Formalising Concurrent UML State Machines Using Coloured Petri Nets[⋆]

Étienne André, Mohamed Mahdi Benmoussa, and Christine Choppy

Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, Villetaneuse, France

Abstract. While UML state machines are widely used to specify dynamic systems behaviours, their semantics is described informally, which prevents the complex systems verification. In this paper, we propose a formalisation of concurrent UML state machines using coloured Petri nets. We consider in particular concurrent aspects (orthogonal regions, forks, joins, shared variables), the hierarchy induced by composite states and their associated activities, internal/external/local transitions, and entry/exit/do behaviours.

1 Introduction

UML [OMG12] became the *de facto* standard for modelling systems, and features a very rich syntax with different diagrams to model the different aspects of a system. UML behavioural state machine diagrams (SMDs) are transition systems used to express the behaviour of dynamic systems in response to external interactions. Although UML is widely used in the industry, its semantics is not formally expressed, which prevents formal verification to be performed.

Related Work. Verification of SMDs has been often tackled. Some approaches directly give UML a semantics. The closest to the set of syntactic constructs we consider here is [JEJ04], where entry/exit behaviours, activities, synchronisation and history states are considered; however, global variables are discarded, and no model checking is performed. In [LLA$^+$13], an operational semantics is proposed for UML SMDs with synchronisation. Most syntactic aspects are taken into account, except real-time aspects and object-oriented issues, such as dynamic invoking and destroying objects. A formalisation has also been proposed using UML-B [SB06]. However, these approaches require the implementation of a standalone, dedicated tool.

Other approaches translate UML specification into an intermediate model of some model checker. Most of these approaches consider quite restrictive subsets of the UML syntax as defined by the OMG [OMG12] (see [LAC$^+$14] for a survey). Out of many approaches, for lack of space, we cite only the most recent and related to our work. A translation of SMDs to SPIN is considered

⋆ This is the author version of the paper of the same name accepted for publication at the 6th International Conference on Knowledge and Systems Engineering (KSE 2014). The final version is available at www.springer.com.

in [JDJ$^+$06] with both hierarchical and non-hierarchical cases; history, fork, join pseudo-states, entry and exit activities and complex data structures are not supported. A semantics is defined in [DJ07] for a syntax subset, including deferring of messages, concurrent composite states and choice pseudo-state; verification is performed using NuSMV. An automated translation from SMDs into CSP♯ (an extension of CSP) is proposed in [ZL10]; modelling techniques such as use of data structures, join/fork, history pseudo-states, entry/exit behaviours (but with no variable) are considered, and properties are checked using PAT [SLDP09]. Note that these formalisms do not provide a graphical representation, in contrast to CPNs. Other approaches (e.g. [KCBL10]) use an intermediary model (e.g., flat state machines) to formalise SMDs to coloured Petri nets (CPNs) [JK09]. In [KCBL10], concurrency (fork/join, concurrent composite states) is taken into account. The main difference with our approach is that [KCBL10] requires to flatten the SMD, hence losing the hierarchy. A translation from SMDs to Petri nets is proposed in [CKZ11] were SMDs include synchronisation, limited aspects of hierarchy, join and fork (with no inter-level transitions) but history pseudo-states and variables are not considered.

Contribution. We introduce here a translation of concurrent SMDs into CPNs. CPNs offer a detailed view of the process with a graphical representation, and benefit from powerful tools (such as CPN Tools [Wes13]) to test and check the model. Our SMDs can communicate on synchronised events; we also take into account the most common syntactic features, i.e. state hierarchy with entry/exit/do behaviours, history pseudo-states, synchronised events, fork/join, shared variables and local/external/internal transitions. Our approach partially relies on the work presented in [ACK12] where we proposed a first attempt of formalising non-concurrent SMDs using CPNs. Here, we extend this previous approach to the concurrency (forks, joins, nested composite states). We also add internal transitions. The addition of concurrency was not easy, as the translation scheme of [ACK12] heavily relied on the assumption of non-concurrency, and we had to entirely reconsider our approach, and rewrite our translation algorithms.

Outline. We present in Section 2 the formalisms we use (viz. SMDs and CPNs). In Section 3, we describe our translation: we first describe our general translation scheme, and recall the assumptions we make (Section 3.1); then, we define functions used in our algorithms (Section 3.2), and give in details our algorithms translating states and behaviours (Section 3.3), and transitions (Section 3.4). We conclude and give some perspectives in Section 4.

2 Basic Concepts Used in This Work

2.1 Our Assumptions on UML State Machine Diagrams (SMDs)

The underlying paradigm of UML SMDs [OMG12] is that of a finite automaton, that is each entity (or subentity) is in one state at any time and can move to another state through a well-defined conditional transition. For lack of space,

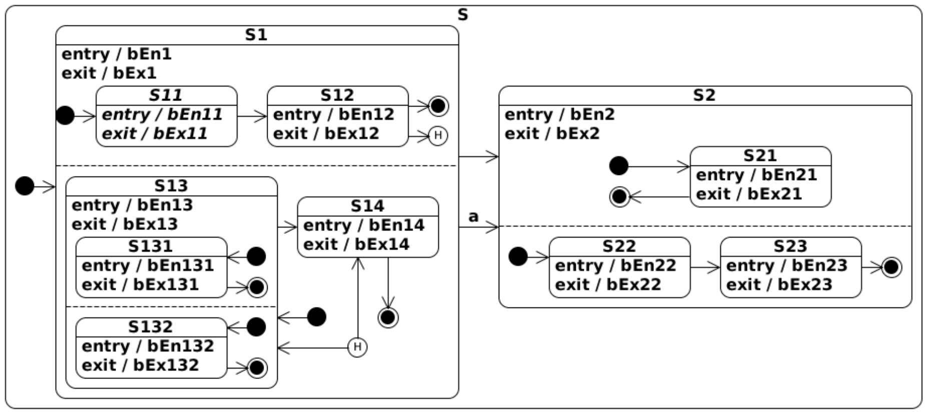

Fig. 1. Example of a state machine diagram

we assume the reader's knowledge on SMDs, and we only present here technical issues required for our work. The UML provides an extended range of constructs for SMDs, and we take into account the following syntactical elements: simple/composite states, entry/exit/do behaviours, concurrency (regions in composite states, fork/join transitions), shared variables, shallow history pseudostates (not detailed for lack of space) and hierarchy (of states and behaviours). In the following, we recall these elements together with some assumptions we need to set for our work. [1] We use the SMD in Fig. 1 as a running example.

States. We consider two kinds of states: simple and composite. A composite state is a state that contains at least one region and can be a simple composite state or an orthogonal state. A simple composite state has exactly one region, that can contain other states, allowing to construct hierarchical SMDs. An orthogonal state (e.g. S1, S2 in Fig. 1) has multiple regions (regions can contain other states), allowing to represent concurrency. $\mathit{regions}(\mathbf{s})$ denotes the set of regions in composite state $\mathbf{s}$, and $\mathit{NbRegions}(\mathbf{s})$ the number of regions in $\mathbf{s}$. We assume that a composite state must not be empty ("A composite State contains at least one region" [OMG12, p.322]), and that each region contains at least one state.

"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*." [OMG12, Section Kinds of States, p.319]. Given a composite state $\mathbf{s}$, we denote by $\mathit{SubStates}(\mathbf{s})$ the set of direct substates of $\mathbf{s}$ (including final states), and by $\mathit{SubStates}^*(\mathbf{s})$ all substates of $\mathbf{s}$, both direct and indirect. If $\mathbf{s}$ is not contained in any state, we call it a *root* state. Otherwise, $\mathit{parent}(\mathbf{s})$ denotes the state containing $\mathbf{s}$. The set of all states of an SMD is denoted by $\mathcal{S}$.

[1] A summary of our assumptions is available in Appendix B.

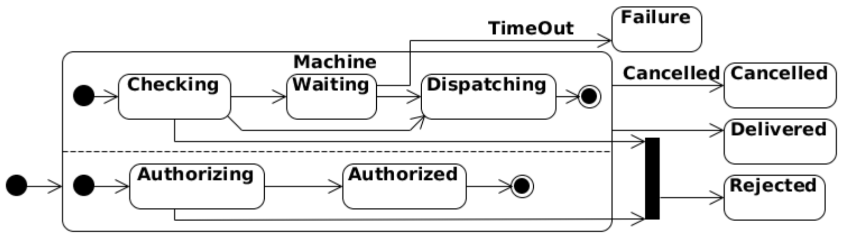

Fig. 2. Example of transitions taken into account

Behaviours. Behaviour expressions may be defined when entering, exiting states (or also when states have a do behaviour) or when firing transitions. As in [ACK12], we abstract behaviours using name b (corresponding to the actual behaviour expression) and function f to express changes induced on the system variables. The behaviour is denoted by (b, f). and when there are no modifications on variables we set f to id (id is the identity function). When a transition has no behaviour $(b, f) = (none, id)$. We assume that a do behaviour is an atomic behaviour that can be executed as many times as wished. This is a rather strong assumption in our setting. Furthermore, to simplify our algorithms (and avoid complicated subcases), we make two further assumptions, (i) only simple states can have a do behaviour, and (ii) each state (be simple or composite) always has an entry and an exit behaviour. These two assumptions could be lifted with no difficulty.

Initial Pseudostates. We require that each region contains one and only one (direct) initial pseudostate, which has one and only one outgoing transition. We also consider that the active state of the system cannot be an initial pseudostate. Note that this is a modelling 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.

Final States. In each region $\mathbf{r}$ of a composite state, we allow exactly one final state (denoted by $\mathbf{r}^F$), whereas the specification allows zero or one ("Each region [...] may have its own initial Pseudostate as well as its own FinalState." [OMG12, p.321]) Note that final states are simple states, and thus are included in relation *SubStates*. We define function *ToFinal*($\mathbf{s}$) that returns the set of simple states with transitions to final states of each region in composite state $\mathbf{s}$. Similarly, *ToFinal**($\mathbf{s}$) returns the set of composite states (including $\mathbf{s}$) that have transitions with the final states of each region in composite state $\mathbf{s}$ or its substates.

Variables. We allow any kind of variables in any behaviour and transition guard. Such variables (integers, lists, etc.) are often met in practice [OMG12].

Transitions. A transition can have a guard, a synchronization event, and a behaviour; 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 behaviour, etc.). Due to concurrency, various kinds of transitions exist: completion transitions (e.g. the transition from `Machine` to `Delivered` in Fig. 2) have no event and exit a composite state when all its regions are in their final state. Exiting a composite state machine through an event (e.g. the transition from `Machine` to `Cancelled` in Fig. 2) results "in the exiting of all substates of the composite State, executing any defined exit Behaviors starting with the innermost States in the active state configuration" [OMG12, Section Transitions, p.325]. This is an exception-like transition. A join transition (e.g. the transition from `Checking` and `Authorizing` to `Rejected` in Fig. 2) exits a given state in each composite state region. All these transitions are taken into account in our work. The "implicit join" (e.g. the transition from `Waiting` to `Failure` in Fig. 2) exits some of the regions from a given state; the other regions are exited whatever is their current active state. For readability sake, we do not take them into account in our algorithms, but our general scheme can perfectly adapt to this case.

Concurrency can also appear when entering a composite state (cf. left-most transition in Fig. 2). "If the Transition terminates on the edge of the composite State (i.e. without entering the State), then all the Regions are entered using the default entry rule." [OMG12, Section Entering a State, p.323]. We take into account forks (a transition with a given state in each region as destination), but not "implicit forks" (where only some of the regions have an explicit destination), although our algorithms could be trivially extended, at the cost of more complicated subcases. We also take into account internal/external/local transitions.

Finally, we make the following assumption: the execution of different transitions in different regions of the same state is done in parallel. For instance, the transition from `Checking` to `Waiting` in the upper region of `Machine` in Fig. 2 (and that could involve exit, do and entry behaviours) is performed in an interleaving manner with, e.g. the transition from `authorizing` to `authorized` in the lower region. Although this not entirely clear in [OMG12], this assumption might not conform to the notion of run-to-completion step. We believe that interleaving is indeed a natural mechanism between two subsystems executed in parallel.

The set of transitions is denoted by T with transitions of the form $t = (\mathcal{S}1, e, g, (b, f), sLevel, \mathcal{S}2)$, where $\mathcal{S}1, \mathcal{S}2 \subseteq \mathcal{S}$ are the source and target set of states respectively ($\mathcal{S}1$ or $\mathcal{S}2$ can contain at least one state – in the case of simple transitions between two states – or more – in case of fork or join transitions), e is the event, g is the guard, (b, f) is the behaviour to be executed while firing the transition, and $sLevel \in \mathcal{S}$ is the level state containing the transition.

We introduce a documented way to travel in the hierarchy of states. We indeed add the concept of *level state* (denoted by *sLevel*) of a transition from $\mathbf{s_1}$ to $\mathbf{s_2}$, that is the innermost state in the hierarchical SMD structure that contains the transition (the level state can be the SMD if $\mathbf{s_1}$ and $\mathbf{s_2}$ are root states). For example, let us consider that `S1` and `S2` in Fig. 1 are root states and the transition labelled by "a" between those states is encompassed by the state machine S. Then S is the level state for this transition because it is the innermost common ancestor of S1 and S2 that contains transition "a". Similarly, for the transition (say t) between S13 and S14, we have $sLevel(\mathtt{t}) = \mathtt{S1}$.

2.2 Coloured Petri Nets with Global Variables

CPNs [JK09] are also a kind of automaton represented by a bipartite graph with two kinds of nodes, places (e.g. p_1 in Fig. 3 (left)) and transitions (e.g. t in Fig. 3 (left)). Places hold tokens, possibly of a complex value, and that should be of the place type (e.g. type $\mathbb{N} \times \mathbb{B}$ in Fig. 3 (left)).

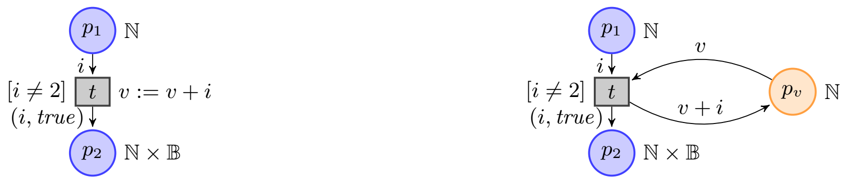

Fig. 3. Global variable notation (left) and corresponding semantics (right)

We use here the concept of *global variables*, a notation that does not add expressive power to CPNs, but renders them more compact. Global variables can be read in guards and updated in transitions, and are supported by some tools (such as CPN Tools). Otherwise, one can simulate a global variable using a "global" place, in which a single token (of the variable type) encodes the current value of the variable. An example is given in Fig. 3 (left). Variable v (of type $\mathbb{N}$) is a global variable updated to the expression $v + i$.

This construction is equivalent to the one in Fig. 3 (right). When a global variable is read in a guard, the token with value $v + i$ is put back in place p_v.

The CPN current state (or marking) is the information on which tokens are present in which places. The state evolves when a transition is fired, and tokens are consumed from its source places (according to the input arc expressions) and generated to its target places (according to the output arcs).

3 Translation of Concurrent State Machines

We first introduce our general translation scheme (Section 3.1). Then, we define functions used in our algorithms (Section 3.2). We then introduce the algorithms translating states and behaviours (Section 3.3), and transitions (Section 3.4).

3.1 General Translation Scheme

In this section, we present the general view of our translation. We define a translation scheme where simple states, final and history pseudostates are translated into places, whereas behaviour expressions (entry, exit and do) and events are translated into CPN transitions. Further places and transitions will also be defined to connect these places and transitions together. The firing of transitions in SMDs is represented by the firing of transitions with tokens in CPNs. Note that composite states will not be translated as such (no CPN place will correspond to a composite state), but their behaviours and all their simple substates will be considered. This is inline with the fact that the active state of an SMD is a (set of) simple state(s). Similarly, initial pseudostates will not be translated as such.

Encoding and Factoring Transitions. A main issue in our translation is to encode transitions between composite states with different regions, in particular the variants of forks and joins. Each such transition (in particular for joins) may in fact correspond to a large number of transitions. For example, in Fig. 1, firing the transition labelled with event "`a`" corresponds to 18 different ways to leave `S1` (3 possible active states in the upper region of `S1`, multiplied by 4 in `S13` and 2 others in the lower region), and hence in 18 CPN transitions, together with all the entry/exit behaviours. Given a transition t, let us denote by $\mathit{combinations}(t)$ the function that computes all possible combinations of outgoing substates. We could translate each transition separately; but this would quickly result in an explosion of the number of CPN transitions corresponding to the *same* entry/exit behaviours and actions.

For sake of readability, maintainability, and size of the translated CPN, this should be factored. Hence, as in [ACK12], we shall propose a scheme such that each behaviour is encoded into only one CPN transition. Since many SMD transitions go through the same behaviours, we need a "memory" mechanism to remember from which place we originate so as to find the correct target. Unfortunately, the factoring scheme of [ACK12] does not easily extend to the concurrent case, and further studies showed us that it could lead to a state space explosion when exploring branches that would lead to no end. We reuse the idea of having one unique transition for each behaviour in the SMD, and we define a structure encoding the hierarchy of all entry (resp. exit) behaviours together, in a sort of tree (resp. reversed tree). An example of two such trees will be given in Fig. 5, with the blue part enclosed in a dashed box on the left (resp. turquoise part enclosed in a dotted box on the right) corresponding to the entry (resp. exit) behaviours. Now, in contrast from [ACK12], we will navigate in these "trees" of behaviours using a memory mechanism, based on guards. For each possible way to leave a composite state when firing a transition, we will create a "path" in these trees: this can be easily obtained using typed tokens (e.g. using a unique integer identifier for each transition), and guards checking the type of the token along all the CPN transitions on the considered path. Checking a guard is very cheap when doing model checking, and this mechanism avoids state space explosion, while keeping the CPN small. For example, in Fig. 5, when exiting from `pF11` and `pF12`, the token will reach `tbEn2`, and will be duplicated in two tokens in `pbEn2`; then, although there are three possible outputs, i.e. `tbEn21`, `tbEn22` and `tbEn23`, we expect from our translation that one token fires `tbEn21` and the other one `tbEn22`, since these transitions correspond to the entry behaviours of the initial states of the two regions of `S2`. (For sake of conciseness, we do not depict explicitly in the figures this mechanism based on guards and tokens.)

Note that, in the case of the 18 combinations corresponding to the transition labelled with event "`a`", we still have 18 CPN transitions; but all exit and entry behaviours are factored, which considerably reduces the resulting CPN size.

Finally, we add a synchronisation CPN transition (e.g. `TEnS2_t1` in Fig. 5) just before entering the places corresponding to the destination simple states, which ensures that only one transition fires at a time in a given region (different transitions can still fire in an interleaving manner in different regions).

Encoding Shared Variables. In [ACK12], the non-concurrency allowed us to store the value of all shared variables in a single token that encoded also the current active simple state of the SMD. In the concurrent case, this approach is no longer possible since several simple states can be active at the same time; encoding the values of the same shared variables in different tokens would lead to consistency problems. Hence, we use global variables in CPNs (see Section 2.2) to encode the value of shared variables. Note that concurrent reading/writing of such variables can lead to "strange" executions (when a region reads a variable and then immediately changes its value, but in between another region concurrently changed its values), but these executions are also possible in the UML semantics.

3.2 Functions

We now describe the functions used in our algorithms. First, we consider two special places p_{in} (resp. p_{out}), corresponding to the root of the tree of entry (resp. exit) behaviours, that will be used by our algorithms. We assume function p associates place $p(\mathbf{s})$ with each state $\mathbf{s}$, and place $p(b)$ to each behaviour b. We also assume function t associates CPN transition $t(b)$ with each behaviour b. Functions $\mathit{SupEN}(\mathbf{s})$ and $\mathit{SupEX}(\mathbf{s})$ return the places representing entry behaviour ($p(b^{EN})$) and exit behaviour ($p(b^{EX})$) of a given state, respectively. Function $\mathit{SubEN}(\mathbf{s})$ returns the transition encoding the entry behaviour in case of a simple state, or the entry behaviour of each substate in case of a composite state.

In the hierarchy of entry or exit behaviours, we need to know from which behaviour we enter and from which behaviour we leave the state. This depends on the level of the transition. We thus define two functions $\mathit{OutTo}(\mathit{sLevel}, \mathbf{s_1})$ and $\mathit{InFrom}(\mathit{sLevel}, \mathbf{s_2})$ that return the entry and exit behaviour to enter/leave the hierarchy, respectively. Note that $\mathbf{s_1}$ of $\mathit{OutTo}(\mathit{sLevel}, \mathbf{s_1})$ (respectively $\mathbf{s_2}$ of $\mathit{InFrom}(\mathit{sLevel}, \mathbf{s_2})$) can be a set of states if the transition is a fork transition (respectively a join transition).

$$\mathit{OutTo}(\mathit{sLevel}, \mathbf{s_1}) = \begin{cases} p_{out} & \text{When } \mathit{sLevel} \text{ is the overall SMD} \\ \mathit{SupEX}(\mathit{sLevel})) & \text{otherwise} \end{cases}$$

$$\mathit{InFrom}(\mathit{sLevel}, \mathbf{s_2}) = \begin{cases} p_{in} & \text{When } \mathit{sLevel} \text{ is the overall SMD} \\ \mathit{SupEN}(\mathit{sLevel}) & \text{otherwise} \end{cases}$$

We finally assume a function $\mathit{init}^*(\mathbf{s})$ that returns the initial state of $\mathbf{s}$, or the (recursive) set of initial states of the initial state if $\mathbf{s}$ is composite.

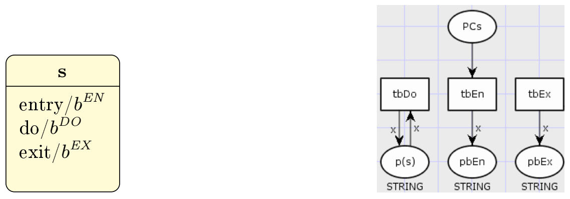

Fig. 4. A simple state (left) and its translation (right)

3.3 Translating States and Behaviours

We describe here the translation of the states and their behaviours (do, entry and exit). One the one hand, we translate each simple, final and history (pseudo)state into a place. On the other hand, we translate the purely hierarchical structure of the SMD, so that to get a tree of entry and exit behaviours, that will be used later when connecting transitions. Each behaviour expression is represented by a transition and a place connected by an arc. We also connect the places corresponding to simple states with their "do" behaviour, if any. Connecting the entry/exit behaviours with the places corresponding to simple states depends on the transitions and will be done by Algorithm 2.

First, as an example, Fig. 4 shows a state `s` and its translation into a CPN. State `s` is represented by place $p(\mathtt{s})$, its entry behaviour b^{EN} is represented by place `pbEn` and transition `tbEn`, and its exit behaviour b^{EX} is represented by place `pbEx` and transition `tbEx`. If the state has a do behaviour b^{DO} then it will be represented by transition `tbDo`.

We use in our algorithms the following graphical notation. Places and transitions generated at a given point are denoted by a *solid* line, whereas places and transitions already generated are denoted by a *dotted* line.

Algorithm 1 translates states and behaviours. Two places p_{in} and p_{out} are created (line 1), then there are two main steps: Step 1 generates the CPN part corresponding to do/entry/exit behaviours as well as history pseudostates and final states. Step 2 is composed of 3 substeps. Step 2.1 adds an arc between the entry behaviour place of a composite state and the entry behaviour transitions of its substates. This represents the fact that, after executing an entry behaviour, we will execute the entry behaviours of the substates. Step 2.2 adds an arc from place p_{in} to the entry behaviour transitions of the root states, and from the exit behaviours of the root states to place p_{out}. Step 2.3 adds an arc from the exit behaviour transition to the exit behaviour place of its parent.

After executing Algorithm 1, we apply the following initialisations to the result of the translation: All guards are initialised to false (and may be later modified during the translation).

Fig. 5 shows the application of Algorithm 1 to a part of the example in Fig. 1. The blue part enclosed in a dashed box on the left of Fig. 5 represents the entry behaviours of `S2` in Fig. 1. The dotted/turquoise part on the right of Fig. 5

Algorithm 1: Encoding the states and behaviours

```
1  Add $p_{in}$  $p_{out}$
   // Step 1
2  foreach state $\mathbf{s} \in \mathcal{S}$ do
3      Add $\boxed{t(b^{EN})} \xrightarrow{NbRegions(\mathbf{s})} p(b^{EN})$ and $p(b^{EX}) \xrightarrow{NbRegions(\mathbf{s})} \boxed{t(b^{EX})}$
4      if s is a simple state and has a "do" behaviour $(p(b^{DO}), f^{DO})$ then
5          Add $p(\mathbf{s}) \rightleftarrows \boxed{t(b^{DO})}$ $f^{DO}$
6      foreach region $\mathbf{r} \in regions(\mathbf{s})$ do
7          if r has a (direct) history pseudostate then Add $p(\mathbf{r}^H)$
8          if r has a (direct) final state then Add $p(\mathbf{r}^F)$
   // Step 2
9  foreach state $\mathbf{s} \in \mathcal{S}$ do
10     if s is a composite state then
           // Step 2.1
11         foreach $\mathbf{s}' \in SubStates(\mathbf{s})$ do
12             foreach transition $t' \in SubEN(\mathbf{s}')$ do connect $p(b^{EN}) \rightarrow \boxed{t'}$
13     if s is root then connect $p_{in} \rightarrow \boxed{t(b^{EN})}$  $\boxed{t(b^{EX})} \rightarrow p_{out}$      // Step 2.2
14     else Add $\boxed{t(b^{EX})} \rightarrow SupEX(parent(\mathbf{s}))$ ;                  // Step 2.3
```

represents the exit behaviours of S1 in Fig. 1. We also added places to represent final states (e.g. pF11), history pseudostates (pH11) and states (e.g. pS11, pS22).

3.4 Translating Transitions

Algorithm 2 describes the translation of UML transitions, using three steps followed by an initialisation function. Step 1 deals with exit behaviours: Step 1.1 deals with event triggered transitions, while other transitions are processed in Step 1.2 (exit from a composite state) and 1.3 (exit from a simple state). For a given transition, Step 2 establishes the connection between its source state exit behaviour and its destination state entry behaviour. Step 3 expresses the destination state entry.

Line 19 updates all relevant guards, so as to guide the token within the hierarchy of behaviour places and transitions. Finally, we use a function *Initialisation*() for token initialisation (line 20) to model the initial state in the global SMD.

Fig. 5 shows the application of Algorithms 1 and 2 to S1 and S2 and also the transition without an event in Fig. 1. In this translation, we consider only the entry behaviours of S2, the exit behaviour of S1, the final states of S1, to get a clear picture to illustrate the algorithm. Step 1 adds transition TExSF_t1 between places pF11 and pF12 (that correspond to the region final states) and place pbEx1 that correspond to the exit behaviour of S1. Step 2 adds transition

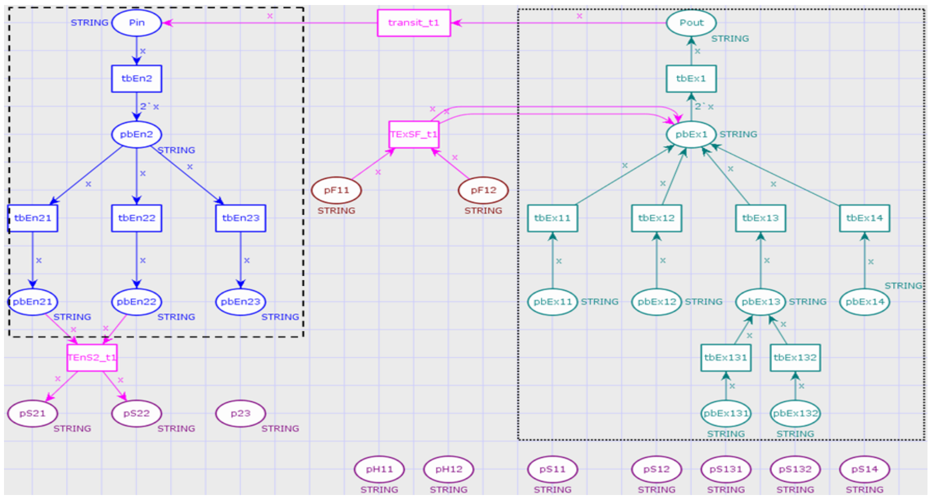

Fig. 5. Application of Algorithms 1 and 2

`transit_t1` to connect exit place `Pout` to entry place `Pin`. Step 3 adds the transition `TEnS2_t1` to link places of `S2` substates and their entry behaviour.

4 Conclusion and Future Works

We present here a formalization of UML concurrent SMDs by translating them into CPNs. We take into account different syntactic elements in our translation, specially the concurrency, including simple and composite states, most kinds of transitions (including forks and joins), behaviours (entry, exit, do), history pseudo-states, etc. Once our implementation is completed (see below), this will allow for automated model checking of SMDs.

Our main future work is to develop a tool automating the translation so as to be able to perform formal verification of SMDs. We implemented a first prototype using Acceleo, but this technology turned out to be slightly inaccurate for our framework [ABC14]. Although we could translate toy examples, we will likely build a home-made tool. Of course, this implementation must extend to the general case the assumptions made in order to simplify the description of our algorithms (e.g. each state always has both an entry and an exit behaviours).

Most syntactic aspects not taken into account in our work (see Appendix A) could be added in a rather straightforward manner – except for timing aspects. Adding them to our translation is an interesting future work. Recall that some tools (including CPN Tools) allow one to defined time(d) coloured Petri nets.

Finally, although it goes beyond of the scope of this paper, a challenging future work would be to formally prove the equivalence between the original SMD and the resulting CPN. Of course, the problem is that the OMG does not formally define a formal semantics for SMDs. However, we could reuse the

Algorithm 2: Encoding the transitions

```
1  foreach transition $t = (\mathcal{S}1, e, g, (b, f), sLevel, \mathcal{S}2) \in T$ do
       // Step 1
2      if t has an event e then
           // Step 1.1
3          foreach $c \in combinations(t)$ do
4              Add $\overset{V := f(V)}{\boxed{e}}$
5              foreach simple state $\mathbf{s} \in c$ do
6                  Add $p(\mathbf{s}) \rightarrow \boxed{e} \rightarrow SupEx(\mathbf{s})$
7      else
           // Step 1.2
8          if $\mathcal{S}1$ is a composite state then
9              Add $\boxed{TEx\mathcal{S}1\_t}$
10             foreach state $\mathbf{s} \in ToFinal(\mathcal{S}1)$ do
11                 Add $p(\mathbf{s}) \rightarrow \boxed{TEx\mathcal{S}1F\_t} \rightarrow SupEx(\mathcal{S}1)$
12         else
               // Step 1.3
13             Add $p(\mathcal{S}1) \rightarrow \boxed{TEx\mathcal{S}1\_t} \rightarrow SupEx(\mathcal{S}1)$
       // Step 2
14     Add $OutTo(sLevel, \mathcal{S}1) \rightarrow \boxed{transit\_t} \rightarrow InFrom(sLevel, \mathcal{S}2)$
       // Step 3
15     Add $\boxed{TEn\mathcal{S}2\_t}$
16     foreach state $\mathbf{s_2} \in \mathcal{S}2$ do
17         foreach simple $\mathbf{s} \in init^*(\mathbf{s_2})$ do
18             Add $SupEn(\mathbf{s}) \rightarrow \boxed{TEn\mathcal{S}2\_t} \rightarrow p(\mathbf{s})$
19     Update all relevant guards
20 Initialisation()
```

operational semantics that we recently proposed for SMDs [LLA+13], and define a trace equivalence taking into account active states, behaviours and events.

References

ABC14. Étienne André, Mohamed Mahdi Benmoussa, and Christine Choppy. Translating UML state machines to coloured Petri nets using Acceleo: A report. In *ESSS*. EPTCS, 2014. 11

ACK12. Étienne André, Christine Choppy, and Kais Klai. Formalizing non-concurrent UML state machines using colored Petri nets. *ACM SIGSOFT Soft. Eng. Notes*, 37(4):1–8, 2012. 2, 4, 7, 8, 14

CKZ11. Christine Choppy, Kais Klai, and Hacene Zidani. Formal verification of UML state diagrams: a Petri net based approach. *ACM SIGSOFT Soft. Eng. Notes*, 36(1):1–8, 2011. 2

DJ07. Jori Dubrovin and Tommi A. Junttila. Symbolic model checking of hierarchical UML state machines. Technical Report B23, Helsinki University of Technology, 2007. 2

JDJ$^+$06. Toni Jussila, Jori Dubrovin, Tommi Junttila, Timo Latvala, and Ivan Porres. Model checking dynamic and hierarchical UML state machines. In *MDV*, 2006. 2

JEJ04. Yan Jin, Robert Esser, and Jörn W. Janneck. A method for describing the syntax and semantics of UML statecharts. *Software and System Modeling*, 3(2):150–163, 2004. 1

JK09. Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009. 2, 6

KCBL10. Elhillali Kerkouche, Allaoua Chaoui, El Bay Bourennane, and Ouassila Labbani. A UML and colored Petri nets integrated modeling and analysis approach using graph transformation. *Journal of Object Technology*, 9:25–43, 2010. 2

LAC$^+$14. Shuang Liu, Étienne André, Christine Choppy, Jin Song Dong, Yang Liu, Jun Sun, and Bimlesh Wadhwa. Formalizing UML state machines semantics for formal analysis – A preliminary survey, 2014. Research report, available at `lipn.fr/~andre/UML-SMD-survey.pdf`. 1

LLA$^+$13. Shuang Liu, Yang Liu, Étienne André, Christine Choppy, Jun Sun, Bimlesh Wadhwa, and Jin Song Dong. A formal semantics for the complete syntax of UML state machines with communications. In *iFM*, volume 7940 of *LNCS*, pages 331–346. Springer, 2013. 1, 12

OMG12. OMG. Unified Modeling Language Superstructure, Version 2.5, beta 1. `http://www.omg.org/spec/UML/2.5/Beta1/PDF/`, oct 2012. 1, 2, 3, 4, 5, 14, 15

SB06. Colin F. Snook and Michael J. Butler. UML-B: Formal modeling and design aided by UML. *ACM TSEM*, 15(1):92–122, 2006. 1

SLDP09. Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. PAT: Towards flexible verification under fairness. In *CAV*, volume 5643 of *LNCS*. Springer, 2009. 2

Wes13. Michael Westergaard. CPN Tools 4: Multi-formalism and extensibility. In *Petri Nets*, volume 7927 of *Lecture Notes in Computer Science*, pages 400–409. Springer, 2013. 2

ZL10. Shaojie Zhang and Yang Liu. An automatic approach to model checking UML state machines. In *SSIRI-C*, pages 1–6. IEEE, 2010. 2

A Summary of the Syntax Taken into Account

We recall in Table 1 the syntactic elements we consider in our translation. Deep history pseudostates, submachine states ("semantically equivalent to a composite state") and implicit forks/joins were discarded for sake of simplicity but can be added in a very straightforward manner, and will be part of our implementation. Also note that shallow history pseudo states can be considered in our work by directly importing the mechanism from [ACK12], as this mechanism is not impacted by the addition of concurrency. We did not consider entry/exit points and choice/merge pseudostates, but we believe that there would be no difficulty for adding them to our scheme. Deferred events and timing aspects were not considered at all, and may require more work, or even lead us to reconsider parts of our translation scheme.

Element	**Considered?**
Simple / composite states	Yes
Orthogonal regions	Yes
Initial / final (pseudo)states	Yes
Terminate pseudostate	No (but trivially extensible)
Shallow history states	Identical to [ACK12]
Deep history states	No (but trivially extensible)
Submachine states	No (but trivially extensible)
Entry / exit points	No (but probably easy)
Entry / exit / do behaviours	Yes
Shared variables	Yes
External / local / internal transitions	Yes
Basic fork / joins	Yes
Implicit fork/joins	No (but trivially extensible)
Choices / merges	No (but probably easy)
Deferred events	No
Timing aspects	No

Table 1. Summary of the syntactic aspects considered

B Assumptions

We recall here the assumptions we made regarding [OMG12]. We give for each of them the reason for the assumption (e.g. unclarity in [OMG12]) and mention whether this is or not a strong assumption in our translation.

States. We require that each region of a composite state contains one and only one initial pseudostate, which has one and only one outgoing transition. In the specification, it not compulsory: each region "may have its own initial Pseudostate as well as its own FinalState" [OMG12, Section Regions, p.318]. Furthermore, a state that becomes active with no initial pseudostate is a semantic variation point. It seems natural to us to require exactly one initial pseudostate with one and only one outgoing transition. This is also a strong assumption in our setting.

Similarly, we also assume that each region in a composite state must contain exactly one final state. Again, this seems to be a very natural assumption; note that it is a less strong assumption in our setting, since our translation would still work without final states.

We also consider that the active state of the system cannot be an initial pseudostate. (The specification is rather unclear about it.) Note that this is a modelling 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.

We also assume that each region in a composite state contains at least one region ("A composite State contains at least one region" [OMG12, p.322]), and that a region contains at least one state. The specification does not explicitly mention this latter point, but the cardinalities in [OMG12, p. 317] seem to allow 0 state in a region. It seems natural to us that a region must be non-empty. This is also a rather strong assumption in our setting.

Behaviours. We assume that do behaviour are atomic behaviours that can be executed as many times wished; this may go against [OMG12] that seems to rely on the notion of a "continuous" behaviour. This is a strong assumption in our setting, partially coming from the fact that CPNs are a discrete model. This assumption could be lifted with the introduction of timing aspects.

Furthermore, we assume that only simple states can have a do behaviour; this assumption could be lifted without any difficulty but simplifies our algorithms.

Transitions. It is clear that, when in the final state in a region $\mathbf{r}$ of a 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 [OMG12], 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 $\mathbf{r}^F$. This assumption was made only because the specification is unclear about this, and either interpretation could be considered in our translation.

Finally, we make the following assumption: the execution of different transitions in different regions of the same state is done in parallel. For instance, the transition from `Checking` to `Waiting` in the upper region of `Machine` in Fig. 2 (and that could involve exit, do and entry behaviours) is performed in an interleaving manner with, e.g. the transition from `authorizing` to `authorized` in the lower region. Although this not entirely clear in [OMG12], this assumption might not conform to the notion of run-to-completion step. We believe that interleaving is indeed a natural mechanism between two subsystems executed in parallel.