$. If a state has an entry behavior $(b, f) = (none, id)$, we assume that this state has no entry behavior (and similarly for exit/do behaviors). For example, in Figure 1, the "do" behavior of $\mathbf{s_{112}}$ is (b_{112}^{DO}, id), whereas the behavior associated with transition from $\mathbf{s_{11}}$ to $\mathbf{s_{21}}$ is $(none, f : i \to i-2)$. Modeling more complex behaviors (sequences of behaviors, branching depending on the variables, etc.) could be very easily performed within our translation scheme, possibly using composite places [6].

As for the "do" behaviors, we make the following two assumptions: (i) "do" behaviors are atomic behaviors that can be executed as many times as wished (including 0); (ii) only simple states can have a "do" behavior (although composite states with a "do" behavior could also be considered).

Initial Pseudostates. We require that each composite state contains one and only one initial pseudostate, which has one and only one outgoing transition. Given a composite state $\mathbf{s}$ whose initial state points to $\mathbf{s_0}$, we write $\mathbf{s_0} = init(\mathbf{s})$. By extension, given an SMD S whose (root) initial state points to $\mathbf{s_0}$, we write $\mathbf{s_0} = init(S)$. We also consider that the active state of the system cannot be an initial pseudostate. Note that this is a modeling choice only: if one wants to model an SMD where the system can *stay* in an initial pseudostate, it suffices to add another state between the initial pseudostate and its immediate successor.

History Pseudostates. An interesting feature of SMDs, not often tackled in the literature, is the notion of history pseudostates (`H` construct). "If the transition terminates on a shallow history pseudostate, the active substate becomes the most recently active substate prior to this entry, unless the most recently active substate is the final state or if this is the first entry into this state." [10, Section 15.3.11, p.555] Given a composite state $\mathbf{s}$, we denote by $\mathbf{s^H}$ its history pseudostate, if any. We write $\mathbf{s^H}.dh$ for the default entry of $\mathbf{s^H}$ (depicted using an arrow exiting from $\mathbf{s^H}$), if any. We restrict ourselves to shallow history pseudostates (`H`), but our scheme could be very easily adapted to the deep history pseudostates (`H*`).

Transitions. A transition can have a guard, a synchronization event, and a behavior; transitions can have as source and destination any (composite or simple) state, with some restrictions (e.g., a transition from an initial state cannot have a behavior, etc.). Exiting a composite state machine through an event results "in exiting of all the substates of the composite state executing their exit activities starting with the innermost states in the active state configuration" [10, Section 15.3.13, p.574]. "A transition to the enclosing state represents a transition to the initial pseudostate in each region." [10, Section 15.3.11, p.551]. For example, in Figure 1, the target of the transition from $\mathbf{s_1}$ to $\mathbf{s_2}$ is actually $\mathbf{s_{22}}$.

For a given composite state $\mathbf{s}$, we differentiate between external and internal transitions. Trivially, a transition is said to be external w.r.t. $\mathbf{s}$ if its source is a state outside $\mathbf{s}$. A transition from a substate of $\mathbf{s}$ is not necessarily internal: graphically, if the arrow linking the source to the global composite state remains within the composite state (see self-loop inside $\mathbf{s_{11}}$ in Figure 1), then it is internal. Otherwise, it is external (see self-loop outside $\mathbf{s_2}$ in Figure 1). The entry and exit behavior should be executed only for external transitions. In Figure 1, b_{11}^{EN} will not be executed in the self-loop in $\mathbf{s_{11}}$. We say that a transition is *classical* if its source is not an initial pseudostate, and not a history pseudostate.

Final States. In each composite state, we allow 0 or 1 final state, since "each region of a composite state may have [...] a final state." [10, Section 15.3.11, p.551] (Note that our scheme would perfectly allow the case with more than 1 final state.) Given a composite state $\mathbf{s}$, we denote by $\mathbf{s^F}$ its final state, if any. By extension, given an SMD S, we denote by S^F its final state (hence the only root final state), if any.

As for transitions, it is clear that, when in the final state in a given composite state $\mathbf{s}$, one can still perform an outgoing transition from $\mathbf{s}$. It also seems clear that an external self-transition on $\mathbf{s}$ can be performed. However, although it is not explicit in [10], we believe that, considering the semantics of internal transitions and of final states, an internal self-transition on $\mathbf{s}$ cannot be performed when the active state of the system is $\mathbf{s^F}$.

Variables. We allow any kind of variables in any behavior and transition guard. Such variables (integers, lists, etc.) are often met in practice [10].

Other Constructs. We do not consider choice pseudostates, terminate states, entry and exit points, deferred events and timed aspects. The 3 former are easy to consider; the 2 latter are subject of ongoing work.

Definition 1 *A State Machine Diagram is a tuple* $SMD = (\mathcal{S}, \mathcal{B}, \mathcal{E}, \mathcal{V}, \mathcal{P}, \mathcal{N}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{T})$ *where:*
1. $\mathcal{S}$ *is a set of states,*
2. $\mathcal{B}$ *is a set of behaviors,*
3. $\mathcal{E}$ *is a set of events,*
4. $\mathcal{V} = \{V_1, \ldots, V_{N_V}\}$ *is a set of variables,*
5. $\mathcal{P} : \mathcal{S} \to 2^{Pr}$ *associates with each state a set of properties within* $Pr = \{isSimple, hasHistory, hasFinal, isInit\}$,
6. $\mathcal{N} : \mathcal{S} \to ((\mathcal{B} \cup none) \times \mathcal{F})$ *associates with each state an entry behavior (or possibly none), where* $\mathcal{F}$ *is the set of functions on the variables, and similarly for* $\mathcal{X}$ *(exit) and* $\mathcal{D}$ *(do) behaviors,*
7. $\mathcal{C} : \mathcal{S} \to 2^{\mathcal{S}}$ *associates with each state its direct substates,*
8. $\mathcal{T}$ *is a set of transitions* $\tau = (\mathbf{s_1}, g, e, (b, f), ext, \mathbf{s_2})$*, where* $\mathbf{s_1}, \mathbf{s_2} \in \mathcal{S}$ *are the source and target states respectively,* g *is the guard,* $e \in \mathcal{E}$*,* $(b, f) \in ((\mathcal{B} \cup none) \times \mathcal{F})$ *is the behavior to be executed while firing the transition, and* $ext \in \mathbb{B}$ *(for external) is a boolean always equal to* T *(true), except if the transition is internal, in which case it is equal to* F *(false). Observe that if the transition is internal, then either* $\mathbf{s_1} \in SubStates^*(\mathbf{s_2})$ *or* $\mathbf{s_2} \in SubStates^*(\mathbf{s_1})$.

Given $\mathbf{s} \in \mathcal{S}$, if $\mathcal{N}(\mathbf{s}) = (none, id)$, then we say that $\mathbf{s}$ has no entry behavior (and similarly for exit/do behaviors).

Observe that relations *SubStates* and $SubStates^*$ can be formally defined using $\mathcal{C}$.

2.2 Colored Petri Nets

In the following, we refer to *Expr* as the set of expressions provided by the net inscription language (net inscriptions are arc expressions, guards, color sets and initial markings), and to $Expr(V)$ as the set of expressions $e \in Expr$ such that $Var[e] \subseteq V$ (where $Var[e]$ denotes the set of variables occurring in e). Note that, in order to allow model checking, we add to the classical definition of CPNs a set of labels *Labels* and a labeling function L to both places and transitions.

Definition 2 *A* colored Petri net *(CPN) [6] is a tuple* $CPN = (P, T, A, Labels, B, V, C, G, E, I, L)$ *such that :*
1. P *is a finite set of* places,
2. T *is a finite set of* transitions *such that* $P \cap T = \emptyset$,
3. $A \subseteq P \times T \cup T \times P$ *is a set of directed* arcs,
4. Labels is a finite set of labels,
5. B *is a finite set of non empty* color sets *(types),*
6. V *is a finite set of* typed variables *such that* $\forall v \in V, Type[v] \in B$,
7. $C : P \to B$ *is a* color set function *assigning a color set to each place,*
8. $G : T \to Expr(V)$ *is a* guard function *assigning a guard to each transition such that* $Type(G(t)) = \mathbb{B}$*, and* $Var[G(t)] \subseteq V$,
9. $E : A \to Expr(V)$ *is an* arc expression function *assigning an arc expression to each arc such that* $Type(E(a)) = C(p)_{MS}$*, where* p *is the place connected to the arc* a*, and MS denotes the multiset,*
10. $I : A \to Expr(V)$ *is an* initialization function *assigning an initial marking to each place such that* $Type(I(p)) = C(p)_{MS}$*, and*
11. $L : P \cup T \to 2^{Labels}$ *is a* labeling function *assigning a set of labels to each place and transition.*

In our algorithms, given $x, y \in P \cup T$, $x \xrightarrow{E(x,y)} y$ denotes the arc $(x, y) \in A$; and, given $t \in T$, $t.guard$ denotes $G(t)$.

3 Translation of State Machines

3.1 General Scheme

We consider in the rest of the paper (including in our algorithms) the SMD $SMD = (\mathcal{S}, \mathcal{B}, \mathcal{E}, \mathcal{V}, \mathcal{P}, \mathcal{N}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{T})$. We will build the CPN $(P, T, A, Labels, B, V, C, G, E, I, L)$ by defining and updating the elements of the tuple throughout this section.

We define a translation scheme where simple states (including final states) are translated into places, whereas behaviors (entry, exit, do) are translated into CPN transitions. Further places and transitions will also be defined.

A major issue when studying SMDs is the precise handling of entry and exit behaviors. Consider transition from $\mathbf{s_1}$ to $\mathbf{s_2}$ with event label `c`, guard `i=4` and behavior $b_{1\to 2}$ in Figure 1. Recall from Section 2.1 that this transition is actually equivalent to three transitions, from $\mathbf{s_{111}}$, $\mathbf{s_{112}}$ and $\mathbf{s_{12}}$ respectively, and with target $\mathbf{s_{22}}$. When in state $\mathbf{s_{111}}$, one has to perform the following sequence: b_{111}^{EX}; `i` $--$; b_{11}^{EX}; b_1^{EX}; $b_{1\to 2}$; b_{22}^{EN}; when in state b_{112}: `i` $--$; b_{11}^{EX}; b_1^{EX}; $b_{1\to 2}$; b_{22}^{EN}; and when in state b_{12}: b_{12}^{EX}; b_1^{EX}; $b_{1\to 2}$; b_{22}^{EN}, all of which apply when `i` $= 4$. Hence, it would be straightforward to translate the transition from $\mathbf{s_{111}}$ to $\mathbf{s_{22}}$ using 5 fresh CPN transitions, and the transition from $\mathbf{s_{112}}$ (resp. $\mathbf{s_{12}}$) to $\mathbf{s_{22}}$ using 4 other CPN transitions (plus a set of fresh places in between). This may quickly result in an explosion of the number of transitions corresponding to entry and exit behaviors, many of them corresponding to the *same* behavior. For sake of readability, maintainability, and size of the translated CPN, this should be factored. Hence, we propose a scheme such that each behavior is encoded into only one CPN transition. Since many SMD transitions go through the same behaviors, we need a "memory" mechanism to remember from which place we originate so as to find the correct target. So we propose to add a control place. Note that this does not lead to a state space explosion, since only transitions synchronized using this control place can be fired.

3.2 Typing

Apart from some tokens used for synchronization, the CPN will contain only one type of token. This main type will be a tuple made of the variables and the history information.

Formally, let $\mathcal{V} = \{v_i, 1 \leq i \leq |\mathcal{V}|\}$ be the variables used within *SMD*. Each variable v_i has type $\mathcal{V}_i$. Let $\mathcal{H} = \{\mathbf{s_{x_i}}, 1 \leq i \leq |\mathcal{H}|\}$ be the composite states containing a history pseudostate (as a direct substate). We define:

$$B_{\mathcal{V}} = \underset{1\leq i\leq|\mathcal{V}|}{\times} \mathcal{V}_i \quad \text{and} \quad B_{\mathcal{H}} = \underset{1\leq i\leq|\mathcal{H}|}{\times} \mathit{SubStates}(\mathbf{s_{x_i}}).$$

The type of the main token is $(B_{\mathcal{V}} \times B_{\mathcal{H}})$. Inscriptions on most CPN arcs are of the form $(v, h) : (B_{\mathcal{V}} \times B_{\mathcal{H}})$. When clear from the context, we write vh for (v, h). Given a function $f : \mathcal{F}$, we write $f_{\mathcal{V}}(vh)$ for $(f(v), h)$. Other places and arcs will have a *null* type $\bullet$. Hence, $B = \{(B_{\mathcal{V}} \times B_{\mathcal{H}}), \bullet\}$.

Initial value of global variables can be either arbitrary, or undefined (using a special value), or assigned by the designer. As for history pseudostates, we stick to the specification and, given a history construct in a composite state $\mathbf{s}$, if a default history pseudostate $\mathbf{s^H}.dh$ is defined, we initialize the corresponding history variable to $\mathbf{s^H}.dh$. Otherwise, we initialize it to $\mathit{init}(\mathbf{s})$.

Example. In Figure 1, variable i is of type integer; we have a history pseudostate in state $\mathbf{s_1}$ (resp. $\mathbf{s_2}$), with possible values $\{\mathbf{s_{11}}, \mathbf{s_{12}}, \mathbf{s_1^F}\}$ (resp. $\{\mathbf{s_{21}}, \mathbf{s_{22}}\}$). Hence, the type of the token is $(\mathbb{N} \times \{\mathbf{s_{11}}, \mathbf{s_{12}}, \mathbf{s_1^F}\} \times \{\mathbf{s_{21}}, \mathbf{s_{22}}\})$. As for the initial value of the token, we arbitrarily set i to 0; h_1 is set to $\mathbf{s_{12}}$ because it is the default entry of the `H` of $\mathbf{s_1}$, and h_2 is set to $\mathit{init}(\mathbf{s_2}) = \mathbf{s_{22}}$ – hence $(0, \mathbf{s_{12}}, \mathbf{s_{22}})$.

3.3 States and Behaviors

We first translate the purely hierarchical structure of the SMD, so that to get a tree of entry and exit behaviors, that will be used later when connecting transitions. We translate each simple state into a place, and each behavior into a transition (and its associated target place). We then connect the entry (resp. exit) behaviors together, so that we get a tree of entry (resp. exit) behaviors. We also connect the places corresponding to simple states with their "do" behavior, if any. Connecting the entry/exit behaviors with the places corresponding to simple states depends on the transitions and will be done in Section 3.4.

Places. We assume a function p, which associates a place $p(\mathbf{s})$ with each state $\mathbf{s}$, and a place $p(b)$ to each behavior b. We assume a function t, which associates a transition $t(b)$ with each behavior b. "The operation $\mathit{LCA}(\mathbf{s_1}, \mathbf{s_2})$ returns an orthogonal state or region that is the Least Common Ancestor of states $\mathbf{s_1}$ and $\mathbf{s_2}$, based on the statemachine containment hierarchy." [10, Section 15.3.12, p.565] In the SMD of Figure 1, we have $\mathit{LCA}(\mathbf{s_{111}}, \mathbf{s_{12}}) = \mathbf{s_1}$, and $\mathit{LCA}(\mathbf{s_1}, \mathbf{s_{21}}) = \mathit{undef}$.

Transitions. We define below two useful functions. Given a state $\mathbf{s}$, function *SubEN* returns the transitions corresponding either to the entry behavior of $\mathbf{s}$ or, if none, to the entry behaviors of its (possibly transitive) substates, if any.

$$\mathit{SubEN}(\mathbf{s}) = \begin{cases} \{t(b^{EN})\} & \text{if } \mathbf{s} \text{ has an entry behavior } b^{EN} \\ \{\} & \text{if } \mathbf{s} \text{ is simple with no entry behavior} \\ \bigcup_{\mathbf{s'}\in \mathit{SubStates}(\mathbf{s})}(\mathit{SubEN}(\mathbf{s'})) & \text{otherwise.} \end{cases}$$

For example, in Figure 1, we have $\mathit{SubEN}(\mathbf{s_1}) = \{t(b_1^{EN})\}$, and $\mathit{SubEN}(\mathbf{s_2}) = \{t(b_{212}^{EN}), t(b_{22}^{EN})\}$.

Similarly, given a state $\mathbf{s}$, function *SupEX* returns the place corresponding either to the exit behavior of $\mathbf{s}$ or, if none, to the exit behavior of its closest ancestor with an exit behavior, if any; otherwise, it returns p_{out}, which represents a special place defined in Algorithm 1. For example, in Figure 1, we have $\mathit{SupEX}(\mathbf{s_1}) = p(b_1^{EX})$, $\mathit{SupEX}(\mathbf{s_{112}}) = p(b_{11}^{EX})$, and $\mathit{SupEX}(\mathbf{s_{211}}) = p_{out}$.

In Algorithm 1, we first add two special places p_{in} (resp. p_{out}), corresponding to the root of the tree of entry (resp. exit) behaviors (line 2). The first loop (lines 3–16) adds separately places and transitions corresponding to simple states and behaviors. For every simple state, we add a corresponding place (line 5), and connect it to the transition corresponding to its "do" behavior (lines 6–9), if any. If the state has an entry behavior, we add a transition corresponding to the behavior itself, and a place modeling that this behavior has been executed (line 11); we connect the transition to the place (line 12). If the state has an exit behavior, we add a transition corresponding to the behavior itself, and a place modeling that this behavior is about to be executed (line 14); we connect the place to the transition (line 15). We finally add a place corresponding to the history substate of the current state, if any (line 16).

The second loop (lines 18–27) connects together the places and transitions corresponding to the entry behaviors, so that

Algorithm 1: Encoding states and behaviors

1 $P \leftarrow \emptyset$; $T \leftarrow \emptyset$; $A \leftarrow \emptyset$
2 $P \leftarrow P \cup \{p_{in}, p_{out}\}$
3 **foreach** *state* $\mathbf{s} \in \mathcal{S}$ **do**
4 **if** $\mathbf{s}$ *is a simple state* **then**
5 $P \leftarrow P \cup \{p(\mathbf{s})\}$
6 **if** $\mathbf{s}$ *has a "do" behavior* (b^{DO}, f^{DO}) **then**
7 $T \leftarrow T \cup \{t(b^{DO})\}$
8 $A \leftarrow A \cup \{p(\mathbf{s}) \xrightarrow{vh} t(b^{DO}) \ ,$
9 $t(b^{DO}) \xrightarrow{f^{DO}(vh)} p(\mathbf{s})\}$
10 **if** $\mathbf{s}$ *has an entry behavior* (b^{EN}, f^{EN}) **then**
11 $P \leftarrow P \cup \{p(b^{EN})\}$; $T \leftarrow T \cup \{t(b^{EN})\}$
12 $A \leftarrow A \cup \{t(b^{EN}) \xrightarrow{f^{EN}(vh)} p(b^{EN})\}$
13 **if** $\mathbf{s}$ *has an exit behavior* (b^{EX}, f^{EX}) **then**
14 $P \leftarrow P \cup \{p(b^{EX})\}$; $T \leftarrow T \cup \{t(b^{EX})\}$
15 $A \leftarrow A \cup \{p(b^{EX}) \xrightarrow{vh} t(b^{EX})\}$
16 **if** $\mathbf{s}$ *has a direct history pseudostate* **then**
17 $P \leftarrow P \cup \{p(\mathbf{s^H})\}$
18 **foreach** *state* $\mathbf{s} \in \mathcal{S}$ **do**
19 **if** $\mathbf{s}$ *is a composite state with an entry behavior* (b^{EN}, f^{EN}) **then**
20 **foreach** *state* $\mathbf{s}' \in SubStates(\mathbf{s})$ **do**
21 **foreach** *transition* $t' \in SubEN(\mathbf{s}')$ **do**
22 $A \leftarrow A \cup \{p(b^{EN}) \xrightarrow{vh} t'\}$
23 **if** $\mathbf{s}$ *has an exit behavior* (b^{EX}, f^{EX}) **then**
24 **if** $\mathbf{s}$ *is root* **then**
25 $A \leftarrow A \cup \{t(b^{EX}) \xrightarrow{f^{EX}(vh)} p_{out}\}$
26 **else**
27 $A \leftarrow A \cup \{t(b^{EX}) \xrightarrow{f^{EX}(vh)} SupEX(\mathbf{s})\}$
28 **foreach** *root state* $\mathbf{s} \in \mathcal{S}$ **do**
29 **foreach** *transition* $t \in SubEN(\mathbf{s})$ **do**
30 $A \leftarrow A \cup \{p_{in} \xrightarrow{vh} t\}$

we get a tree structure, and similarly for the exit behaviors, except that the tree is "inverted" (the "root" has no successor but several predecessors, and so on). If the state is composite and has an entry behavior (lines 19–22), we connect the place corresponding to the entry behavior to each transition corresponding to an entry behavior of a substate. Similarly, if the state has an exit behavior, we connect the corresponding transition to the place corresponding to the exit behavior of its closest ancestor with an exit behavior, if any, or to p_{out} otherwise (lines 23–27).

Finally, the third loop (lines 28–29) connects p_{in} to the first entry behaviors met in the state machine hierarchy.

We give in Figure 2 the CPN resulting from Algorithm 1 applied to the SMD of Figure 1. We write vh for i, h_1, h_2

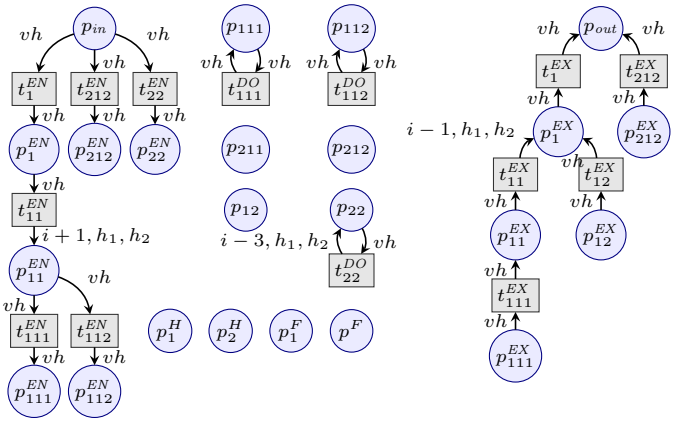

Figure 2: Application of Algorithm 1 to the SMD of Figure 1

and explicit it only when some value is modified. Observe that the resulting CPN may contain some "useless" information, in the sense that some places or transitions may be unreachable, or some arcs never used. Indeed, recall that Algorithm 1 does not depend on the transitions of the SMD, but only on its hierarchical structure. For instance, the arc from p_{11}^{EN} to t_{112}^{EN} will never be traveled, because no transition in the SMD allows one to enter $\mathbf{s_{112}}$ directly from outside $\mathbf{s_{11}}$ (i.e., requiring to execute b_{111}^{EN} and then b_{112}^{EN}).

3.4 Transitions

The places and transitions created so far only concern the hierarchy of state machines with their respective entry, do and exit behaviors. We now update the CPN to take into account the transitions of the SMD.

Auxiliary Functions. Let $SupEN(\mathbf{s})$ be the function which returns the place corresponding either to the entry behavior of $\mathbf{s}$ or, if none, to the entry behavior of its closest ancestor with an entry behavior, if any, and p_{in} otherwise. In the SMD Figure 1, $SupEN(\mathbf{s_1}) = p(b_1^{EN})$ and $SupEN(\mathbf{s_2}) = p_{in}$.

Given two states $\mathbf{s_1}, \mathbf{s_2}$ (with $\mathbf{s_2}$ simple and $\mathbf{s_2} \in SubStates^*(\mathbf{s_1})$) and $ext \in \mathbb{B}$, we define function $InFrom(\mathbf{s_1}, \mathbf{s_2}, ext)$ that returns the place preceding the first transition met in the entry behavior hierarchy from $\mathbf{s_1}$ to $\mathbf{s_2}$. If no such behavior exists, then it returns $p(\mathbf{s_2})$. In the case of an internal transition ($ext = F$), the hierarchy excludes $\mathbf{s_1}$. This is conform to the fact that an internal transition must not execute the containing composite state's entry/exit behaviors. For example, in the SMD of Figure 1, we have $InFrom(\mathbf{s_1}, \mathbf{s_{111}}, T) = p_{in}$, viz. the place preceding $t(b_1^{EN})$; $InFrom(\mathbf{s_1}, \mathbf{s_{111}}, F) = p(b_1^{EN})$, viz. the place preceding $t(b_{11}^{EN})$; and $InFrom(\mathbf{s_2}, \mathbf{s_{211}}, T) = p(\mathbf{s_{211}})$.

We also assume a function $init^*$ which, given a state $\mathbf{s}$, returns the initial simple state, hence $\mathbf{s}$ if it is a simple state, or the simple state at the bottom of the containment hierarchy otherwise. For example, in the SMD of Figure 1, we have $init^*(\mathbf{s_1}) = \mathbf{s_{111}}$ and $init^*(\mathbf{s_2}) = \mathbf{s_{22}}$. By extension, given an SMD SMD, we write $init^*(SMD)$ for $init^*(init(SMD))$.

Principle of the Translation (Algorithm 2). We trans-

late each classical transition $\tau = (\mathbf{s_1}, e, g, (b, f), ext, \mathbf{s_2}) \in \mathcal{T}$ as follows. Recall from Section 2 that the actual destination of the transition is $init^*(\mathbf{s_2})$ and that any substate of $\mathbf{s_1}$ can take this transition.

For each simple substate $\mathbf{s}$ of $\mathbf{s_1}$ (including $\mathbf{s_1}$ itself), we add a fresh transition t_τ. If the transition should execute some behavior (i.e., if $b \neq none$), we add an arc from t_τ to a fresh place $p(b)$, and an arc from $p(b)$ to another fresh transition $t(b)$. (Recall that all arcs have vh as an arc expression, unless otherwise specified.) This construct implies that the transition behavior will be performed right after the transition was fired. Then, we shall connect place $p(\mathbf{s})$ to transition t_τ, and two cases arise. (1) If no exit behavior should be performed for this particular transition (i.e., from $\mathbf{s}$ to $init^*(\mathbf{s_2})$ considering ext), then we simply add an arc from $p(\mathbf{s})$ to t_τ. (2) If a sequence of exit behaviors should be performed, we use a control place mechanism. We first add a fresh transition t_1, and a fresh place p_1 with type $\bullet$. We connect $p(\mathbf{s})$ to t_1; we connect t_1 to both p_1 (with $\bullet$ as an arc expression) and to the place corresponding to the first exit behavior that should be performed. Then, we connect the place following the last exit behavior that should be performed to t_τ, and we connect p_1 to t_τ (with $\bullet$ as an arc expression). This ensures that all exit behaviors will be performed in the correct order.

Now consider guard g. We have to add it to the first transition met after leaving $p(\mathbf{s})$: if no exit behavior should be performed for this transition, this is t_τ, and t_1 otherwise.

The connection between t_τ and $p(init^*(\mathbf{s_2}))$ is similar. If no entry behavior should be performed, we connect directly t_τ (or $t(b)$ if $b \neq none$) to $p(init^*(\mathbf{s_2}))$ Otherwise, we add a mechanism using a control place, so that the necessary entry behaviors are performed. As for the function f, we set $f(vh)$ as an arc expression for the first transition following t_τ (resp. $t(b)$ if $b \neq none$), so that it gets performed right after the transition was fired (resp. after b is performed).

Finally, we have to set the initial token of the CPN. If no entry behavior should be performed, we add the initial token to $init^*(S)$. Otherwise, we add a fresh place p_{start}, that will contain the initial token, and we add a mechanism using an extra control place to ensure that all necessary entry behaviors leading to $init^*(S)$ will be performed.

This fully formalized Algorithm 2 is available in [1].

Example. We give in Figure 3 a part of the CPN resulting from the application of Algorithms 1 and 2 to the SMD of Figure 1. We only consider the places and CPN transitions involved in the SMD transitions from $\mathbf{s_{112}}$ to $\mathbf{s_{211}}$ (event `a`) and from $\mathbf{s_{112}}$ to $\mathbf{s_{22}}$ (event `c`). We display with a thick border the places and transitions added by Algorithm 2. Observe that transitions t_1, a and place p_1 were added for modeling SMD transition from $\mathbf{s_{112}}$ to $\mathbf{s_{211}}$ (event `a`), whereas transitions t'_1, c, $t_{1\to2}$, t'_2 and places p'_1, $p_{1\to2}$, p'_2 were added for modeling SMD transition from $\mathbf{s_{112}}$ to $\mathbf{s_{22}}$ (event `c`). Observe that transition through `a` is a case with no entry behavior, hence the behavior transition a is directly connected to the destination place p_{211}, without synchronization.

Algorithm 3: Encoding history pseudostates

```
1  foreach composite state s containing one history
   pseudostate s^H do
2      Let h be the history variable corresponding to
       state s in the token
3      foreach simple s' ∈ ({s} ∪ SubStates*(s)) do
4          foreach CPN arc with target p(s') do
5              Update the arc inscription so that h ← s'
6      foreach non-final state s' ∈ SubStates(s) do
7          Let t be a fresh transition
8          T ← T ∪ {t}  ;  A ← A ∪ {p(s^H) --vh--> t}
9          if s has a final state s^F and s^H.dh is defined
           and s' = s^H.dh then
10             t.guard ← [h = s' or h = s^F]
11         else
12             t.guard ← [h = s']
13         s_d ← init*(s')
14         if InFrom(s, s_d, F) = p(s_d) then
15             A ← A ∪ {t --vh--> p(s')}
16         else
17             Connect t to SupEN(s') and then
               SupEN(s_d) using a synchronization
               mechanism (with a fresh t' and a fresh p')
```

3.5 Representation of History Pseudostates

In Section 3.2, we added a history type within the token. Then, for a given substate $\mathbf{s}$ within a state containing a history pseudostate construct, it is sufficient to update this type to $\mathbf{s}$ for each incoming transition. For sake of simplicity, we always update this type for any transition leading to, or internal to, the substate. For a given substate $\mathbf{s}$, this can be translated within the CPN by updating this type to $\mathbf{s}$ in any arc leading to a place corresponding to a substate of $\mathbf{s}$.

We give our translation scheme in Algorithm 3. Given a state $\mathbf{s}$ and its history pseudostate $\mathbf{s^H}$, we update the value of the history variable h on each arc leading to $\mathbf{s}$ and its substates (lines 4–5). Then, for each substate $\mathbf{s'}$ of $\mathbf{s}$, we create a transition with the appropriate guard (lines 8–12) and to which $p(\mathbf{s^H})$ is connected (line 8). The rest of the algorithm (lines 13–17) connects this fresh transition to the destination place, either directly or through the sequence of entry behaviors using the same synchronization mechanism as in Algorithm 2.

We give in Figure 4 a part of the CPN resulting from the application of Algorithms 1, 2 and 3 to the SMD of Figure 1. We only consider the places and CPN transitions involved in handling history pseudostate $\mathbf{s_1^H}$, hence the branching to substates $init^*(\mathbf{s_{11}}) = \mathbf{s_{111}}$ and $\mathbf{s_{12}}$. We display with a thick border the places and transitions added by Algorithm 3. Transitions leading to $\mathbf{s_1^H}$ were already added by Algorithm 2, and are not shown here.

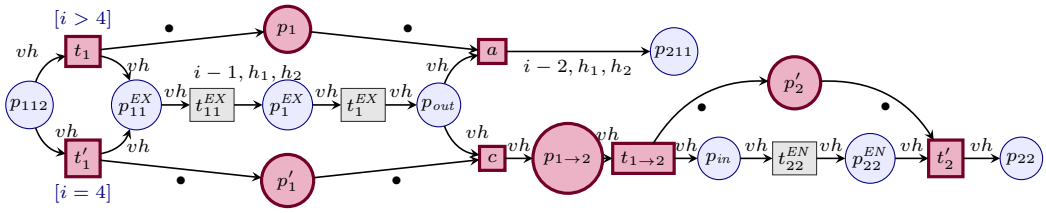

Figure 3: Modeling 2 transitions from the SMD of Figure 1

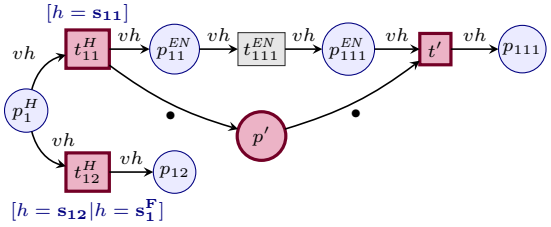

Figure 4: Modeling history pseudostate $\mathbf{s_1^H}$ of Figure 1 SMD

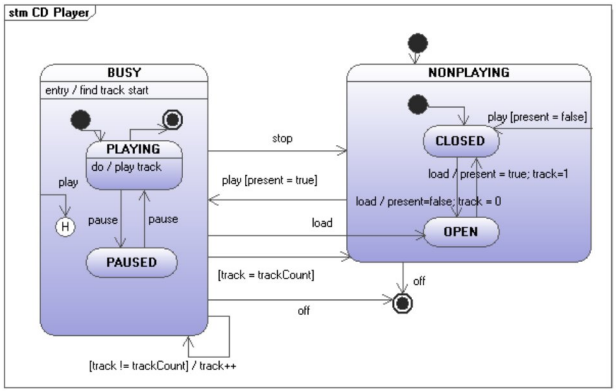

Figure 5: SMD for a CD Player [13]

3.6 Example: CD Player

We give in Figure 5 an example of SMD specifying a CD player [13]. This example features a hierarchy of state machines with internal and external transitions, several global variables, a simple entry behavior, and a simple history pseudostate. The behavior specified by this state machine is not concurrent, hence interesting to demonstrate our approach.

We give in Figure 6 a partial translation of this SMD (the full CPN is available in [1]). We denote **PLAYING** by PL, **PAUSED** by PA, `track` by t, `present` by p, `trackCount` by N, *play track* by PT, *find track start* by FTS. The token is of the form (p, t, h), where $p \in \{T, F\}$, $t \in \mathbb{N}$, and $h \in \{\text{PL}, \text{PA}\}$ is the history variable for **BUSY**. To keep the figure clear, we translated only all the transitions internal to **BUSY** and **NONPLAYING**, the transition from **NONPLAYING** to S^F (event `off`), the self loop on **BUSY** (except the case where the SMD is in the final state of **BUSY**), and the transition from **BUSY** to **OPEN** (event `load`). (All missing transitions are similar to the ones depicted.) Observe that p_{out} is isolated because there is no exit behavior. Also observe that transitions with a composite state as source (e.g., **BUSY**) correspond to an outgoing transition from each of its substates. Also observe that all arcs with target PL or PA update h accordingly. Finally note that the two synchronization mechanisms in the top left part (around p_{in}) and added by Algorithm 2 are useless: indeed, when one enters p_{in}, one will go through place FTS, transition FTS, another transition, and reach place PL. This comes from the fact that we have only one entry behavior, and this could be simplified (see Section 4).

By applying our algorithms, we generated a CPN corresponding to this case study, and were able to verify several properties using CPN Tools. For example, we formally checked that, whenever the SMD is in the PL state, the track number is valid (i.e., $1 \leq t \leq N$); also, being in the PL state always implies that there is a CD in the player (i.e., $p = T$).

4 Discussion and Perspectives

Conclusion. We introduced here a complete and thorough approach for translating non-concurrent SMDs to CPNs, allowing the formal specification of hierarchy of states with inter-level/internal/external transitions, entry/do/exit behaviors, complex variables and history states.

Future Work. The main future work is to introduce concurrent behaviors, viz., synchronization through events between different SMDs, and different regions in composite state machines, allowing to use fork and join. Our aim is to use the same idea as in [3], consisting in maintaining a pool of events, that are produced and consumed in different parts of the CPN. Global variables should also be maintained within a pool of variables, each of them in a different place.

Several optimizations could be made to the resulting CPN. So far, the main token has the same type in the whole net. This could be optimized, since some information is needed at some point of the net only. Variables appearing only in some composite state, and set in every incoming transition from the outside, could be discarded from the token when exiting the composite state, and added when entering it. Similarly, some history pseudostate information can be discarded; for example, in the SMD of Figure 1, variable h_2 is not necessary outside $\mathbf{s_1}$ (but note that variable h_1 is necessary even outside $\mathbf{s_1}$). Some synchronization mechanisms are also useless, when some entry behaviors always lead to the same place. For example, in Figure 6, one could get rid of all synchroniza-

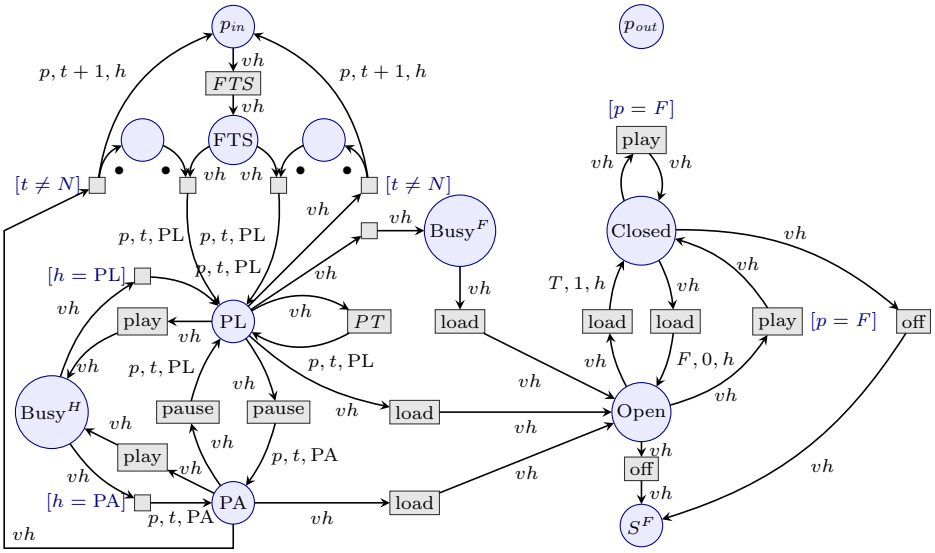

Figure 6: Translation of the CD Player SMD into a CPN (partial scheme)

tion mechanisms (around p_{in}), because it can be shown that every time one enters p_{in}, one will eventually reach place PL.

An implementation is in progress (based on the prototype of [3]). Such a tool will allow us to generate CPNs corresponding to large SMDs. Applying model checking techniques to the SMD can then be done as follows. We define *Labels* as the union of all events, behaviors and state names (including final states) appearing within *SMD*. For each behavior b, we set $L(t(b)) \leftarrow \{b\}$. For each transition $\tau \in \mathcal{T}$ with event $\mathtt{a}$, we set $L(t_\tau) \leftarrow \{a\}$. For each state $\mathbf{s}$, we set $L(p(\mathbf{s})) \leftarrow \bigcup_{\mathbf{s}' \in \mathbf{s} \cup ancestors(\mathbf{s})} \mathbf{s}'$, viz., the union of all ancestor states. Hence, in Figure 5, one can check linear time properties, such as $\Box((p = T) \wedge \mathtt{play} \rightarrow \Diamond\, PT)$ (where $\Box$ reads as "always", and $\Diamond$ "eventually").

Adding time is another important future work; this could be achieved using timed extensions of CPNs.

Acknowledgment. We are grateful to Charles Lakos for fruitful comments, to Zhang Shaojie and Liu Yang for helpful discussions on a preliminary version of this work, and to Anna Dedova for help with CPN Tools.

References

[1] É. André, C. Choppy, and K. Klai. Formalizing non-concurrent UML state machines using colored Petri nets (report). Research report, 2012. `www-lipn.univ-paris13.fr/specif/documents/UMLPN.pdf`.

[2] P. Bhaduri and S. Ramesh. Model checking of statechart models: Survey and research directions. *CoRR*, cs.SE/0407038, 2004.

[3] C. Choppy, K. Klai, and H. Zidani. Formal verification of UML state diagrams: a Petri net based approach. *SIGSOFT Softw. Eng. Notes*, 36:1–8, january 2011.

[4] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Computer Programming*, 8:231–274, 1987.

[5] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 2003.

[6] K. Jensen and L. M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009.

[7] Y. Jin, R. Esser, and J. W. Janneck. A method for describing the syntax and semantics of UML statecharts. *Software and System Modeling*, 3(2):150–163, 2004.

[8] J. Lian, Z. Hu, and S. M. Shatz. Simulation-based analysis of UML statechart diagrams: methods and case studies. *Software Quality Journal*, 16(1):45–78, 2008.

[9] K. L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992.

[10] OMG. Unified Modeling Language Superstructure Version 2.2. `http://www.omg.org/spec/UML/2.2/Superstructure/PDF/`, Feburary 2009.

[11] R. G. Pettit IV and H. Gomaa. Modeling behavioral patterns of concurrent objects using Petri nets. In *ISORC*, pages 303–312, 2006.

[12] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards flexible verification under fairness. In *CAV'09*, volume 5643 of *LNCS*. Springer, 2009.

[13] S. Zhang and Y. Liu. An automatic approach to model checking UML state machines. In *SSIRI-C'10*, pages 1–6. IEEE, 2010.