

# Formalising Concurrent UML State Machines Using Coloured Petri Nets<sup>1</sup>

Étienne André, Mohamed Mahdi Benmoussa and Christine Choppy

**Abstract.** With the increasing complexity of dynamic concurrent systems, a phase of formal specification and formal verification is needed. UML state machines are widely used to specify dynamic systems behaviours. However, the official semantics of UML is described in a semi-formal manner, which renders the formal verification of complex systems delicate. In this paper, we propose a formalisation of UML state machines using coloured Petri nets. We consider in particular concurrent aspects (orthogonal regions, forks, joins, variables), the hierarchy induced by composite states and their associated activities, external, local or inter-level transitions, entry/exit/do behaviours, transition priorities, and shallow history pseudostates. We use a CD player as a motivating example, and run various verifications using CPN Tools.

**Keywords:** Modelling; State machines; Formalisation; Formal semantics; Coloured Petri nets

## 1. Introduction

The Unified Modelling Language (UML) proposed by the Object Management Group (OMG) [OMG15] became the *de facto* standard for modelling systems, both in the industrial and academic contexts. UML features a very rich syntax with different diagrams to model the different aspects of a system. We consider here the latest version of UML (*i.e.* 2.5), and we are interested here in UML behavioural state machine diagrams (hereafter SMDs), that are transition systems used to express the behaviour of dynamic systems in response to external interactions. A major problem with UML is that the official specification given by the OMG in [OMG15] gives a formal syntax, but a semantics that is only described in natural language. Hence, this prevents formal verification techniques to be applied.

We assume that we can use communicating FSMs (finite state machines) or UML state machines to express systems. However FSMs are generally used to express small problems and they tend to become unmanageable for complex systems (as mentioned, *e.g.* in [Sam09, Part 2: “UML extensions to the traditional FSM formalism”]). We think that using UML state machines is easier to read and to use graphically.

As presented below, there are several ways to equip the UML, and SMDs in particular, with a formally defined semantics, and we propose here a new translation of concurrent SMDs into coloured Petri nets (hereafter CPNs) [JK09]. We chose CPNs because they offer a detailed view of the process with a graphical

---

<sup>1</sup> This manuscript is the author version of the manuscript of the same name published by Formal Aspects of Computing in 2016. The official published manuscript can be found at <https://link.springer.com/article/10.1007/s00165-016-0388-9>.

representation and a proper handling of concurrency and data, and they benefit from powerful tools (such as CPN Tools [Wes13]) to test and check the models.

## 1.1. Related Works

There are several approaches to formally give a semantics to SMDs. However, most of these approaches consider quite restrictive subsets of the UML syntax defined by the OMG [OMG15]. These approaches can usually be classified in two categories: approaches that directly define an operational semantics, or approaches that translate SMDs into a language equipped with a formal semantics.

### 1.1.1. Operational Semantics

Several works attempted to directly provide operational semantics for UML state machines. In [Bee02], an operational semantics is provided for a subset of UML state machines. The approach uses terms to represent states, and transitions are nested into Or-terms, which makes it hard to extend to support the other features, especially compound transitions in the presence of fork and join pseudostates.

In [MPT03], a comparison between various interpretations of the statecharts semantics in the framework of structural operational semantics is presented. However, beyond the fact that statecharts are quite different from the latest UML state machines specification, only some syntactic aspects are covered.

In [Sch05], non-determinism is considered in orthogonal composite states, but only a subset of features is supported, and neither the event pool mechanism nor the concept of run-to-completion step (an important concept in SMDs) are discussed.

In [FS07], another formal semantics is defined for a subset of UML state machines features. The remaining features are informally transformed to the formalised features. However, the informal transformation procedure as well as the extra costs it introduces might make this approach not suitable for tool development and formal verification.

We believe the most complete work providing an operational semantics is [LLA+13], where we considered the entire syntax of UML state machines, with the exception of timed aspects.

**Limits of direct operational semantics approaches** The drawbacks of providing an operational semantics are twofold. First, these approaches require the development of an *ad-hoc* tool, which may be time consuming. In contrast, translation approaches benefit from state-of-the-art verification tools (the only tool to be implemented is the tool to perform the translation, which we believe is much less cumbersome than implementing a model-checking tool). These state-of-the-art model checkers often benefit from several years of development and maturation, and embed many optimisations to perform efficient verification.

Second, these operational semantics approaches are less easy to maintain: in case of new syntactic constructs introduced by the OMG, or in case of changes of the syntax and/or the (semi-formal) semantics, then not only the operational semantics shall be changed, but also the associated verification tool.

### 1.1.2. Translation to Existing Formalisms

Considering translations, we consider in particular (variants of) CSP (Communicating sequential processes) and Petri nets, as they are the most common target formalisms in the literature; furthermore, translations to CSP and Petri nets are also quite complete works in terms of the subset of the UML syntax considered in the translation. These two formalisms are also natural: CSP is a hierarchical language that captures the hierarchy of SMDs, whereas Petri nets have a token system that captures the active state of SMDs. We also report approaches translating SMDs into other formalisms such as extensions of automata, abstract state machines or Promela (the input language of the SPIN model-checker).

**Translation to CSP** Ng and Butler [NB02, NB03] propose to translate UML state machines into CSP. However, many features of UML state machines, such as the priority mechanism, are not modelled.

Zhang and Liu [ZL10] provide an approach which translates UML state machines into CSP#, an extension of the CSP language, which serves as the input modelling language of PAT [SLDP09]. Advanced modelling techniques such as use of data structures, join/fork, history pseudostates, entry/exit behaviours (but with no

use of variables inside) are considered, and liveness and safety properties are checked using PAT. However, the transition selection (including priorities) does not seem to be considered.

Jacobs and Simpson [JS15] present a translation of SysML activities and state machines into CSP, which includes a limited set of syntactic constructs of UML state machines.

Note that CSP does not provide a graphical representation, in contrast to Petri nets.

**Translation to Petri nets** A natural target formalism for translating UML state machines is Petri nets (and their extensions), as it is graphical, and the OMG mentions them in the UML specification 1) when mentioning that “Activities [are] defined using Petri-net-like graphs” [OMG15, p.283], and 2) when mentioning that join pseudostates in state machines are “similar to junction points in Petri nets” [OMG15, p.311].

Several approaches use CPNs for modelling and analysing concurrent systems. The work proposed in [PG00] presents an approach using CPNs to model and validate the behavioural characteristics of concurrent object architectures modelled by UML. The authors discuss how to map active/passive objects as well as message communications into CPNs. Though not specifically dealing with UML state machines, that paper provides a general idea of transforming UML diagrams to CPNs. In [PG01], state-dependent objects with their statecharts are mapped to CPNs. Each state-dependent object contains an encapsulated statechart that will be used when receiving an event. In the approach, it is not clear which set of syntactic elements for statecharts is taken into account during the mapping to CPNs. In the same manner, the work in [PG06] consists in modelling and analysing concurrent object-oriented software using behavioural patterns and CPNs. This approach uses collaboration diagrams to model a set of objects (each object has an associated UML state machine diagram). Each object has a stereotype to indicate which patterns to use, knowing that there is a set of defined patterns. In our work, we are interested in defining a semantics (using a translation algorithm) of UML state machines.

Trowitzsch and Zimmermann [TZ05] propose to translate a subset of timed UML state machines into stochastic Petri nets. The approach does not cover many UML state machines features, but timed events are discussed.

Lian *et al.* [LHS08] present a tool that allows to perform different operations to analyse UML statechart diagrams at different levels of model complexity. The analysis operations are based on the analysis of Petri net models converted from UML statecharts.

Hillah and Thierry-Mieg [TH08] present an approach that translates UML models (activity diagrams, etc.) to an instantiable Petri net. The translation is implemented using the BCC (Behavioural Consistency Checker) prototype.

More recently, a translation from SMDs to Petri nets was proposed in [CKZ11] where SMDs include synchronisation, limited aspects of hierarchy, join and fork (with no inter-level transitions); history pseudostates and variables are not considered.

In [ACK12], we propose an approach different from the work by [CKZ11], where we support a larger subset of UML state machine features, including state hierarchy, local/external transitions, entry/exit/do activities, etc. However, a limitation of that approach is that concurrency is left out: hence fork and join pseudostates, as well as orthogonal composite states are not considered.

[ABC14a] extends [ACK12] by reintroducing the concurrency; hence [ABC14a] supports the syntactic elements considered in [ACK12], with the addition of fork and join pseudostates, as well as some cases of inter-level transitions. This paper builds on top of the work presented in [ABC14a].

Finally, Luciano *et al.* [BP01] propose another approach to formalize UML with high-level Petri nets, *i.e.* Petri nets whose places can be refined to represent composite places. Note that high-level Petri nets are not coloured Petri nets, but both high-level Petri nets and CPNs give a higher level of abstraction in Petri nets, and lead to more compact models. In that work, class diagrams, state diagrams and interaction diagrams are considered. Customisation rules are provided for each diagram. But the authors do not provide details about those customisation rules; instead, they illustrate the steps with the hurried philosopher problem. The analysis and validation are also discussed, especially how to represent in UML the properties (such as absence of deadlocks, fairness etc.), as well as how to translate them into Petri nets representations.

**Automata** Other approaches translate UML specification into an intermediate model of some model checker, *e.g.* SPIN [Hol03]. The first work with SPIN as a target [LMM99] presents a translation scheme for SMDs into a Promela model (the input language of SPIN), and then invokes SPIN for verification; advanced modelling technique such as fork, join, history pseudostates, entry and exit behaviour of states, variables and multiple state machines are not considered. In [GLM02] an extension of [LMM99] is proposed to include

multiple UML state machines communicating asynchronously. This approach considers a subset of UML state machines and does not consider the pseudostates (except the initial pseudostate) and actions.

A translation of SMDs to SPIN is also considered in [JDJ+06] with both hierarchical and non-hierarchical cases; history, fork, join pseudostates, entry and exit activities are not supported. In [CJ09], a prototype tool is designed to link SPIN with RSARTE (IBM Rational Software Architect RealTime Edition), a modelling tool for UML diagrams. This work focuses on all kinds of RT-UML diagrams, *i.e.* UML diagrams related with real-time features. As part of UML, state machines are also translated into Promela in their approach. Since their work is not aiming at specifically model checking UML state machines, it does not provide detailed discussions about each feature of UML state machines, but discusses the communications between different objects. In contrast, we do not focus here on real-time, but aim at considering most of the syntax of UML state machines in our translation.

Knapp *et al.* [KMR02] present a work to verify real-time systems using UML state machines, UML collaboration diagrams and UPPAAL timed automata. They propose a prototype tool called HUGO/RT that verifies automatically whether scenarios specified by UML collaboration with time constraints are indeed realised by a set of timed UML state machines. The tool implements the translation of timed UML state machines into timed automata, and the time-annotated collaborations into an observer UPPAAL timed automaton. The UPPAAL model checker [LPY97] is used to verify the timed automata representing the model against the observer timed automaton.

**Other translation approaches** An approach proposed in [Per95] provides a representation of statecharts using transitions structures. This approach considers a simplified version of statecharts and does not consider variables.

In [GP98] the authors propose a transformation of statechart diagrams into graphs to show the intended semantics. However there is a lack of information about which elements are taken into account such as entry/exit actions, internal transitions, or (fork/join, junction, choice) pseudostates.

A work proposed in [LBC99] defines a process-algebra semantics for statecharts. This semantics is accompanied with a language called SPL (statecharts process language). However, this approach takes into account only subsets of Statecharts in which boundary-crossing transitions are disallowed.

In [BRS00], abstract state machines (ASMs) are used to formalise informal state requirements. ASMs allow to formulate conditions for system validation and verification. The syntax model covers many UML state machines features such as deferred events, completion events and internal activities. However, pseudostates such as fork, join, junction, choice, terminate are not considered.

## 1.2. Contribution

We introduce here a new translation of concurrent SMDs into CPNs. We take into account most syntactic features, *i.e.* state hierarchy with entry/exit/do behaviours, shallow history pseudostates, (synchronised) events, fork/join, variables and local/external or inter-level transitions.

This translation represents our understanding of the state machine specification (proposed by the OMG). We use a tool (Papyrus) for the graphical representations of state machine diagrams that implements exactly the constraints of the specification proposed by the OMG. For the ambiguity parts of some elements in the specification of UML, we propose an interpretation.

We entirely revise the translation mechanism of [ABC14a], leading to what we think is a significantly clearer solution. This also leads to a simpler translation algorithm. In particular, in [ABC14a] we used a fixed structure in CPNs to represent entry and exit behaviours of SMDs and this yielded an unwanted complexity. This is now replaced by instructions associated with transitions. Another new contribution is that in order to clarify our approach, we present in a systematic way some cases (in Section 4.5) to be taken into account in order to translate a transition (triggered or not by an event) between two states, that may be simple, composite, orthogonal etc., with/without entry/do/exit behaviours. We release the constraint we had in [ABC14a] that was requesting all states to have entry and exit behaviours. In addition to the syntax taken into account in [ABC14a], we also added history pseudostates. Our solution is also consistent with the notion of run-to-completion step of UML SMDs, and we present an extension to take into account the transition selection algorithm of the OMG specification. We also present in a detailed way our case study, and show how the associated CPN can be used to achieve some property verification (properties were not addressed in [ABC14a]).

We also give hints on how some of our assumptions can be lifted, and how to extend our translation to the syntactic elements not considered in this work. Altogether, this makes our translation cover the vast majority of syntactic constructs defined in [OMG15], with only two significant exceptions: deferred events and the integration of real-time constraints.

**Outline** We first present in Section 2 the formalisms we use (*viz.* SMDs and CPNs). We present in Section 3 a motivating example of a CD player, that is used throughout the article. In Section 4, we describe our formalisation of UML state machines by translating them into coloured Petri nets. In Section 5, we extend our approach so as to take into account the transition selection algorithm of the UML. In Section 6, we apply our translation scheme to the CD player. We conclude and give some perspectives in Section 7.

## 2. Basic Concepts

### 2.1. UML State Machine Diagrams

In this section, we first recall state machines as proposed by the OMG (Section 2.1.1), and we then formalise the syntax while introducing assumptions (Section 2.1.2). We put more focus on some points that require some precise care, and do not present those that we do not consider (*e.g.* choice pseudostate, junction, entry point, exit point, etc.).

#### 2.1.1. Recalling UML State Machines

The underlying paradigm of UML SMDs [OMG15] is that of a finite automaton, *i.e.* each entity (or subentity) is in one state at any time and can move to another state through a well-defined conditional transition. The UML provides an extended range of constructs for SMDs. In the following, we recall those elements. In addition to our motivating example (depicted in Fig. 3 in Section 3), we also use an SMD (depicted in Fig. 1) with the elements that we take into account. This example does not correspond to a real life case study but contains most important subtle situations of SMDs (especially regarding transitions).

**States** UML defines three kinds of states, that may (or not) contain regions: simple states (*e.g.* S11 in Fig. 1), composite states (*e.g.* S1 and S13 in Fig. 1) and submachine states.

A *simple* state has no region, and hence neither any internal state nor any transition. 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 and S13 in Fig. 1) has multiple regions (regions can contain other states), allowing to represent concurrency. Regions are separated using a dashed line. Each composite state must not be empty, and each region contains at least one state. Given a region of a composite state, we refer to its *direct substates* as the set of states immediately contained in this region (*e.g.* in Fig. 1 the direct substates of the lower region of S1 are S13, S14 and the final state of the region), and to its *indirect substates* as the transitive closure of the substate relation (*e.g.* in Fig. 1 the indirect substates of the lower region of S1 are S13, S131, S132, S14, and three final states).

A *submachine state* refers to an entire State Machine that can be nested within a state.

A *root state* is a state that does not belong to any composite state (*e.g.* S1 and S2 in Fig. 1).

**Final states** A final state (*e.g.* the right-most state in the lower region of S2 in Fig. 1) is a special kind of state which, when entered, signifies that the enclosing region has completed.

**Behaviours** Behaviours may be defined when entering states, when exiting states, while being in states (when states have a do behaviour) or when firing transitions. The entry behaviour is executed either when the state is entered through an external transition (*i.e.* when the arrow crosses the state border or touches it from outside, see below), or if that state is the target of a transition from an initial pseudostate belonging to a composite state that is itself entered through an external transition. The exit behaviour is executed either when the state is exited through an external transition or, in the case of a state belonging to a composite state, when that composite state is itself exited. The do behaviour is executed only after the execution of the entry behaviour of the state, and continues to be executing (in parallel with others behaviours, if any)

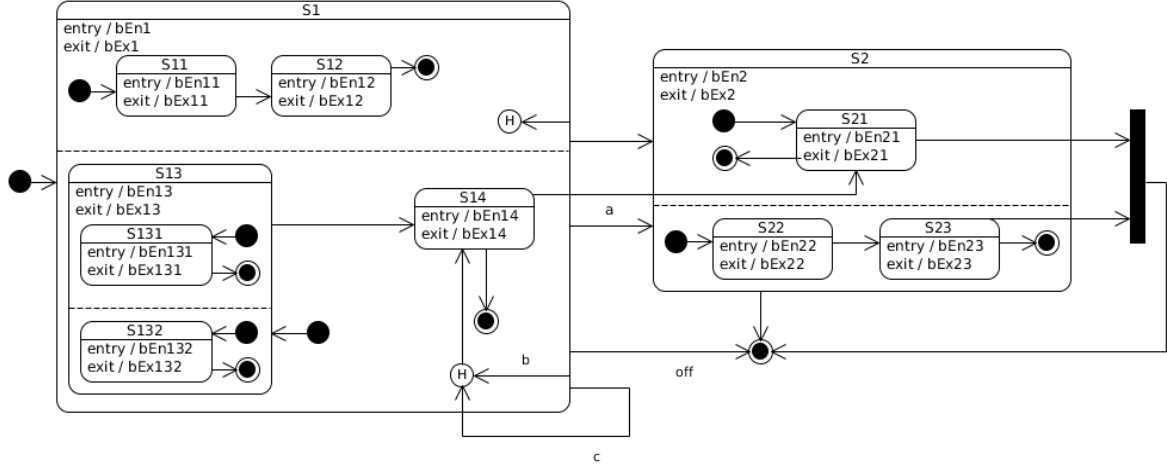


Fig. 1. Example of a state machine diagram

until it completes or the state is exited. (In Section 4.4 we explain how to take into account concurrently executable actions.)

**PseudoStates** The difference between a state and a pseudostate is that the active system state can be a state (or several states), but not a pseudostate. UML defines different kinds of pseudostates, described as follows.

**Initial** An initial pseudostate is the starting point of a region. Each composite state (simple or orthogonal) and each state machine may have an initial pseudostate. A transition outgoing from an initial pseudostate may have an associated effect behaviour but not an associated event or guard.

**History** Only composite states can have, at most, one history pseudostate in each region. UML defines two kinds of history pseudostate: *shallow history* pseudostates and *deep history* pseudostates.

A shallow history pseudostate “is a kind of variable that represents the most recent active state configuration of its containing State, but not the substates of that substate” [OMG15, Section Pseudostate and PseudostateKind, p.310], which means that the pseudostate saves only the latest visited state inside its containing composite state. Shallow history pseudostates are depicted using an “H”. In Fig. 1, there is one shallow history pseudostate in each of the two regions of S1.

In contrast, a deep history pseudostate saves the most recent active state configuration of all visited states inside the containing composite state (the state configuration is restored when a transition is terminating on the pseudostate).

**Fork and join** UML defines two kinds of pseudostates that allow the merge or the split of transitions. The join pseudostate allows many transitions (originating from states of different orthogonal regions) to be merged into one transition. The outgoing transition from a join pseudostate is executed only after the execution of all incoming transitions. The transitions that terminate on a join pseudostate cannot have a guard and/or an event. The fork pseudostate allows one transition to be split into transitions that terminate on states in regions. As the join pseudostate, the transition that terminates on the fork pseudostate cannot have a guard or event. For example, in Fig. 1, when the system is jointly in states S21 and S23, the join pseudostate can directly lead to the root final state. We leave out “implicit” forks and joins, *i.e.* that do not explicitly enter or exit all regions of a composite state. In a composite state with three regions, an implicit fork could have two incoming transitions, one coming from the first region, one coming from the second, but no transition coming from the third region.

**Transitions** UML defines three kinds of transitions: external, local and internal.

The UML specification seems to be ambiguous w.r.t. transition kinds, with two contradictory definitions (Section 14.2.3.8.1 on p.312 and Section 14.2.4.10 on p.332 in [OMG15]). We choose here to follow the (informal) semantics defined in [OMG15, Section 14.2.4.10, p.332], as it refers to graphical examples that make the illustration much more clear than the ambiguous text.

An internal transition is not depicted explicitly in the SMD ([OMG15, p.332]); still, they can be defined in internal transitions compartments ([OMG15, Section 14.2.4.5, p.317]) where they can have a guard and an associated behaviour. In order to keep our translation simple, we leave internal transitions out, and discuss in Section 4.6 how to integrate them back.

A local transition “can originate from the border of the containing composite State, or one of its entry points, or from a Vertex within the composite State” ([OMG15, p.332]). It does neither execute the exit nor the entry behaviour of its containing state; graphically, the arrow touches from inside but does not cross the border of the containing composite state. For example, the transition to the history pseudostate of the lower region of S1 and labelled with event **b** in Fig. 1 is a local transition.

An external transition is a transition that is neither an internal transition nor a local transition. For example, the transition to the history pseudostate of the lower region of S1 and labelled with event **c** in Fig. 1 is an external transition. All other transitions of Fig. 1 are external, with the exception of the transition leading to the history pseudostate of the upper region of S1.

*High-level transitions* are transitions originating from a composite state.

We also emphasize the transitions called “inter-level transitions”, that are a kind of transitions that cross the border of some composite state. For example, the transition from S14 to S21 in Fig. 1 is an inter-level transition. Many works in the literature do not consider inter-level transitions (e.g. [PG00, NB02, TZ05, CKZ11, JS15]), whereas our translation does.

Each kind of transition can have a guard (e.g. `track ≠ trackCount` on the self-transition below BUSY in Fig. 3), an event (e.g. `stop` on the top-most transition between BUSY and NONPLAYING in Fig. 3), a behaviour (e.g. `track ++` on the self-transition below BUSY in Fig. 3), and can be a completion transition (a transition without event) or a transition with event.

The order of exit behaviours execution while exiting a state by a transition is as follows: if the source state of the transition (a completion transition or a transition with event) is a simple state then we need to execute only the exit behaviour of this state. If the source state of the transition is a composite state, in the case of a completion transition we only execute the exit behaviour of the composite state. In the case of a transition with event, “When exiting from a composite State, exit commences with the innermost State in the active state configuration. This means that exit Behaviors are executed in sequence starting with the innermost active State.” [OMG15, Section State, p.308] This rule is applied to each region of the composite state, if it is an orthogonal state.

The order of entry behaviours execution while entering a state by a transition is symmetrical: if the target state is a simple state then we execute the entry behaviour of this state. If the target state is a composite state, then we execute at first its entry behaviour after which, for each of its regions, we execute in sequence the entry behaviours of all (direct and indirect) substates of that region that are initial states (i.e. states that are target of an initial pseudostate). The execution of the behaviours of the various regions of a composite state is performed in a concurrent manner.

The *run-to-completion step* in the SMD is the full execution of the behaviours associated with a transition with an event, or a completion transition. That is, the guard is tested, the exit behaviours of the source state(s) are executed (if any), the behaviour associated with the transition is executed, and the entry behaviours of the target state(s) are executed (if any).

The UML specification also defines a notion of *transition selection*: whenever an event is dispatched, zero, one or more transition(s) can be executed. In short, completion events have dispatching priority, i.e. they are dispatched ahead of any pending event occurrences in the event pool. However, the relative order between completion transitions is unspecified. For a given event, the enabled transitions are those who can be triggered by this event, such that all source states are active, and the guard associated to which is satisfied. Then, a transition *t* can fire provided no transition is enabled in a (possibly indirect) substate of the source state(s) of *t*. This has two implications: first, several transitions can fire together during the same run-to-completion step, as long as they are in different regions. Second, there may be some non-determinism, e.g. in the case of a simple state with two outgoing transitions with the same trigger, and guards that may be both satisfied (in which case only a single transition will fire). Also note that, whenever a dispatched event matches no enabled transition, then the event is just discarded.

### 2.1.2. Formalisation and Assumptions

Let us present the set of elements of UML SMDs that we take into account in our translation and the assumptions that we made on some definitions of UML. The set of UML SMD elements that we consider in this work is the following.

- Simple states, final states, simple composite states and orthogonal composite states;
- Entry, exit and do behaviours, involving variables;
- Initial, shallow history, fork and join pseudostates;
- Hierarchy of states and behaviours;
- Transitions: simple, with event, with guard and activity effect, local, external, inter-level, high-level.

Note that we will discuss how to lift some of our assumptions and how to consider more syntactic constructs in [Section 4.6](#).

**States** We take into account two kinds of states, *viz.* simple and composite. Submachine states are not considered in our translation (see [Section 4.6](#) for hints).

**Behaviours** As in [[ACK12](#), [ABC14a](#)], we abstract behaviours using name  $b$  (corresponding to the actual behaviour expression) and function  $f$  to express changes induced on the system variables, that we will denote by  $\mathcal{V}$ . This abstraction captures both behaviours described using expressions (*e.g.* `fadein` in the entry behaviour of state `PLAYING` in [Fig. 3](#)), and behaviours involving modifications of variables (*e.g.* `track ++` in the behaviour associated with the self-transition on state `BUSY` in [Fig. 3](#)). The set of all behaviour expressions will be denoted by  $\mathcal{B}$ ; *none* denotes absence of behaviour expression.

The behaviour is hence denoted by  $(b, f) \in ((\mathcal{B} \cup \{\text{none}\}) \times \text{List}(\mathcal{F}))$ , *i.e.* a behaviour expression, and an ordered list of functions in  $\mathcal{F}$ , where  $\mathcal{F}$  is the set of functions assigning to one variable of  $\mathcal{V}$  a value depending on the values of the other variables. 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; we discuss it in [Section 4.6](#). We also require that only simple states have a do behaviour (see [Section 4.6](#) for a discussion).

**Initial pseudostates** We require that each region contains one and only one (direct) initial pseudostate, which has one and only one outgoing transition. In accordance to the specification [[OMG15](#)], 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.

Recall that a transition outgoing from an initial pseudostate may have an associated effect behaviour; to keep our translation simple, we leave out this aspect.

We assume a function  $\text{init}(\mathbf{s})$  (where  $\mathbf{s}$  denotes a composite or simple state) that returns the set of direct simple substates that are the target of an initial pseudostate transition, or  $\{\mathbf{s}\}$  if  $\mathbf{s}$  is a simple state. Similarly,  $\text{init}^*(\mathbf{s})$  returns the set of all (direct and indirect) simple substates that are the target of an initial pseudostate transition, or  $\{\mathbf{s}\}$  if  $\mathbf{s}$  is a simple state. We lift  $\text{init}^*$  to sets of states by returning the union of  $\text{init}^*$  for each state in the set. By extension, we denote by  $\text{init}^*(SMD)$  the set of all (direct and indirect) simple states that are the target of the (unique) initial pseudostate transition at the top level of the state machine *SMD*.

**History pseudostates** In this work, we consider only shallow history pseudostates (deep history states are discussed in [Section 4.6](#)). Furthermore, we rule out default shallow history transitions (transitions that specify the system behaviour if the composite state was never entered, or when the latest state visited is the final state). Given a history pseudostate  $H$ , we denote by  $\mathbf{r}(H)$  the region of which  $H$  is a direct substate. The SMD transitions of which  $H$  is the target can be of any kind (originating from simple or composite states, local or external, join, inter-level), with a single exception: to keep our algorithm simple, we rule out fork transitions of which one of the target states is a history pseudostate. Lifting this assumption can be done easily, but at the price of more complicated algorithmic statements (see [Section 4.6](#)).

**Final states** “Each region [...] may have its own initial Pseudostate as well as its own FinalState.” [[OMG15](#), Section 14.2.3.2, p.304]: we allow zero or one final state(s) in each region of a composite state.



We define function  $final(\mathbf{s})$  that returns the set of direct final states of all regions in composite state  $\mathbf{s}$ . Similarly,  $final^*(\mathbf{s})$  returns the set of all (direct or indirect) final states of all regions in a (composite) state  $\mathbf{s}$ , or  $\{\mathbf{s}\}$  if  $\mathbf{s}$  is a simple state. We lift  $final^*$  to sets of states by returning the union of  $final^*$  for each state in the set.

**Variables** We allow any kind of variables in any behaviour and transition guard. Such variables (Boolean, integers, lists, etc.) are often met in practice (e.g. [ZL10, ACK12, OMG15]). We do not go into details, and we assume that variables all have a type, and that expressions defined subsequently (guards, modifications of the variables in behaviours) are well-typed.

**Fork and join pseudostates** Concerning fork and join pseudostates, we make a rather important change w.r.t. the spirit of the specification (though without impact on the semantics), as follows. Consider the join pseudostate in Fig. 1 that joins transitions from states S21 and S23 to the root final state. In this situation, the UML specification considers one pseudostate (the join pseudostate) and three transitions, two leading to the join pseudostate, and one originating from the join pseudostate. In contrast, we consider a single transition with multiple source states (for join pseudostates) or multiple target states (for fork pseudostates). Then, the join and fork pseudostates themselves will not be formalised in Definition 1.

**Transitions** We take into account both external and local transitions; we also consider inter-level transitions but with a restriction on concurrency: if the transition crosses the border of the source (resp. target) state, then that state must not be an orthogonal composite state. Each transition can be labelled by one or more event(s); in order to keep our algorithm simple, we consider only transitions with one event (details on how to relax this assumption can be found in Section 4.6.1).

We reuse from [ABC14a] 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. For example, the level state of the transition from state PLAYING to PAUSED in Fig. 3 is BUSY. The level state is the SMD itself if both the source and the target of the transition are root states. For example, consider the transition from BUSY to NONPLAYING labelled with event stop in Fig. 3: the level state of this transition is the state machine itself. The concept of level state allows us to differentiate between local and external transitions. For example, in Fig. 1, the level state of the transition labelled with  $\mathbf{b}$  to the history pseudostate of the lower region of S1 is S1 (since this transition is a local transition) whereas the level state of the transition labelled with  $\mathbf{c}$  to the same history pseudostate is S.

Often, the level state of a transition is also the least common ancestor of the source and target states, *i.e.*  $LCA(\mathbf{S1}, \mathbf{S2})$ . (Recall that “the operation  $LCA(\mathbf{S1}, \mathbf{S2})$  returns the Region that is the least common ancestor of Vertices S1 and S2, based on the StateMachine containment hierarchy” [OMG15, p.356].) An example where the  $LCA$  does not match the concept of level state is the transition labelled with  $\mathbf{c}$  from S1 to the history pseudostate H of the lower region of S1 in Fig. 1. Indeed,  $LCA(\mathbf{S1}, \mathbf{H})$  is S1, whereas the level state of the same transition is S (the SMD itself).

**Formalisation of an SMD** We formalise the syntax of state machines below. Our definition of the UML state machines syntax does not differ from the UML, with the exception of some syntactic constructs that are discarded, and some assumptions that we made (and that are explicitly mentioned in our work). However, our presentation of the syntax differs, so as to ease our subsequent translation. For example, forks and joins in our definition are complex transitions (with multiple source and target states), whereas in the UML specification forks and joins are just pseudostates, *i.e.* nodes.

**Definition 1.** A *State Machine Diagram* is a tuple

$$SMD = (\mathcal{S}, \mathcal{B}, \mathcal{E}, \mathcal{V}, \mathcal{P}, \mathcal{N}, \mathcal{X}, \mathcal{D}, SubStates, \mathcal{T}), \text{ where}$$

1.  $\mathcal{S}$  is a set of states (including pseudostates),
2.  $\mathcal{B}$  is a set of behaviour expressions,
3.  $\mathcal{E}$  is a set of events,
4.  $\mathcal{V} = \{V_1, \dots, V_{N_V}\}$  is a set of  $N_V$  variables (for some  $N_V \in \mathbb{N}$ ),
5.  $\mathcal{P} : \mathcal{S} \rightarrow Pr$  associates with each state a single property within  $Pr = \{isSimpleNotFinal, isComposite, isFinal, isInit, isHistory\}$ ,

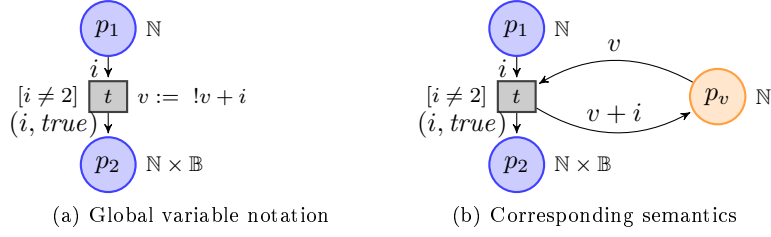


Fig. 2. Global variable notation and corresponding semantics

6.  $\mathcal{N} : \mathcal{S} \rightarrow ((\mathcal{B} \cup \{none\}) \times List(\mathcal{F}))$  associates with each state an entry behaviour, *i.e.* an ordered list of functions in  $\mathcal{F}$ , where  $\mathcal{F}$  is the set of functions assigning to one variable of  $\mathcal{V}$  a value depending on the values of the other variables,
7.  $\mathcal{X} : \mathcal{S} \rightarrow ((\mathcal{B} \cup \{none\}) \times List(\mathcal{F}))$  associates with each state an exit behaviour,
8.  $\mathcal{D} : \mathcal{S} \rightarrow ((\mathcal{B} \cup \{none\}) \times List(\mathcal{F}))$  associates with each state a do behaviour,
9.  $SubStates : \mathcal{S} \rightarrow 2^{\mathcal{S}}$  associates with each state the set of its direct substates,
10.  $\mathcal{T}$  is a set of transitions of the form  $\mathbf{t} = (\mathbf{S}_1, e, g, (b, f), sLevel, \mathbf{S}_2)$ , where
  - $\mathbf{S}_1, \mathbf{S}_2 \subseteq \mathcal{S}$  are the source and target set of states respectively, with the following constraints:
    - $\mathbf{S}_1$  or  $\mathbf{S}_2$  may contain exactly one state (in the case of simple transitions between two states), or more than one state (in case of fork or join transitions);
    - if  $\mathbf{S}_1$  (resp.  $\mathbf{S}_2$ ) contains more than one state, each of these states must be a direct substate of a different region of the same composite state; and this composite state cannot have more regions than the number of states in  $\mathbf{S}_1$  (resp.  $\mathbf{S}_2$ );
  - $e \in \mathcal{E} \cup \{noEvent\}$ , where *noEvent* is a special value denoting that the transition has no event,
  - $g$  is the guard (*i.e.* a Boolean expression over the variables of  $\mathcal{V}$ ),
  - $(b, f) \in ((\mathcal{B} \cup \{none\}) \times List(\mathcal{F}))$  is the behaviour to be executed while firing the transition, and
  - $sLevel \in \mathcal{S}$  is the level state containing the transition, and such that any state belonging to  $\mathbf{S}_1 \cup \mathbf{S}_2$  is a (possibly indirect) substate of  $sLevel$ .

Note that the functions mentioned earlier in this section (*e.g.* *init\**, *final\**, etc.) can be defined using the elements of the tuple given in [Definition 1](#).

## 2.2. Coloured Petri Nets with Global Variables

Let  $\mathbb{N}$  denote the set of natural numbers, and  $\mathbb{B}$  the Boolean type.

Coloured Petri nets (CPNs) [JK09] are a kind of automaton represented by a bipartite graph with two kinds of nodes, *viz.* places (drawn as circles with the name inside, *e.g.*  $p_1$  in [Fig. 2\(a\)](#)) and transitions (drawn as rectangles with the name inside, *e.g.*  $t$  in [Fig. 2\(a\)](#)). Note that, in most of our figures (exported from CPN Tools), places are often depicted using ellipses instead of circles. Places can contain tokens, possibly of a complex value; the tokens in a place should be of the place type (*e.g.* type  $\mathbb{N} \times \mathbb{B}$  in [Fig. 2\(a\)](#)). A special type, null (denoted by  $\bullet$ ), is used for tokens that do not carry any value. Directed arcs connect places to transitions (input arcs), and transitions to places (output arcs). Arcs are labelled by an expression where some variables of the corresponding place type may appear (*e.g.*  $v + i$  in [Fig. 2\(b\)](#)). Transitions may have a guard (*e.g.*  $[i \neq 2]$ ), that is a condition to be met for the transition to fire (no guard shown means true).

**Global variables and code segment** 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 along a “code segment”. An example is given in [Fig. 2\(a\)](#):  $v$  is a global variable of type  $\mathbb{N}$ . This variable is updated in the code segment associated with transition  $t$  to the expression  $!v + i$  (where  $i$  is the value of the token coming from  $p_1$ );  $!v$  is the CPN ML notation denoting an access to the value of  $v$ .

Such global variables updated in code segments are supported by some tools (such as CPN Tools, with some limitations though). 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.

This construction is equivalent to the one in Fig. 2(b). When a global variable is read in a guard, the token with value  $v + i$  is put back on place  $p_v$ .

Let us now define formally coloured Petri nets extended with global variables.

**Definition 2.** A *coloured Petri net extended with global variables* (hereafter CPN) is a tuple

$$CPN = (P, T, A, B, V, C, G, E, MI, VI), \text{ where}$$

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.  $B$  is a finite set of non empty *colour sets* (types),
5.  $V$  is a finite set of *typed variables* such that  $\forall v \in V, Type(v) \in B$ ,
6.  $C : P \rightarrow B$  is a *colour set function* assigning a colour set to each place,
7.  $G : T \rightarrow Expr(V)$  is a *guard function* assigning a guard to each transition such that  $Type(G(t)) = \mathbb{B}$ , and  $Var[G(t)] \subseteq V$ ,
8.  $E : A \rightarrow Expr(V)$  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,
9.  $MI : P \rightarrow Expr(V)$  is a *marking initialization function* assigning an initial marking to each place such that  $Type(MI(p)) = C(p)_{MS}$ , and
10.  $VI$  is a *variable initialization function* assigning to each variable an initial value such that, for all  $v \in V$ ,  $Type(VI(v)) = Type(v)$ .

**Current state** We do not recall the semantics of CPNs, that can be found in, e.g. [JK09]. In short, the *marking* is the information on which tokens are present in which places together with their values. 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). In our setting of CPNs extended with global variables, the current state is the marking and the value of the global variables.

### 3. A Motivating Example of a CD Player

In this section, we exemplify UML state machines using a CD player described using a UML state machine. This CD player will be used as a running example; its verification will be carried out in Section 6.

The following description represents a specification of the CD player.

*This device is a CD player with five buttons: load, play, pause, stop, off. The typical use of the CD player is as follows: before the user starts using the CD player for the first time, the drawer is closed, with no CD inside and no button is pressed. When the user presses the load button, the drawer opens and if the user presses it again the drawer closes. If the drawer is closed and there is a CD inside, then the player is ready to play, and the track is set to the first track; otherwise nothing happens. When the user presses the play button, if there is no CD in the drawer then nothing happens; otherwise the player starts reading the CD from the beginning. If the user presses the pause button while reading CD, the player stops and the drawer remains closed. If the user presses the pause button again, the player continues reading the CD. The player features a light; this light is on when the player reads the CD, and off when the reading is paused or when there is no CD in the drawer. Finally, when the user presses the stop button, the player stops reading and the user can get back the CD (by pressing the load button) or start reading again (by pressing the play button). At any time, the CD player can be turned off by pressing the off button.*

Fig. 3 shows the UML state machine diagram of the CD player.<sup>2</sup>

The Boolean variable `present` encodes whether there is a CD in the player, and the integer variable

<sup>2</sup> This example is inspired from a model in [ZL10], augmented with some features such as concurrent states.

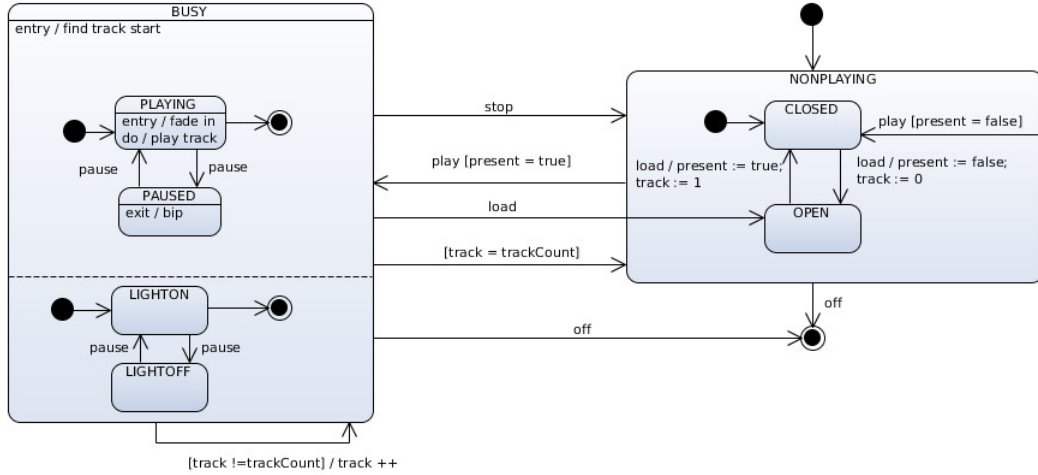


Fig. 3. The CD-Player example

`track` encodes the current track played by the player. We assume that `trackCount` is a constant set to some predefined value (*e.g.* 5).

Let us informally describe the role of each simple state of the CD player:

- **CLOSED** denotes that the player is currently non-playing and closed.
- **OPEN** denotes that the player is currently non-playing and open.
- **PLAYING** denotes that the player is reading a track (*i.e.* playing music).
- **PAUSED** denotes that the player is in pause mode, *i.e.* the track is suspended.
- **LIGNTOFF** denotes that a light (*e.g.* a LED on the player front) is set.
- **LIGNTOFF** denotes that the light is off.

The player switches from the state **NONPLAYING** to **BUSY** (and vice versa) depending on which button the user presses. The player switches from state **CLOSED** to **OPEN** when the user presses the `load` button and vice versa. The player changes from the state **NONPLAYING** to the state **BUSY** when the user presses the `play` button, and there is a CD in the player (guard `present = true`). We can switch from the state **PLAYING** to the state **PAUSED** by pressing the `pause` button. The same button allows the change from the state **LIGNTOFF** and **LIGNTOFF**. If the user presses the `load` button, the player reads all tracks of the CD (depicted by a transition with a guard on `track`); but if the user presses the `load` button while playing, then the player switches from the state **BUSY** to the state **OPEN**. The user can switch off the device anytime (transition `off` to the root final state).

This example is relatively simple: two variables, two composite states, six non-final simple states, a few transitions. Still, it is not clear at all whether the following properties hold in any configuration of the system:

1. “the CD player cannot be both closed and open”;
2. “whenever the CD player is in state **PLAYING**, there is a CD in the player”;
3. “whenever the player is paused, the light is off”;
4. “the value of `track` never exceeds `trackCount`”.

These properties cannot be formally verified without a formal semantics. In the following, we will formalise SMDs so as to be able to verify properties such as these.

## 4. Formalizing UML State Machines

We present in this section our translation. We start by introducing the general idea (Section 4.1). We then introduce a method to name the CPN places and transitions resulting from our translation (Section 4.2). We then present the translation algorithm (Section 4.3), and focus especially on the translation of the hierarchy of

entry and exit behaviours (Section 4.4). We apply our algorithm to sample situations (Section 4.5). Finally, we briefly describe how to extend the set of UML constructs considered in this work, when applicable (Section 4.6). We will extend our scheme to encode priorities between transitions in Section 5.

#### 4.1. General Scheme

Let us first present the general view on our translation from UML SMDs to CPNs.

**General idea** We define a translation scheme where simple states (including final states) are translated into places, whereas SMD transitions are translated into CPN transitions.

In fact, each run-to-completion step is encoded in our scheme by a single CPN transition, which correctly encodes the “unit” aspect of a run-to-completion step. We believe that this a rather elegant solution. In particular, it differs from previous solutions (*e.g.* [ZL10, CKZ11, ABC14a]), where some unwanted interleaving between independent transitions could happen. Note that our solution still allows for interleaving between the various entry or exit behaviours of concurrent states involved in a single transition.

Although we use coloured Petri nets, all the tokens of the resulting CPN are of type null (“•”); however, our CPN is still coloured due to the use of global variables and, most importantly, due to the extensive use of code segments attached to CPN transitions. For sake of clarity, we do not give the type of the arcs and places in our figures, as it is always “•”.

Finally, although we correctly encode the hierarchy induced by the composite states, our translation can be seen as a “flattening” of the state machine, as composite states are not preserved as such (only simple states are encoded into places). This is in contrast with other approaches using hierarchical (coloured) Petri nets (*e.g.* [CKZ11]). Preserving the hierarchy in the destination formalism is the subject of future work.

**Entry and exit behaviours** Recall that we abstract behaviours using a behaviour expression  $(b, f)$ , with  $b$  a behaviour expression and  $f$  a list of functions modifying the values of the global variables. The translation of the functions is straightforward (as explained in more details in Section 4.4): each modification of the SMD variables along an SMD transition will be translated to the very same modification of the corresponding CPN global variables in the code segment associated with the CPN transition(s) encoding the SMD transition. That is, each time we exit a state or enter a state by an SMD transition, we add the exit or entry behaviour (if any) into the code segment of the associated CPN transition. However, to keep this work simple (and contrarily to [ACK12, ABC14a]), we simply drop the behaviour expressions. We discuss how to reintroduce them in Section 4.6.

**Do behaviours** We translate the do behaviour into a CPN transition (more details in Section 4.4), that contains in its associated code segment a function encoding the behaviour functions  $f$ . Then, we add an arc from the state place to the behaviour transition, and another arc from the behaviour transition to the state place. This is consistent with our assumption that a do behaviour can be executed as many times as wished (Section 2.1.2). We discuss how to consider variants of this assumption in Section 4.6. Again, we leave out the behaviour expression  $b$ .

**Composite states and pseudostates** In contrast to simple states, we translate neither composite states, nor pseudostates (such as fork, join, initial or history pseudostate) into CPN places (more details in Section 4.5). Recall that the difference between pseudostates and states is the fact that the current active state of the system can be a state but cannot be a pseudostate. Hence, the fact that pseudostates are not explicitly translated into CPN places is in line with the fact that the active state of an SMD is a (set of) simple state(s). However, their behaviours and all their simple substates (in the case of composite states) will be translated to places or transitions.

Since join/fork pseudostates are formalised as transitions with multiple source/target states (see Definition 1), they require no specific treatment: each fork and join pseudostate is represented by a CPN transition that will link between the places encoding the source states(s) and target state(s).

**Shallow history pseudostates** Recall that, when entered, a shallow history pseudostate in a composite state redirects to the latest state visited within this composite state (details about the translation with an example in Section 4.5.6). This will be achieved using a global variable (one per history pseudostate) that

| SMD element                       | Situation                                | CPN element | CPN naming        |
|-----------------------------------|--|-------------|-------------------|
| Variable $v$                      | -  | variable    | $v$               |
| Final state                       | Inside an orthogonal state $s$           | place       | $s\_iF$           |
|                                   | Inside a simple composite state $s$      | place       | $sF$              |
|                                   | The root final state                     | place       | $F$               |
| Simple state $s$                  | -  | place       | $s$               |
| History pseudostate               | Inside one region                        | place       | $Sh$              |
|                                   | Inside multiple regions                  | place       | $S\_ih$           |
| do behaviour of the state $s$     | -  | transition  | $s\_do$           |
| Transition without event          | Single transition between $S1$ & $S2$    | transition  | $S1\_S2$          |
|                                   | Multiple transitions between $S1$ & $S2$ | transition  | $S1\_S2\_i$       |
| Transition with event $e$         | Single transition between $S1$ & $S2$    | transition  | $S1\_e\_S2\_j$    |
|                                   | Multiple transitions between $S1$ & $S2$ | transition  | $S1\_e\_S2\_i\_j$ |
| Transition to history pseudostate | Without event                            | transition  | $S1\_H\_S2$       |
|                                   | With event $e$                           | transition  | $S1\_He\_S2$      |

Table 1. Conventions for naming CPN elements

is updated to the current active state each time the place encoding this state (or one of its substates, for composite states) is entered. This update is done in the code segment of any transition leading to this place.

**Variables** Here, we use the concept of global variables that we defined in the formalism of CPNs (see Section 2.2) to encode the value of SMD variables. This is convenient, as global variables in CPNs can be read in guards and updated along transitions, the same way as the SMD variables.

**Handling transitions** A main issue in our translation is to encode transitions between composite states with different regions (orthogonal states), in particular high-level transitions, the variants of forks, joins and transitions with event (in Section 4.5 we explain the translation of transitions with events or without event and with different states, such as simple states or composite states). Each such UML transition may in fact correspond to a large number of transitions. For example, in Fig. 1 taking the SMD transition labelled with event  $a$  corresponds to 18 ways to leave  $S1$  (3 possible active states in the upper region of  $S1$ , multiplied by 4 in  $S13$  plus 2 others in the lower region), and hence in 18 CPN transitions. Given an SMD transition  $t$ , let us denote by *combinations*( $t$ ) the function that computes all possible combinations of outgoing simple states. Each such combination will be translated to one CPN transition. For example, consider the transition from  $S1$  to  $S2$  labelled with  $b$  in Fig. 8(a). The states  $S11$  and  $S12$  form one combination; hence, a corresponding CPN transition will be created, called  $S1\_b\_S2\_1$  in Fig. 8(b) (1 denotes that this is the first combination). In the case of a UML transition without event, we translate it to a single CPN transition, as this combinatorial explosion does not happen in this case: indeed, to exit a composite state (with one or multiple regions) using a completion transition, each region needs to be in its final state, which yields a single CPN transition.

## 4.2. Naming CPN Elements

We present in Table 1 the naming conventions for the CPN variables, places and transitions corresponding to the various elements of UML SMDs taken into account in our translation. Note that  $i$  denotes either the  $i$ th region of an orthogonal composite state, or the  $i$ th transition between  $S1$  and  $S2$ ;  $j$  denotes the  $j$ th combination considered for this particular transition (we assume that the regions, the transitions and the combinations are ordered using some lexicographic order).

### 4.3. Translation Algorithm

We describe in [Algorithm 1](#) the procedure that translates a UML state machine model to a coloured Petri net model. Throughout the algorithm, the plain elements (places and transitions of the resulting CPN) denote elements added by the current statement, whereas dashed elements denote elements added by previous statements. We divide the algorithm into three steps.

**First step** The first step ([line 1–line 4](#)) deals with the translation of states with their do behaviour, if any. For each simple state in the SMD, we add a new place named with the name of the state (as defined in [Section 4.2](#)). Then, we add a transition encoding the do behaviour, if any ([line 4](#)).

**Second step** The second step of the algorithm ([line 5–line 18](#)) deals with the translation of the transitions such that no target state is a history pseudostate. In this step, we have two parts. The first part ([line 6–line 12](#)) deals with the transitions with an event. Recall from [Section 4.1](#) that, given a transition  $\mathbf{t}$ , function *combinations*( $\mathbf{t}$ ) returns the list of all possible combinations of outgoing simple states. In terms of data structures, we assume this result is in the form of a list of sets of states; each set of states represents a configuration of the composite state, *i.e.* one active state in each orthogonal region of the composite state, in a hierarchical manner. For each such combination, we first add a corresponding CPN transition ([line 8](#)). The first **foreach** loop ([line 9–line 10](#)) links the place corresponding to each simple state in the considered combination to the CPN transition encoding the considered SMD transition; the second **foreach** loop ([line 11–line 12](#)) links the CPN transition encoding the considered SMD transition to the places encoding each of the simple states pointed by an initial pseudostate in the target state of the SMD transition.

In the second part of the second step ([line 13–line 18](#)), we deal with the UML transitions without event. First, we add a CPN transition that will correspond to the UML transition ([line 13](#)). Then, we link all places corresponding to a final state of the source of the UML transition to the CPN transition ([line 15–line 16](#)). Finally, just as for transitions with an event (see above), we link the CPN transition to the places encoding each of the simple states pointed by an initial pseudostate in the target state of the SMD transition ([line 17–line 18](#)).

**Third step** The last step ([line 19–line 26](#)) deals with the transition targeting a single shallow history pseudostate. Recall from [Section 2.1.2](#) that we rule out fork transitions targeting history pseudostates, hence the target state is necessarily unique here. In contrast, the source of the transition can be multiple, and one or more source state(s) can be a composite state, which requires to enumerate again the various combinations of source simple states ([line 20](#)).

Then, for each possible target of the history pseudostate (“ $\mathbf{s}_2$ ”), *i.e.* for each possible non-final state in the region of the history pseudostate, we add a dedicated transition ([line 22](#)). This CPN transition will be guarded later on (by function *DecorateH*() ) so that it can only be taken if  $\mathbf{s}_2$  was the latest visited state in this region. This transition has as sources the places encoding all simple states in the considered combination ([line 24](#)), and as target the places encoding all initial states of the target of the history pseudostate ([line 26](#)); indeed, there can be more than one if the target of the history pseudostate is a composite state.

**Initial place** We add a special place, that will contain the (unique) initial token ([line 27](#)). This place is connected via a transition to all initial simple states of the SMD ([line 28–line 29](#)). The transition will be decorated (by function *AddBehaviours*()) with all the necessary entry behaviours that shall be performed when initialising the SMD. For example, in the SMD in [Fig. 6\(a\)](#), the unique initial simple state is  $\mathbf{S1}$ . Then, in the CPN in [Fig. 6\(b\)](#), a place *init* is connected to  $\mathbf{S1}$  via a transition along which the appropriate entry behaviours (of  $\mathbf{S}$  and  $\mathbf{S1}$ ) are performed.

**Handling guards and behaviours** We encode the addition of the behaviours and guards to the appropriate code segments using three functions *AddGuards*(), *DecorateH*() and *AddBehaviours*() ([line 30](#)).



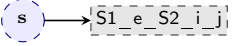

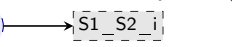
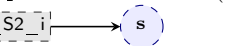
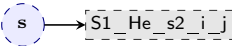
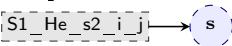
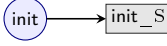
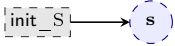
The first function *AddGuards*() is straightforward: it simply adds to any CPN transition the guard of the corresponding SMD transition. For example, given the guard on the left-most transition guarded by  $i = 0$  in the SMD in [Fig. 7\(a\)](#), *AddGuards*() adds the equivalent guard to the corresponding CPN transition in [Fig. 7\(b\)](#), *i.e.*  $[!i = 0]$ .

The second function *DecorateH*() adds the necessary guards and variable updates to the CPN transitions, as follows. First, any CPN transition  $\mathbf{S1\_He\_s2\_i\_j}$  added by [Algorithm 1](#) ([line 22](#)) is decorated with a

**Algorithm 1:** Translating an SMD  $(S, \mathcal{B}, \mathcal{E}, \mathcal{V}, \mathcal{P}, \mathcal{N}, \mathcal{X}, \mathcal{D}, \text{SubStates}, \mathcal{T})$  into a CPN

---

```

// Step 1: Simple states and their do behaviours
1 foreach simple state  $s \in S$  do
2   Add 
3   if  $s$  has a do behaviour then
4     Add 
// Step2: Transitions
5 foreach transition  $t = (S_1, e, g, (b, f), sLevel, S_2) \in \mathcal{T}$  with no history pseudostate within  $S_2$  do
6   if  $e \neq \text{noEvent}$  then
7     foreach  $c \in \text{combinations}(t)$  do
8       Add  $[S_1\_e\_S_2\_i\_j]$  //  $j$ th combination of the  $i$ th transition from  $S_1$  to  $S_2$  with  $e$ 
9       foreach simple state  $s \in c$  do
10        Add 
11        foreach simple state  $s \in \text{init}^*(S_2)$  do
12          Add 
13      else
14        Add  $[S_1\_S_2\_i]$  //  $i$ th transition without event from  $S_1$  to  $S_2$ 
15        foreach simple state  $s \in \text{final}^*(S_1)$  do
16          Add 
17        foreach simple state  $s \in \text{init}^*(S_2)$  do
18          Add 
// Step 3: History states
19 foreach transition  $t = (S_1, e, g, (b, f), sLevel, H) \in \mathcal{T}$  do
20   foreach  $c \in \text{combinations}(t)$  do
21     foreach state  $s_2$  such that  $s_2$  is a non-final direct substate of  $r(H)$  do
22       Add  $[S_1\_He\_s_2\_i\_j]$  //  $j$ th combination of the  $i$ th transition from  $S_1$  to  $H$  with  $e$ 
23       foreach state  $s \in c$  do
24         Add 
25       foreach simple state  $s \in \text{init}^*(s_2)$  do
26         Add 
27 Add  //  $S$  is pointed by the initial pseudostate of SMD
28 foreach  $s \in \text{init}^*(SMD)$  do
29   Add 
30 AddGuards() ; DecorateH() ; AddBehaviours();

```

---

guard “[ $S_h = s_2$ ]”, where  $S$  is the region of the history pseudostate (recall that  $S_h$  is the CPN variable encoding the history pseudostate). That is to say that the history pseudostate has target  $s_2$  only if  $s_2$  is the latest visited state in its region. Second, for any region  $S$  that features a history pseudostate, for any direct substate  $s_1$  of  $S$ , for any direct or indirect simple substate of  $s_1$ , we add the following code segment to any CPN transition leading to the place enclosing this simple state:  $S_h := “s_1”$ . This is to record that



the latest visited state in  $S$  is now  $s1$ . An exception is when  $s1$  is a final state: in that case, the code segment is not  $Sh := "s1"$ , but  $Sh := "si"$ , where  $si$  is the state target of the initial pseudostate of the region. This is consistent with the fact that, when a composite state is left by its final state, and entered through its history pseudostate, then the composite state enters using the default entry, *i.e.* via the initial pseudostate. We will exemplify the handling of history pseudostates in [Section 4.5.6](#).

The third function *AddBehaviours()* is less straightforward, and will be detailed in [Section 4.4](#).

**Defining the initial state of the CPN** Recall that the initial state is an initial marking, and an initial value for all global variables. Concerning the initial marking, we add a token to the unique initial place *init*.

Concerning the CPN global variables encoding the SMD variables, their initial value is that of the SMD; if no initial value is defined in the SMD, either the model can be considered as ill-formed, or a default initial value can be assigned (*e.g.* 0 for integers, false for Booleans, etc.).

Concerning the CPN global variables encoding the value of the history pseudostates, recall that we do not take into account the default shallow history transitions; hence, we initialise these variables to the state following the initial pseudostate of the region of this history pseudostate. That is, when a history pseudostate is entered although its region was never visited before, it will target the state following the initial pseudostate. This is consistent with the specification (“If no default history Transition is defined, then standard default entry of the Region is performed” [[OMG15](#), p.307]).

#### 4.4. Adding Behaviours to Code Segments

Let us now explain the function *AddBehaviours()*. Recall that a UML behaviour is abstracted using a pair  $(b, f)$ , where  $b$  is a behaviour expression and  $f$  is a list of functions expressing changes induced on the system variables. Also recall that we only translate  $f$ .

**Do behaviours** Each do behaviour is encoded by a CPN transition linked with the place encoding the corresponding state ([line 4](#) in [Algorithm 1](#)). Hence, *AddBehaviours()* simply needs to add to this transition the code segment equivalent to the functions  $f$ . For example, the do behaviour of  $S2$  in [Fig. 6\(a\)](#) (*i.e.*  $j = j + 1$ ) is translated into a code segment associated with the transition  $S2\_do$  in [Fig. 6\(b\)](#) (*i.e.*  $j := j + 1$ ).

**Exit and entry behaviours** A major difficulty when formalising UML SMDs is to correctly handle the exit and entry behaviours when taking an SMD transition. Function *AddBehaviours()* is responsible to appropriately decorate the code segment of the CPN transitions to take into account all entry and exit behaviours to be executed when taking an SMD transition.

Recall that, when taking an SMD transition, the exit behaviours should be executed starting from the innermost states of the source state(s) until the state containing the transition (the “level state” in [Definition 1](#)); then, the entry behaviours should be executed starting from the level state until the innermost initial state of the target state(s) of the transition.

This mechanism is relatively straightforward in the absence of concurrency, *i.e.* when composite states are not orthogonal. Consider the transition from  $S0$  to  $S1$  in the SMD in [Fig. 4\(a\)](#): when taking this transition, first the exit behaviour of  $S0$  (*i.e.*  $i = 0$ ) should be executed; second, the behaviour associated with the transition should be executed (*i.e.*  $j = 3$ ); third, the entry behaviour of  $S1$  should be executed (*i.e.*  $j = j + 1$ ), followed by that of  $S11$  (*i.e.*  $i = j * 3$ ). As a consequence, the following code segment is added to the CPN transition  $S0\_S1$  in [Fig. 4\(b\)](#) encoding this SMD transition:  $i := 0; j = 3; j := j + 1; i := j * 3$ ;

However, the execution of the behaviours becomes more complicated when concurrency is involved. Consider the transition from  $S0$  and  $S1$  in the SMD in [Fig. 5\(a\)](#). Once  $S0$  is exited and the transition behaviour is executed, then the entry behaviour of  $S1$  is executed. Then, the entry into  $S11$  and  $S12$  is done concurrently, which means that the following sequences of behaviours are possible:

1.  $exS0(); enS1(); enS11(); enS111(); enS12(); enS121()$
2.  $exS0(); enS1(); enS11(); enS12(); enS111(); enS121()$
3.  $exS0(); enS1(); enS11(); enS12(); enS121(); enS111()$
4.  $exS0(); enS1(); enS12(); enS11(); enS111(); enS121()$
5.  $exS0(); enS1(); enS12(); enS11(); enS121(); enS111()$
6.  $exS0(); enS1(); enS12(); enS121(); enS11(); enS111()$

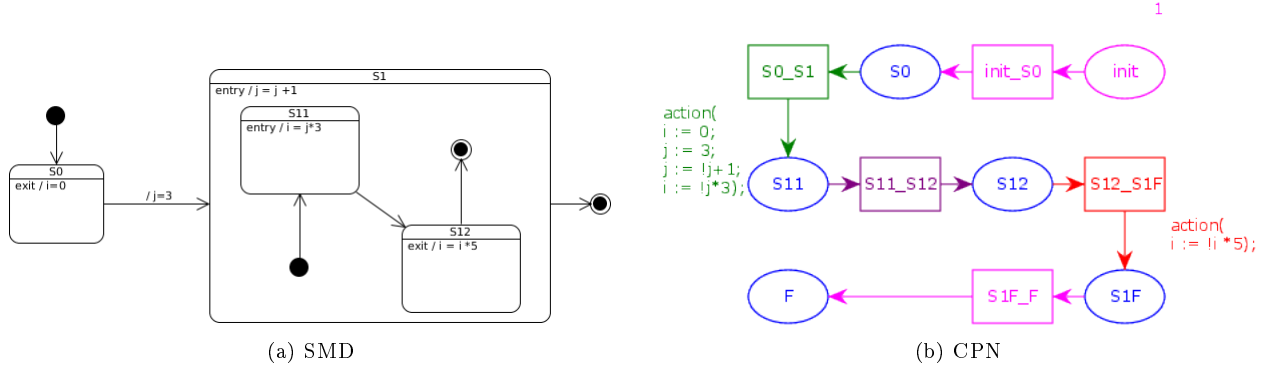


Fig. 4. Example of a state machine without concurrency, with behaviours

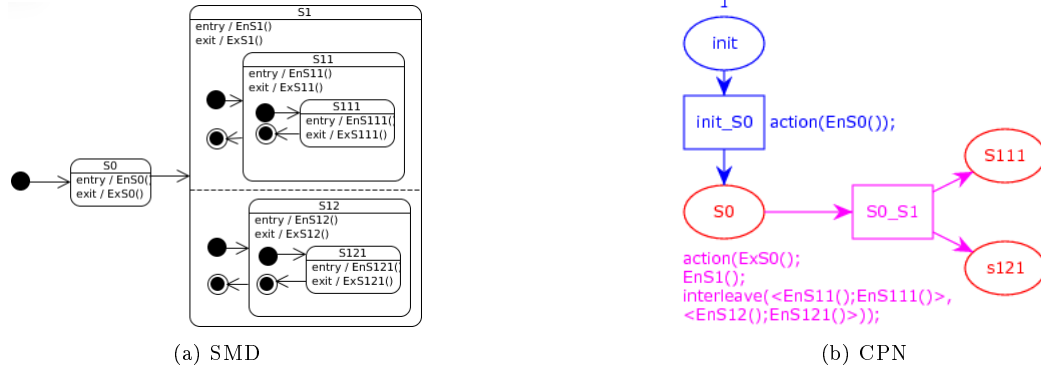


Fig. 5. Example of a state machine with concurrency and behaviours

In fact, the only requirement is that the entry behaviour of some state must occur after the entry behaviour of its containing state(s), *i.e.* the entry behaviours from the level state to each innermost initial pseudostate must be executed sequentially (and conversely for exit behaviours); however, all these possible sequences can be executed in an interleaving manner. All those scenarios are possible in the SMD model and should be represented and taken into account in the translation.

A simple solution could be to just enumerate all these possibilities, and create one CPN transition for each sequence of behaviours. Instead, we propose here a function *interleave()* that is used to model all interleavings, while keeping our representation compact. Given a list of entry (or exit) behaviours  $\text{exS1}, \dots, \text{exSn}$ , we denote by  $\langle \text{exS1}, \dots, \text{exSn} \rangle$  the sequence of behaviours executed in this particular order. Given a list of several sequences  $\text{seq}_1, \dots, \text{seq}_n$ , our function  $\text{interleave}(\text{seq}_1, \dots, \text{seq}_n)$  considers all the possible interleavings between these sequences of behaviours. That is, in a given sequence  $\text{seq}_i$ , the behaviours are executed sequentially; but the order between the behaviours of  $\text{seq}_i$  and that of other sequences can be arbitrary. In fact, this function is syntactic sugar to a list of CPN transitions, each of them containing one particular interleaving. Using this function makes our figures much clearer.<sup>3</sup>

For example, the interleaving of the above behaviours can be represented as follows using *interleave()*:  
 $\text{exS0}(); \text{enS1}(); \text{interleave}(\langle \text{enS11}(); \text{enS111}() \rangle, \langle \text{enS12}(); \text{enS121}() \rangle)$

Finally, recall that there is some non-determinism in UML in the case of concurrently executable transition

<sup>3</sup> Although we use this function in our screenshots of CPN Tools, this function is in fact not supported by CPN Tools. Hence, in our model of the CD player (Section 6), we do have to duplicate our transitions to model all possible interleavings, which make the models hard to read. This was a motivation for proposing such a syntactic sugar. Integrating such a function into CPN Tools could be either done in collaboration with the developers of CPN Tools, or using a preprocessing script.

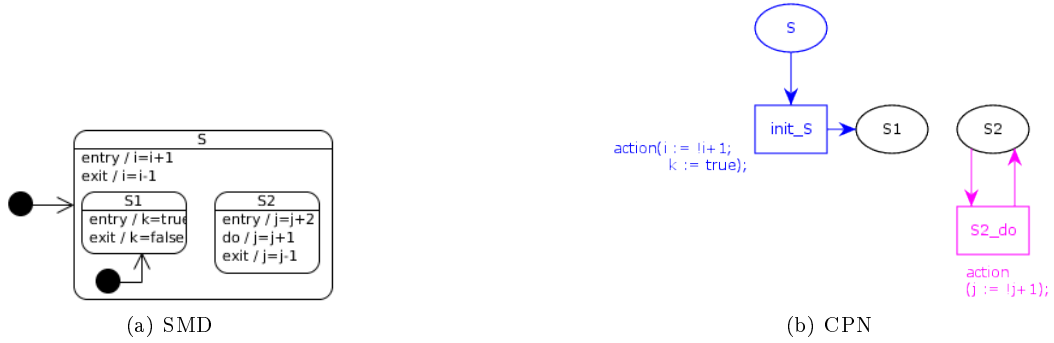


Fig. 6. Example: simple states and “do” behaviour

actions causing data races on global variables. The execution of a CPN is non-deterministic too, *i.e.* the choice of the transitions that will be fired is non-deterministic. However, in the verification phase, the model checker must explore all combinations, *i.e.* it will consider all possible executions of the state machine. That is to say that the result of the model-checking phase is of course deterministic.

Although the case of concurrently executable transition actions causes data races on global variables, these transition actions are still executed in an instantaneous manner. When we add time to our transformation (see Section 7.2), these actions will definitely be executed in 0-time. The result is (in fact) equivalent: after executing the behaviours in some order in the CPN, the various possible resulting configurations (depending on non-determinism) will be equivalent to that of the SMD after the end of the run-to-completion step.

#### 4.5. Application of the Translation to Sample Cases

We exemplify here our translation (formalised in Algorithm 1) on several situations.

##### 4.5.1. Simple States and “Do” Behaviours

**Situation considered** Fig. 6(a) shows an example of a composite state  $S$  with four simple substates, *viz.*  $S1$ ,  $S2$  the final state (SF) of  $S$ .

**Translation** The two simple states in Fig. 6(a) are translated to two corresponding places ( $S1$ ,  $S2$ ,  $SF$  and  $F$ ) in Fig. 6(b). The third place is the unique initial place. We also add a transition ( $S2\_do$  in Fig. 6(b)) corresponding to the do behaviour of  $S2$ .

##### 4.5.2. Transitions Without Events and Orthogonal Regions

**Situation considered** We consider in Fig. 7(a) transitions without event between, simple/composite and orthogonal states. In this case, exiting an orthogonal state requires to exit each region inside (through the final states), and then to enter the orthogonal state by entering each initial state of each region. Note that the entry into the substates of  $S1$  is done in a concurrent manner: that is, when we enter the state  $S1$  we execute the entry behaviour of  $S1$  after which we can execute the entry behaviour of  $S11$  then the entry behaviour of  $S12$ , or first  $S12$  and then  $S21$ .

**Translation** The transition from  $S0$  to  $S1$  is translated into the CPN transition  $S0\_S1$  in Fig. 7(b). The *interleave()* function is used to model that the entry behaviours of  $S11$  and  $S12$  are executed in a concurrent manner. The transition between  $S1$  and  $S2$  is translated into the CPN transition  $S1\_S2$  in Fig. 7(b). We exit the state  $S1$  through the final states of its regions (*i.e.*  $S1\_1F$  and  $S1\_2F$ ), since this transition is a completion transition.

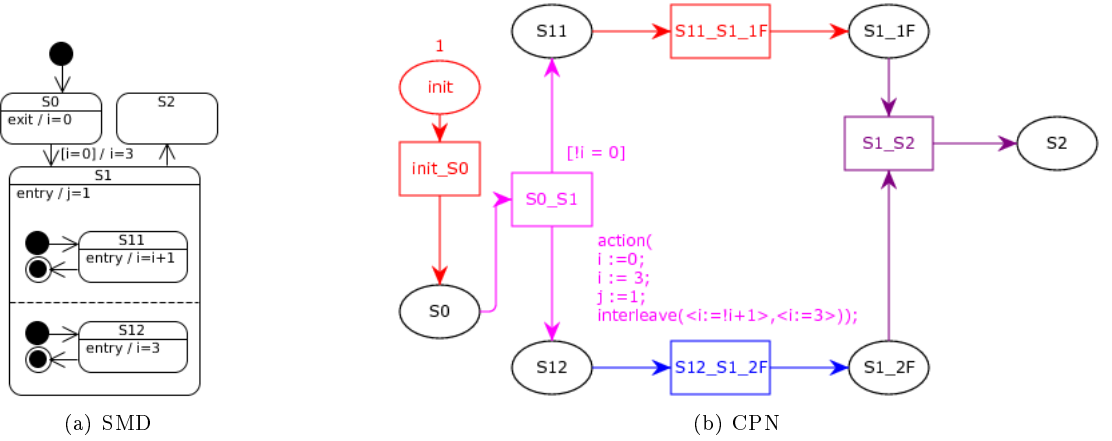


Fig. 7. Example: transitions without event and orthogonal regions

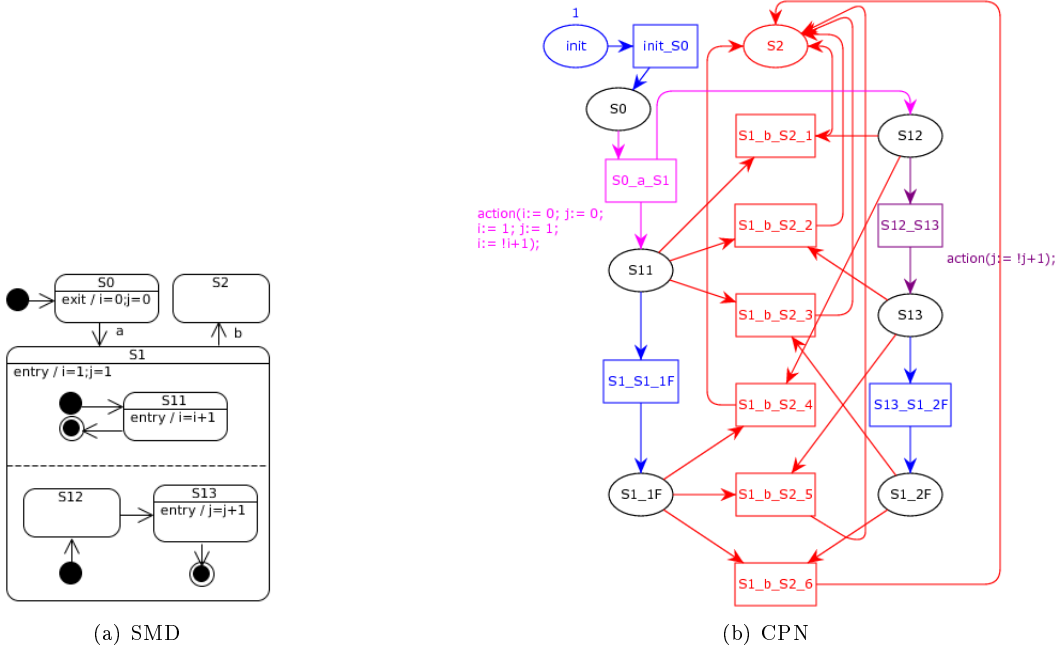


Fig. 8. Example: transitions with event and orthogonal regions

#### 4.5.3. Transitions With Events and Orthogonal Regions

**Situation considered** We consider in Fig. 8(a) transitions with event between simple/composite states. Recall that exiting an orthogonal state through a transition labelled with an event yields several combinations. When exiting S1 through the transition labelled by the event **b**, the system state can be in any of these combinations:  $\{(S11, S12), (S11, S13), (S11, S1\_2F), (S1\_1F, S12), (S1\_1F, S13), (S1\_1F, S1\_2F)\}$ .

**Translation** We give the translation in Fig. 8(b). The transition labelled with event **b** between the states S1 and S2 is modelled by six CPN transitions (one CPN transition for each combination of the simple states in the orthogonal regions). As an example, for the combination between S11 and S13, we add the transition S1\_b\_S2\_2.

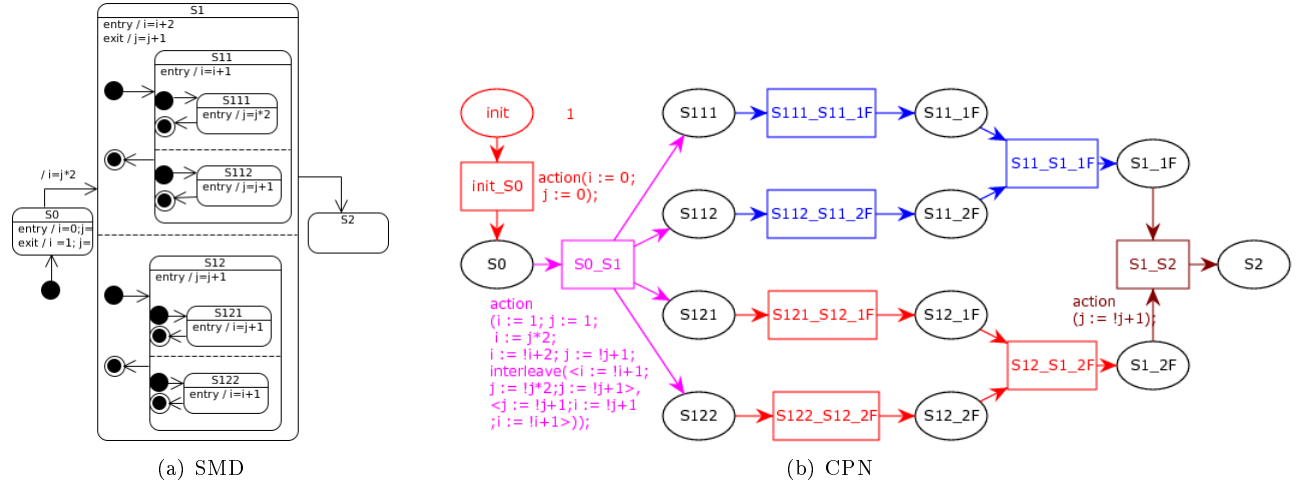


Fig. 9. Example: Hierarchy of orthogonal regions

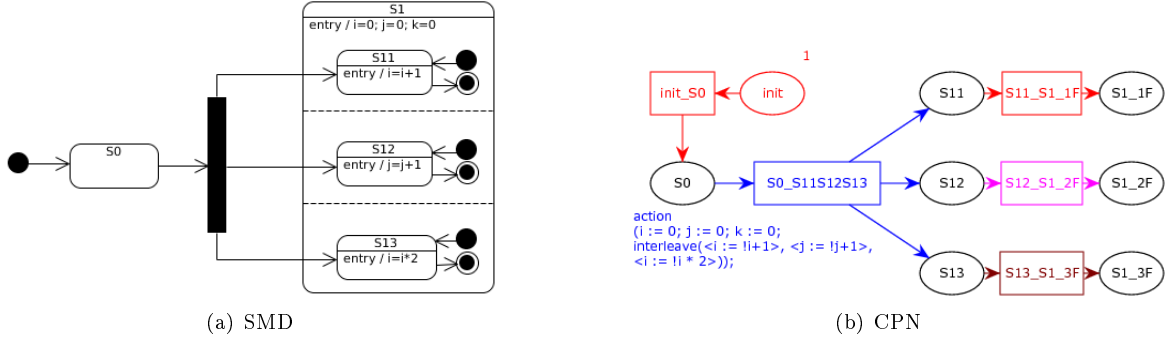


Fig. 10. Example: fork pseudostate

#### 4.5.4. Hierarchy of Orthogonal Regions

**Situation considered** We consider here a hierarchy of orthogonal regions of depth 3 in Fig. 9(a). The orthogonal state  $S1$  contains in its regions other orthogonal states ( $S11$  and  $S12$ ), themselves containing two regions each. To enter  $S1$  from  $S0$ , we need to execute at first the entry behaviour of  $S1$ , then execute in parallel the entry behaviours of  $S11$  and  $S12$  to respect the hierarchy of behaviours.

**Translation** The translated CPN is given in Fig. 9(b). Recall that, in order to give a compact CPN, we use the function *interleave()* that executes the entry or exit behaviours in an interleaving manner. Despite the depth of the hierarchy, our CPN is indeed very compact. A former version of our work [ABC14a] would have resulted in a much larger net, due to the exhaustive enumeration of all combinations of sequences of entry behaviours.

#### 4.5.5. Fork Pseudostate

**Situation considered** We consider a fork pseudostate in Fig. 10(a), that has a unique source state  $S0$ , and three target states  $S11$ ,  $S12$  and  $S13$ .

**Translation** We model this fork pseudostate with a single transition  $S0\_S11S12S13$  that has as source the place translating  $S0$ , and as target the places translating  $S11$ ,  $S12$  and  $S13$ , as shown in Fig. 10(b). The code segment of the transition  $S0\_S11S12S13$  contains the exit behaviours of  $S0$  and the entry behaviours

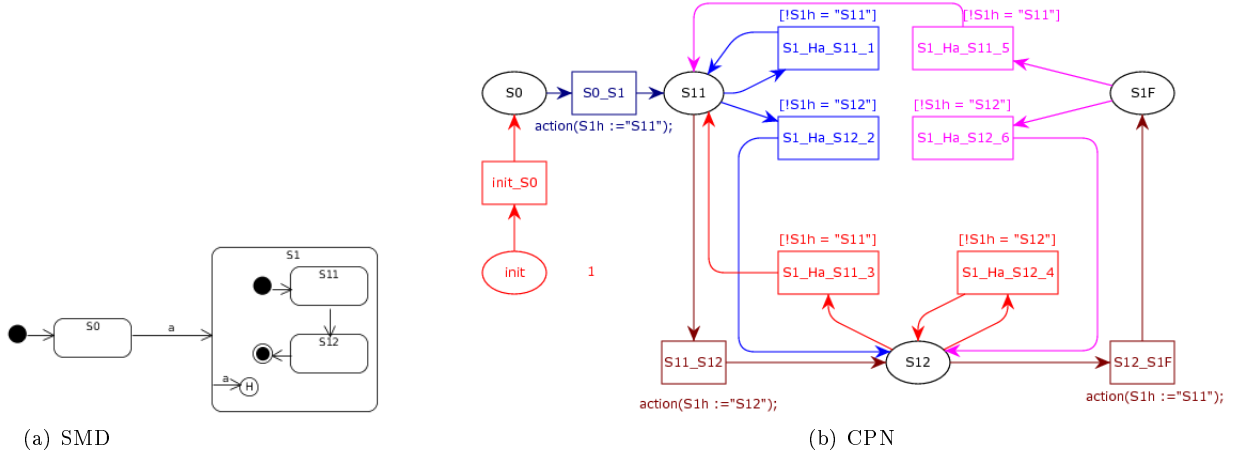


Fig. 11. Example: Shallow history pseudostate

(performed in an interleaving manner) of  $S11$ ,  $S12$ , and  $S13$ . Observe that, due to the relative order of the behaviours  $i = i + 1$  and  $i = i * 2$ , the value of  $i$  can be either 1 or 2 after executing this transition.

#### 4.5.6. Shallow History Pseudostates

**Situation considered** We consider a shallow history pseudostate in Fig. 11(a). Recall that only composite states can have a history pseudostate. The transition of the history pseudostate (*i.e.* the transition labelled with the event  $a$  in Fig. 11(a)) can be reached from any state in the region containing this pseudostate (*i.e.*  $S11$ ,  $S12$  and the final state of  $S1$ ). When this transition is taken, the state machine reaches the state of the region which was visited last.

**Translation** We associate with each history pseudostate a variable that saves the current state each time a transition is traversed inside the region that contains the history pseudostate.

In Fig. 11(b), the variable  $S1h$  is updated in any transition leading to a CPN place modelling a substate of  $S1$  (*e.g.* the code segment  $S1h := "S11"$  on transition  $S0\_S1$ ).

Then, for any direct substate of  $S1$ , two CPN transitions corresponding to the  $a$  transition are created: one leading to  $S11$  (if  $S11$  was the state visited last), and one leading to  $S12$  (if  $S12$  was the state visited last). That is, when the history pseudostate is reached by a transition, then the CPN reaches the place encoding the state that was visited last, which is retrieved thanks to the value of the variable of the history pseudostate, that is tested in a guard (*e.g.* the transition  $S1\_Ha\_S11\_1$  with guard “[ $S1h = "S11"$ ]”).

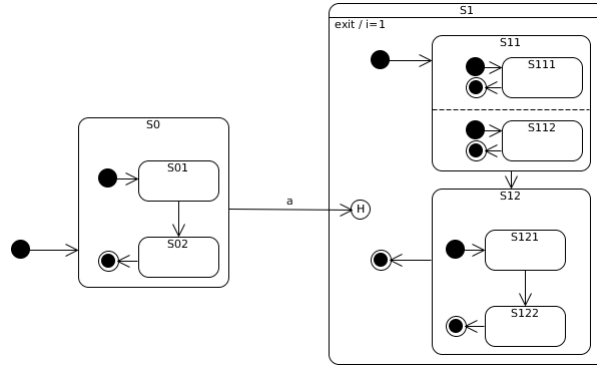
We finally give another example of a shallow history pseudostate, involving an inter-level transition, together with some orthogonal regions. The SMD is given in Fig. 12(a), and the CPN in Fig. 12(b). Observe that the variable  $S1h$  is updated to  $S11$  not only when entering the place encoding  $S11$ , but also when entering the places encoding its substates ( $S111$  and  $S112$ ).

## 4.6. Beyond our Translation

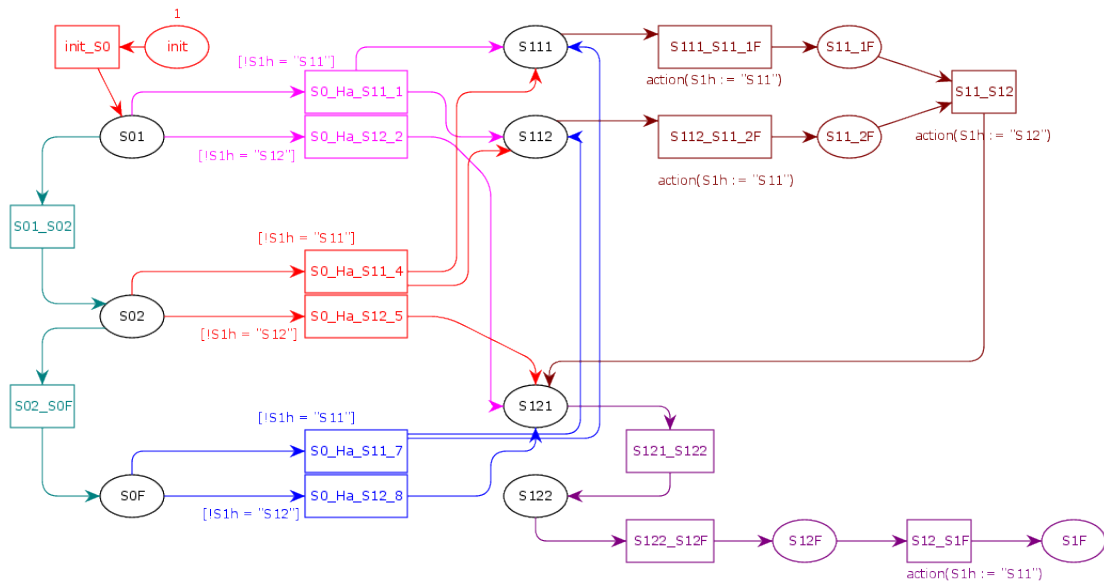
### 4.6.1. Lifting Assumptions

Most of the assumptions we made were to make our algorithm simpler. We discuss how to lift them here.

**Initial pseudostate with behaviour** Recall that a transition outgoing from a initial pseudostate may have an associated effect behaviour; to keep our algorithm simple, we left out this aspect. However, we could consider it by simply adding to the list of behaviours executed the effect behaviour of the initial pseudostate; this behaviour would be executed after the entry behaviour of the direct parent, and before that of the considered state.



(a) SMD



(b) CPN

Fig. 12. Example: Shallow history pseudostate with concurrency

**Modelling behaviour expressions** Recall that, although we do model the modification in variables in entry, exit or do behaviours, we do not keep in our translation the behaviour expressions (*e.g.* the entry behaviour expression “fade in” in state PLAYING in Fig. 3). This prevents us to verify properties such as “when a track is played (do behaviour play track), then a fade in (entry behaviour fade in) has necessarily occurred before”. To consider such a property, a global Boolean variable fade in could be added, that would be set to true when entering the place corresponding to state PLAYING, and set to false when leaving it. Hence, verifying this property reduces to verifying that the Boolean variable fade in was set to true before the CPN transition encoding the do behaviour play track fires.

**Do behaviours** An assumption we made is that only simple states can have a do behaviour. To allow composite states to have a do behaviour, it suffices to add the self-loop introduced in step 1 of Algorithm 1 to any (direct or indirect) simple substate of the composite state.

Furthermore, recall from Section 2.1.2 that we assume that a do behaviour can be executed as many times as wished. This is a stronger assumption. Indeed, CPNs are a discrete model, that cannot capture the continuous nature of some behaviours such as play track. Our translation could not be lifted easily to a continuous nature (other formalisms such as CSP would not be able either). However, we could at least

require that the do behaviour is executed at most once. This would be possible by requiring that the self-loop introduced in step 1 of [Algorithm 1](#) can only be executed once; this could be performed using an additional global variable (or a token system), that would only allow this transition to fire at most once every time the place associated with the state is entered. Continuous do behaviours could be easier to handle when extending our work to time (see [Section 7.2](#)).

**Implicit forks and joins** Implicit fork / join transitions do not enter / exit from all the regions of an orthogonal state. There is no difficulty in considering them in our work, except that step 2 of [Algorithm 1](#) should take this particular situation into consideration as follows: for an implicit join, the regions not depicted in the join can be exited from any state. For an implicit fork, the regions not depicted in the fork must be entered through their initial pseudostate.

**Fork and history pseudostates** We required that a fork does not have as a target a history pseudostate. This assumption could easily be lifted, by fusing steps 2 and 3 in [Algorithm 1](#), while considering more subcases (*e.g.*, an **if** condition that would check whether any of the target states is a history pseudostate, and then adding transitions in the line of step 3).

**Transition between pseudostates** We do not consider in our work the transitions between pseudostates (*e.g.* a transition from an initial pseudostate to a fork pseudostate) in order to keep our algorithm simple. However, we could consider this situation by simply replacing the pseudostate target of the transition by its target states.

**Transition with multiple events** In the specification, each transition can be labelled by one or more event(s); in our work, we considered only transitions with one event. We could lift this assumption by simply replacing a transition with  $n$  events into  $n$  identical transitions with one event, as a transition with multiple events can be fired as soon as one of its events matches the current triggered event.

#### 4.6.2. Extending the Syntax

**Default shallow history transitions** Default shallow history transitions could be added very easily to our translation (we left them out to reduce the number of cases in [Algorithm 1](#)). Basically, two operations should be added: first, we shall initialize the variable associated with a history pseudostate to the value of the target state of the default shallow history transition. Second, when reaching the final state of a region featuring a history state, this variable shall be set, not to the initial state, but to the target state of the default shallow history transition.

**Deep history pseudostates** Again, this could easily be added to our translation: instead of recording the value of the latest state visited, we should instead record the value of the hierarchy of states, *i.e.* not only the direct states of the composite state, but all of their (direct and indirect) substates. Hence, the type of the variable used to model history pseudostate is not anymore a state, but a list of states; furthermore, this variable should be updated not only when entering a direct substate, but also when entering indirect substates of the composite state to which the history pseudostate belongs. Finally, when a transition has as target a history pseudostate, it should target the exact hierarchy of states visited last, and not only the upper-level state.

**Internal transitions** Internal transitions can be executed provided their associated trigger is dispatched and their guard is satisfied; then, they can execute some behaviour, but do not provoke any entry/exit behaviour. Hence, they can be considered as local transitions in our translation, with the particularities that their source and target state is identical, and that no state is entered/exited when executed.

**Junctions and choices** Junction pseudostates could be added very easily: the target of a junction depends on the evaluation of the respective guards. The same mechanism could be applied in CPNs, except that we need to duplicate the CPN transition (one guarded CPN transition for each of the cases).

Modelling choice pseudostates is slightly harder: recall that choices are similar to junctions, except that the guards are evaluated *dynamically*, *i.e.* when the compound transition traversal reaches this pseudostate. That is, the evaluation may depend on the exit behaviours performed while exiting the active states towards



| Element   | Considered?                   |
|---|-------------------------------|
| Simple / composite states                               | Yes                           |
| Orthogonal regions                                      | Yes                           |
| Initial / final (pseudo)states                          | Yes                           |
| Terminate pseudostate                                   | No (but trivially extensible) |
| Shallow history states                                  | Yes                           |
| Deep history states                                     | No (but trivially extensible) |
| Submachine states                                       | No (but trivially extensible) |
| Entry / exit points                                     | No (but seems feasible)       |
| Entry / exit / do behaviours                            | Yes                           |
| variables   | Yes                           |
| External / local / high-level / inter-level transitions | Yes                           |
| Internal transitions                                    | No (but trivially extensible) |
| Basic fork / joins                                      | Yes                           |
| Implicit fork/joins                                     | No (but trivially extensible) |
| Junction pseudostate                                    | No (but probably easy)        |
| Choice pseudostate                                      | No (but seems feasible)       |
| Deferred events   | No                            |
| Timing aspects  | No                            |

Table 2. Summary of the syntactic aspects considered

this pseudostate. It is not a problem as such to translate these pseudostates to CPNs; however, if none of the guards evaluates to true, the CPN must still produce some output token, which may lead to problems in model checking. Note however that the OMG specification considers this situation as an ill-formed model. In addition, choice pseudostates will render the transition selection algorithm (Section 5) more cumbersome. This said, to evaluate whether a transition containing a choice pseudostate is enabled, it suffices to check the existence of one full path from the source state configuration to the dynamic choice Pseudostate in which all guard conditions are satisfied; hence, if at least one of the guards of the choice pseudostate (dynamically) evaluates to true, this does not pose a particular problem (and otherwise, once more, the model would be ill-formed).

**Entry points, exit points, terminate pseudostates, submachine states** Entry and exit points do not pose any particular problem.

Terminate pseudostates (implying that the execution of the entirely SMD is immediately aborted) can be simply modelled using a global Boolean variable, that is set to false if the terminate pseudostate is reached. Then, any CPN transition is guarded by a check that this variable is true. This would be similar to the translation of “activity final” nodes of UML activity diagrams into coloured Petri nets in [ACR13].

A submachine state is “semantically equivalent to a composite state” [OMG15, p.309], and hence is in fact already taken into account by our translation.

**Time, deferred events** These two points are not considered in our work, and it is unclear whether our translation scheme could easily be extended to these aspects; hence, integrating these aspects are among our main future works (see Section 7.2).

## 5. Transition Selection

Until now, we presented a standalone theory to formalise UML state machines using coloured Petri nets. This theory extends previous works (*e.g.* [ABC14a]) by considering most syntactic aspects of UML state

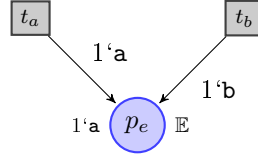


Fig. 13. Modelling the events place and the random event arrival

machines (or discusses how to consider them in a straightforward manner). However, our scheme does not take into account the priority mechanism of UML.

In this section, we extend our work to take both the event dispatching and the priority management of UML into consideration.

Beyond the historical reasons (the original version of this work [ABC14a] did not take priorities into consideration), there are several reasons to separate the handling of priorities from the rest of the translation. A first advantage of considering such a two-step strategy is that our translation (not handling priorities) formalised as in Algorithm 1 is a standalone work, that takes many syntactic aspects of the UML into account, with of course the exception of priorities; as many (if not most) translation works from the literature omit priorities, Algorithm 1 is a good mean to compare our work with previous works. Second, the translation output by Algorithm 1 is elegant, and “graphically similar” to the original UML state machine; this will not be the case anymore when considering priorities. Third, we believe it is interesting to separate the handling of priorities from the rest of the translation, as this is a specific aspect of UML state machines that is formalised in a dedicated part of the OMG specification (*i.e.* [OMG15, p.315]).

## 5.1. Event Arrival and Dispatching

We reuse the idea proposed in [CKZ11] to represent events in a given CPN place. First, we assume in the CPN an enumerated type “event” (denoted by  $\mathbb{E}$ ), the values of which can be the various events appearing in the source SMD (*e.g.*  $\mathbf{a}$ ,  $\mathbf{b}$ ,  $\mathbf{c}$ ), together with a special value  $\bullet$ , that denotes “no event”. Then, we add one extra place to the CPN, of type “event”. This place represents the events awaiting processing, *i.e.* it contains a set of events (*i.e.* of CPN tokens the value of which is an event) to be dispatched; hence, it acts as the events pool of the OMG specification.

We wish to impose neither the way that events appear, nor the way that events are dispatched (“The order of event dispatching is left undefined, allowing for varied scheduling algorithms”, [OMG15, Section 14.2.3.9.1, p.314]). However, we believe that, if needed, both the event arriving scheme and the event dispatching scheme can be easily encoded using additional CPN fragments to be connected to our translation. For example, Fig. 13 depicts a situation where any event can occur anytime: the place  $p_e$  (of type  $\mathbb{E}$ ) is the events place, that contains (here) one event  $\mathbf{a}$  (“ $1\mathbf{a}$ ” is the CPN notation for one token of value  $\mathbf{a}$ ). The CPN transition  $t_a$  (resp.  $t_b$ ) models the arrival, in any order, at any time, of an event  $\mathbf{a}$  (resp.  $\mathbf{b}$ ). More complex arrival schemes can easily be modelled by the user whenever needed.

A small contrast with the UML specification spirit is that completion events are not represented as tokens, *i.e.* they do not appear in the waiting events place (*i.e.* the events pool). In fact, they will be processed directly when handling priorities: that is, any transition triggered by an event will be blocked whenever a completion transition is enabled.

## 5.2. Handling Priorities

### 5.2.1. Overview

In order to handle priorities, we will make some minor changes to the translation scheme introduced in Section 4, as well as a few additions. The global spirit of our solution is as follows: First, the events arriving (*i.e.* waiting to be dispatched) are now encoded into CPN tokens. Second, for each possible event (say  $\mathbf{a}$ ), a dedicated (unique) CPN transition will appear in the resulting CPN; this will be achieved by simply fusing all CPN transitions (created from Section 4) handling an occurrence of this event  $\mathbf{a}$ . Priorities will be handled using a (static) CPNML code segment associated with that (unique) CPN transition.

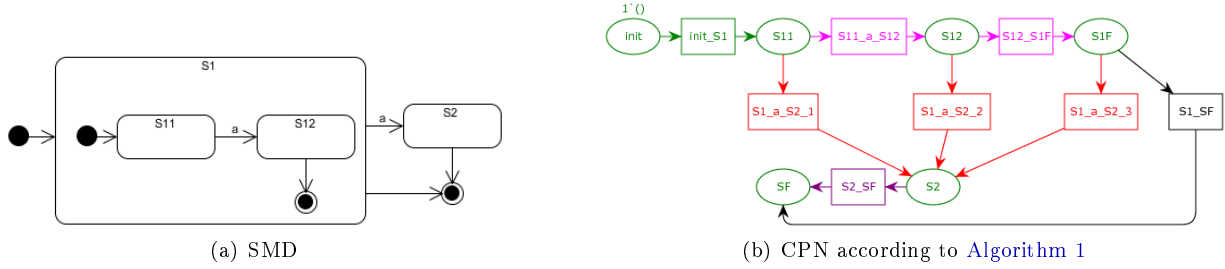
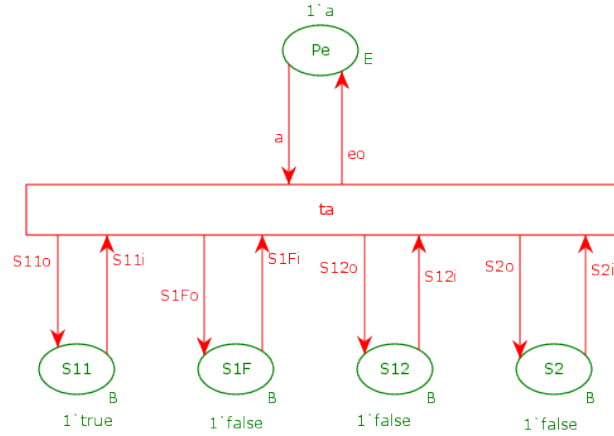


Fig. 14. Example with priorities


 Fig. 15. An example of a fused transition for event  $a$ 

In order to know whether some transitions can fire (which is needed to handle priorities), we will need to test for the presence (or not) of tokens in some places of the CPN. A solution could have been read-arcs. However, to make our scheme simpler, we propose the following: all places created in Section 4 (that used to possibly contain a token of type  $\bullet$ ) now systematically contain a unique token, of type  $\mathbb{B}$ . Basically, when the UML simple state encoded by this place is active, then the place contains *true*, and otherwise *false*.

In the following, we describe our scheme step by step.

### 5.2.2. Fused Transitions

Any CPN transition in our scheme of Section 4 encoding the same event (say  $a$ ) is now fused: that is, this gives birth to a super CPN transition with many incoming places and many outgoing places. We also delete redundant arcs, *i.e.* so that at most one arc goes from a given place to a given transition, and conversely. The behaviours and guards (that were attached to CPN transitions as CPN ML code segment in Section 4) are removed for now. Instead, we will attach a new CPN ML code segment to each of these fused CPN transitions in order to implement the priority mechanism, and determine which UML transition will indeed be processed (note that there can be none, one or more); the behaviours of these transitions will also be executed within that CPN ML code segment.

In addition, we also add to this transition incoming and outgoing arcs from and to any place that is also the source of a CPN transition encoding a completion event (in the entire CPN, not only in the current region). This allows for the attached CPN ML code segment to test whether some completion events are enabled, in which case no transition labelled with an event (*e.g.*  $a$ ) should fire.

In order to test for the presence of tokens (or, in fact, whether their value is *true*) in the CPN ML code, we need to name the variables on the incoming and outgoing arcs: for each place  $p$ , we denote by  $pi$  the value coming from this place to the transition, and by  $po$  the value from the transition back to the place. An exception is for the event place: for a transition handling event  $a$ , the only value accepted as input is  $a$ ;

as for the output, we denote it by  $eo$ , as either the token will be sent back (in case a completion transition is enabled) with the same value  $a$ , or the event will be consumed and the value sent back is  $\bullet$ .

**Example 1.** Consider the SMD given in Fig. 14(a) and its corresponding translation according to Algorithm 1 given in Fig. 14(b). Each simple state is represented with a corresponding place (e.g. state S11 is represented with place S11). Each transition originating from a simple state (with and without event) is represented with a corresponding CPN transition (e.g. the transition between S11 and S12 is represented with the CPN transition S11\_a\_S12). Each transition with an event originating from a composite state is represented with a set of CPN transitions encoding the combination (e.g. the transition labelled with event  $a$  from S1 to S2 is represented with the CPN transitions S1\_a\_S2\_1, S1\_a\_S2\_2 and S1\_a\_S2\_3). Note that transitions from a composite state without event have the same mechanism as transitions from simple states.

The CPN model resulting from the application of Algorithm 1 does not consider the firing priority of transitions. For example, the transition from S1 to the root final state has (considering the OMG transition selection algorithm) higher priority than the transition from S1 to S2. In order to take into account this kind of situation (and the OMG transition selection algorithm), we propose a mechanism of transition fusion that handles the firing priorities. For example, in Fig. 14(a) there is only one event  $a$  that can be in conflict of priority with the completion event. In Fig. 15, we give the fused transition for event  $a$ , this transition has as pre/post places the events place  $p_e$  (the event pool place), as well as all places source or target of a transition labelled with  $a$  in Fig. 14(b). The fused transition  $ta$  will replace all transitions with event  $a$ . The source/target states of this transition  $ta$  are all the states that are source/target of the previous transitions with event  $a$ . Note that the states that are source or target of a completion transition (in conflict of priority with the transition labelled with  $a$ ) will be also linked to  $ta$ . For example, states S1F and S2 are not only connected to  $t_a$  because they are connected to a transition labelled with  $a$  in Fig. 14(b), but also because they are source of a completion transition (which has higher priority than the transition labelled with  $a$ ).

### 5.2.3. Algorithm for Handling Priorities

We describe in Algorithm 2 the procedure handling the selection algorithm and the firing priorities of transitions explained above. We reuse the places and transitions resulting from the application of Algorithm 1.

We assume in Algorithm 2 a function `AddCodeSegments()` that updates the code segment of each transition in the CPN; this code handles priorities and conflicts, and is described in Sections 5.2.4 and 5.2.5.

As an example, Fig. 17 presents the application of Algorithm 2 to the CPN model in Fig. 16 (resulting itself from the application of Algorithm 1). For sake of readability, code segments are omitted.

### 5.2.4. Detecting and Handling Completion Events

Completion events should be processed ahead of any dispatched event. This is achieved as follows. On the one hand, the CPN transitions handling completion events are left untouched, with the exception that the *true/false* token system should be kept consistent: that is, the CPN transitions handling completion events shall now take both as pre and as post all tokens of the source and target places, ensure that all source tokens are *true*, and replace them with *false* whereas the target tokens are replaced with *true*. For example, the completion transition from S2 to SF in Fig. 14(a) is translated into the CPN fragment given in Fig. 18. By keeping the completion transitions (almost) unchanged, we preserve the fact that any such transition can fire any time whenever enabled (*i.e.* whenever the source states are active).

On the other hand, we must prevent transitions encoding events to occur whenever any completion transition is enabled. Knowing whether a completion transition is enabled is easy: it suffices to find a set of CPN places source of a CPN transition encoding a completion event that all contain a *true* token, and the guard of which is satisfied. The function checking this can be computed statically once for all (although its execution will of course be dynamic). In fact, it can be computed once and for all, for all fused transitions: hence, this can be factored using a single Boolean function.

This function uses the value of the tokens (*true* or *false*) and checks the guards. For example, the function checking whether a completion event is enabled is given below for the SMD in Fig. 14(a) as follows. (Note that there is no guard in this example.)

```

1 function completionEventEnabled ()
2   S12i or S1Fi or S2i or SFi

```

**Algorithm 2:** Handling firing priorities

---

```

1 Add  $\textcircled{\text{Pe}}$  // Create the events pool place
2 foreach event  $e \in \mathcal{E}$  do
  // Step 1
  // Add the event to the events pool
3 Add an "e" token to  $\textcircled{\text{Pe}}$ 
  // Add the transition corresponding to the fusion, and link  $\text{Pe}$  and  $t_e$ 
4 Add  $\textcircled{\text{Pe}} \xrightarrow{ei} t_e \xrightarrow{eo} \textcircled{\text{Pe}}$ 
  // Step 2
5 foreach transition  $t = (\mathbf{S}_1, e_1, g, (b, f), sLevel, \mathbf{S}_2) \in \mathcal{T}$  such that  $e_1 = e$  do
6   Delete  $t$ 
7   foreach simple state  $s_1 \in \mathbf{S}_1$  do
8     Add  $s_1 \xrightarrow{s_1^i} t_e \xrightarrow{s_1^o}$ 
9   foreach simple state  $s_2 \in \mathbf{S}_2$  do
10    Add  $s_2 \xrightarrow{s_2^i} t_e \xrightarrow{s_2^o}$ 
11   foreach completion transition  $(\mathbf{S}'_1, noEvent, g', (b', f'), sLevel', \mathbf{S}'_2) \in \mathcal{T}$  do
12     foreach simple state  $s \in \mathbf{S}'_1$  do
13       Add  $s \xrightarrow{s^i} t_e \xrightarrow{s^o}$ 
14 AddCodeSegments();

```

---

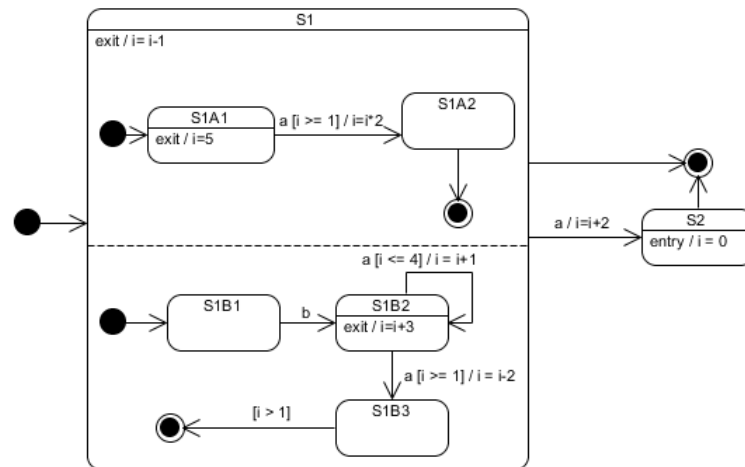


Fig. 16. Example with priorities and concurrency

This function may involve more than just disjunctions when completion transitions are guarded, or in case of concurrency: for an orthogonal composite state to launch a completion event, all regions must be in their final state. For example, the function checking whether a completion event is enabled is given below for the SMD in Fig. 16:

```

1 function completionEventEnabled ()
2   S1A2i or (S1B3i and i > 1) or (S1AFi and S1BFi) or S2i or SFi

```

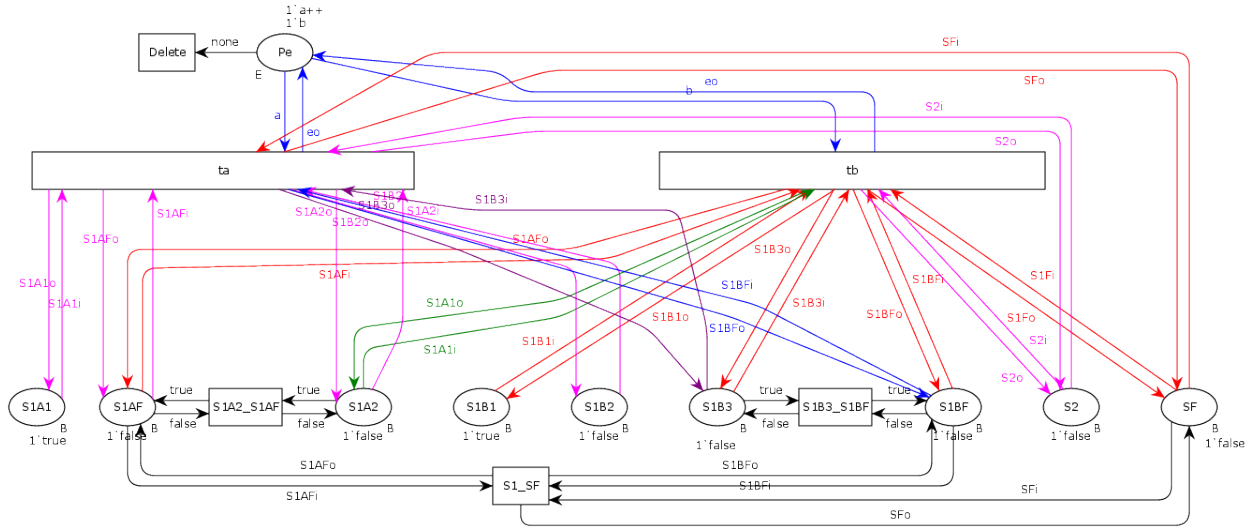


Fig. 17. Example with priorities and concurrency: translation

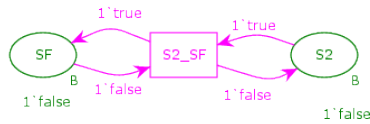


Fig. 18. Modelling a completion transition

5.2.5. Detecting and Handling Conflicts

Let us come to the main part of this section: determining which UML transitions should be executed. Whenever an event *a* occurs (*i.e.* a token of value *a* arrives to the events place), it may be processed. The transitions (none, one or more) triggered by this event must conform to the UML priority mechanism. This can be implemented using a greedy algorithm described in the OMG specification as follows. “States in the active state configuration are traversed starting with the innermost nested simple States and working

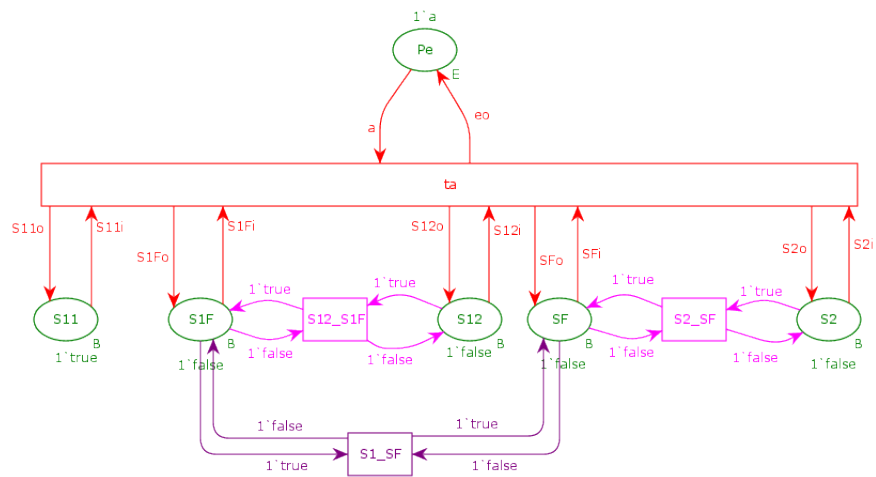


Fig. 19. Example with priorities: complete translation

outwards. For each State at a given level, all originating Transitions are evaluated to determine if they are enabled.” [OMG15, Section 14.2.3.9.5, p.316] Whereas this algorithm must be applied dynamically, the CPN ML code implementing this algorithm can be statically computed once for all, for each possible event. That is, given an event **a**, we statically compute all possible transitions that may be triggered by **a**, and check their source states and associated guards using the greedy algorithm described above.

Basically, one or more CPN transitions encoding UML transitions labelled with an event **a** can fire if 1) an event **a** can be dispatched (*i.e.* it arrived in our events place, and was not yet processed), and 2) no completion transition is enabled. This is achieved by checking that an event **a** is indeed present in the events place, and by checking that no completion event is enabled, which is achieved thanks to the CPN ML function defined above.

Next, it should be determined which transitions should fire. We achieve this in the CPN ML code associated with the fused transition as follows:

1. If a completion transition is enabled, then no transition labelled with **a** fires, and the event token is put back to the events place (waiting until no completion event is enabled).
2. Otherwise, we create a set of Boolean flags (all initially true), one per region in which a direct substate is the source of a transition labelled with **a**. These flags indicate whether the transitions emanating from a direct substate in this region should still be explored.
3. We also copy all variables of the SMD (encoded in CPN global variables); indeed, we will apply some behaviours (that may modify these variables) before testing the guards of other transitions.
4. Then, starting from the innermost nested simple states that are source of a transition labelled with **a**, we check whether all places corresponding to source states of that transition (that may contain several source states, if a join, or if a compound transition) contain a token; and if so, whether the transition guard (checked using the copied variables) is true; if so, we disable all flags of regions containing that state (to prevent higher-level transitions from firing). Note that the transitions in other regions (*i.e.* not in conflict with this transition) are not disabled that way.
5. As long as no enabled transition is found, we traverse the state hierarchy towards the top.

**Example 2.** Let us first exemplify this approach on the non-concurrent example in Fig. 14(a): the CPN ML code attached to the fused transition **a** is given below.

```

1  if completionEventEnabled() then
2      (* No firing of "a" for now: return all tokens as they are, including the event itself *)
3      S11i, S12i, S1Fi, S2i, SFi, "a"
4
5  else
6      (* Prepare the resulting values, initially equal to the input values *)
7      let S11o, S12o, S1Fo, S2o, SFo =
8          S11i, S12i, S1Fi, S2i, SFi
9      in
10
11     (* Define flags for all regions *)
12     let flagS1, flagR = true, true in
13
14     (* Copy variables *)
15     (* no variable to copy in this SMD *)
16
17     if flagS1 then
18         if S11i (* and no guard here *) then
19             S11o := false (* disable input places *)
20             S12o := true (* enable output places *)
21             flagR := false (* block parents regions *)
22             (* no behaviour to execute here *)
23
24     if flagR then
25         if S11i (* and no guard here *) then
26             S11o := false (* disable input places *)
27             S2o := true (* enable output places *)
28             (* no parents regions to block *)
29             (* no behaviour to execute here *)
30         else if S12i (* and no guard here *) then
31             S12o := false (* disable input places *)
32             S2o := true (* enable output places *)
33             (* no parents regions to block *)
34             (* no behaviour to execute here *)
35         else if S1Fi (* and no guard here *) then

```

```

36     S1Fo := false (* disable input places *)
37     S2o := true (* enable output places *)
38     (* no parents regions to block *)
39     (* no behaviour to execute here *)
40
41     (* Return all tokens values *)
42     S11o, S12o, S1Fo, S2o, SFo, "none"

```

The first part of the above CPN ML code checks whether some completion event is enabled somewhere in the SMD; if so, all tokens are put back to their place unchanged, including the event. Otherwise, we define the output values (that label the arcs leaving the fused transition), initially equal to the input.<sup>4</sup> Then, Boolean flags are defined for each region not reduced to a simple state (*i.e.* S1, as well as the top-level region, the flag of which is denoted “flagR”). Then, for each such region, we check whether an **a** transition can fire: that is, we successively check that the region is not yet disabled (“if flagS1 then”), that a token is present in the input place (“if S11”) and that the transition guard is true (no guard for this example). If so, we disable the tokens in the input places, enable the tokens in the output places, block the upper regions (“flagR := false”), and execute the behaviours, by possibly reusing the *AddBehaviours()* function (no behaviour in this simple example). Note that the various combinations for transitions exiting composite states (three in the case of the **a** leaving S1) were already computed by Algorithm 1 and can be just enumerated here. Eventually, all output values are returned, and the event is consumed, hence replaced with a • token (denoted by “none” in the CPN ML code).

**Example 3.** Let us now consider the more complex example in Fig. 16, involving behaviours, guards and concurrency. We give the CPN ML code below (note that not all combinations in the regions of S1 are given, for sake of conciseness).

```

1  if completionEventEnabled() then
2    (* No firing of "a" for now: return all tokens as they are, including the event itself *)
3    S1A1i, S1A2i, S1AFi, S1B1i, S1B2i, S1B3i, S1BFi, S2i, SFi, "a"
4
5  else
6    (* Prepare the resulting values, initially equal to the input values *)
7    let S1A1o, S1A2o, S1AFo, S1B1o, S1B2o, S1B3o, S1BFo, S2o, SFo =
8      S1A1i, S1A2i, S1AFi, S1B1i, S1B2i, S1B3i, S1BFi, S2i, SFi
9    in
10
11    (* Define flags for all regions *)
12    let flagS1A, flagS1B, flagR = true, true, true in
13
14    (* Copy variables *)
15    let i' = i in
16
17    if flagS1A then
18      if S1A1i and i'>=1 then
19        S1A1o := false (* input places *)
20        S1A2o := true (* output places *)
21        flagR := false (* block parents regions *)
22        i:=5 ; i:=i*2 (* transition behaviours *)
23
24    if flagS1B then
25      if S1B2i and i'<=4 then
26        S1B2o := false (* input places *)
27        S1B2o := true (* output places *)
28        flagR := false (* block parents regions *)
29        i:=i+3 ; i:=i+1 (* transition behaviours *)
30      else if S1B2i and i'>=1 then
31        S1B2o := false (* input places *)
32        S1B3o := true (* output places *)
33        flagR := false (* block parents regions *)
34        i:=i+3 ; i:=i-2 (* transition behaviours *)
35
36    if flagR then
37      if S1A1i and S1B1i (* and no guard here *) then
38        S1A1o := false ; S1B1o := false (* input places *)
39        S2o := true (* output places *)
40        (* no parents regions to block *)
41        i:=5 ; i:=i-1 ; i:=i+2 ; i:=0 (* transition behaviours *)

```

<sup>4</sup> The “let...in” syntax is CPN ML to define variables.



```

42  else if S1A1i and S1B2i (* and no guard here *) then
43      S1A1o := false; S1B2o := false (* input places *)
44      S2o := true (* output places *)
45      (* no parents regions to block *)
46      interleave(<i:=5>, <i:=i+3>) ; i:=i-1; i:=i+2 ; i:=0 (* transition behaviours *)
47  else if S1A1i and S1B3i (* and no guard here *) then
48      ... etc ...
49
50  (* Return all tokens values *)
51  S1A1o, S1A2o, S1AFo, S1B1o, S1B2o, S1B3o, S1BFo, S2o, SFo, "none"

```

The addition of concurrency has the following consequences on this example. First, if a transition can fire in the upper region of **S1**, then we do not block the lower region; only the parent region (the root region) is blocked to prevent the transition from **S1** to **S2** to be executed. Hence, one transition in the upper region of **S1** and/or one transition on the lower region of **S1** can be executed. Second, in an orthogonal region, we need to enumerate all combinations of source states; this CPN ML code can be automatically generated thanks to the *combinations* function of Section 4. Third, in the case of an exit of **S1** when in orthogonal regions, the behaviours are interleaved (*e.g.* when in **S1A1** and **S1B2**) using the *interleave* function defined in Section 4.

### 5.2.6. Discussion

We discuss several aspects of our solution in the following. First, this scheme respects the UML priority scheme, and is in line with the fact that a dispatched event that meets no transition is discarded (“If no Transition is enabled and the corresponding Event type is not in any of the deferrableTriggers lists of the active state configuration, the dispatched Event occurrence is discarded and the run-to-completion step is completed trivially” [OMG15, Section 14.2.3.9.1, p.314]). Indeed, when the CPN ML code associated with transition **a** is executed, if no completion transition is enabled, but also no transition fires (because either the tokens are not present, or the guard is not satisfied), then the event token is still discarded. In contrast, if a completion transition was enabled, in that case, the event is not discarded: it is just put back to the events place, hence postponed until no completion transition is enabled.

Second, our CPN ML code segments are relatively complex. However, recall that building this code (which requires traversal of the hierarchical structure of states and transitions) is performed only once for all (for each event) during the translation phase, and not during the model checking phase. During the model checking phase, only the *execution* of this code (which mainly consists in checking Booleans and executing the behaviours) is performed.

Third, although we do conform to the conflict resolution mentioned by the UML, our work has one limitation, that occurs in the case of two transitions for which the conflict resolution mechanism does not solve the conflict (*i.e.* two transitions outgoing from the same state, with both guards satisfied). This situation is ambiguous in the OMG specification (“If that event occurs and both guard conditions are true, then at most one of those Transitions can fire in a given run-to-completion step.” [OMG15, Section 14.2.3.9.3, p.315]). For example when in **S1B2** in Fig. 16, if  $1 \leq i \leq 4$  and event **a** is available, both transitions to **S1B2** and **S1B3** can be executed. In our approach, we *statically* solve this situation: only one transition fires, and always the same; in fact this is the first one appearing in the CPN ML code segment, which depends on the traversal order of the state hierarchy during our translation. In the case of the example of **S1B2** in Fig. 16, the first transition in the code is the self-loop to **S1B2**. In order to make this choice more flexible, and although choosing one transition statically still remains in line with the specification, we propose directions for future works: (i) forbid non-mutually-exclusive guards for transitions outgoing from the same state with the same trigger (ii) require a priority mechanism in the UML diagram, that our algorithm would then follow (iii) allow for a random choice of the transition to fire.

As a minor remark, note that, when an event is consumed, it is actually not strictly speaking consumed, but sent back to the events place with a  $\bullet$  value; hence, it will remain there forever. This is consistent with the UML semantics, but will lead to an accumulation of useless tokens in the event place, that might slow down model checking. A possibility to avoid this is to delete these tokens thanks to an additional CPN transition that has the events place as source, no target place, and accepts only  $\bullet$ -valued tokens.

Finally, recall that deferred events are not considered in our work. Considering them (which would certainly not be simple though) could be achieved thanks to our events place: when deferred, an event would be moved to another place (that would contain all deferred events); then whenever a transition is ready to accept a deferred event, that event would have higher priority than regular events.

## 6. Application to the CD Player

### 6.1. Application of the Translation

We applied our translation scheme to the CD player presented in [Section 3](#). The translation was performed manually, by following [Algorithm 1](#) and then by fusing transitions according to [Section 5](#). (An implementation to automate this process is the subject of ongoing work, see [Section 7](#).)

We handle the event arriving scheme in a slightly different manner from [Section 5.1](#): since we wish to model that any event can arrive anytime, and any event can be dispatched anytime, we always keep one event of each possible value in our events place. To do so, we initially add one event of each value to the events place (*viz.* `load`, `off`, `pause`, `play`, `stop`); and then, instead of consuming events when dispatching an event (by returning a  $\bullet$ -valued token back to the events place), we always return the event, be it handled or not. This ensures that any event is always available, while keeping the state space finite (having an unbounded number of waiting events and/or of  $\bullet$ -valued tokens in the events place might lead to an infinite state space if no reduction technique is used by the model checker).

### 6.2. The Limits of Discrete Formalisms for Continuous Behaviours

A quick simulation using CPN Tools made us realize that the translated player does not behave the way we intended. In particular, when in `BUSY`, the transitions labelled with `pause/off/load/stop` can never be executed. The explanation is as follows: recall that we assume that “do” behaviours can be executed zero, one or many times. As a consequence, the completion transition of `PLAYING` can occur anytime; since completion transitions have a higher priority than transitions with events, the transition (via `pause`) to `PAUSED` is always hindered by the enabled completion transition.<sup>5</sup> The same occurs in `LIGHTON`.

This is not only a limit of our translation, but of any formalization using *discrete* formalisms (be automata, Petri nets, CSP, SPIN, etc.). In fact, without a *continuous* formalism (that would, *e.g.* be able to express continuous time), it is not possible to properly model that “playing a track” is of continuous nature and, once it has finished, then the enclosing region can be completed.

Recall that this case study is a slightly modified version from [\[ZL10\]](#); in that previous work, this problem was not detected because priorities between transitions do not seem to have been properly encoded.

### 6.3. Modifying the Model

In our case study (presented in [Section 3](#) and [Fig. 3](#)), we suggest a discretisation of the continuous nature of the track playing. First, we remove the “do” behaviour of `PLAYING`. Second, we encode the track playing by elapsing some (discrete) time. That is, we assume that a track has a duration `minuteCount` (all tracks have the same duration to keep the model simple). Then, we create a variable `minute`, initially set to 0 when starting a new track. Then, time elapsing is modelled by incrementing `minute` (as long as `minute < minuteCount`) when in state `PLAYING` with an event `timeElapse`. Finally, both regions of `BUSY` can only reach their final state whenever the track has completed, *i.e.* `minute = minuteCount`: that way, these completion transitions only prevent the `pause` event to be processed once the track reached its end. Note that extending our work to *dense time* models (see [Section 7.2](#)) will help to consider case studies with a such a continuous nature in “do” behaviours without the need to perform a discretisation.

The modified UML model is given in [Fig. 20](#).

<sup>5</sup> Technically, due to the “do” behaviour in `PLAYING`, there should exist runs in which the “do” behaviour is executed, and hence the completion transition does not prevent other transitions to occur. However, this does not happen in our translation, as the fact that the completion transition *may* occur already prevents other transitions to occur. In order to avoid this issue, a possibility would be to modify our encoding so as to allow in any state both the completion transition, and any regular transition (labelled with an event) to occur. However, to be consistent with the semantics, if a regular transition occurs, then a completion transition can only occur after one more execution of the “do” behaviour. This could easily be achieved using a global Boolean variable (for each “do” transition) in the resulting CPN, blocking the completion transition if a regular transition occurred from this state until another execution of the “do” transition occurs.

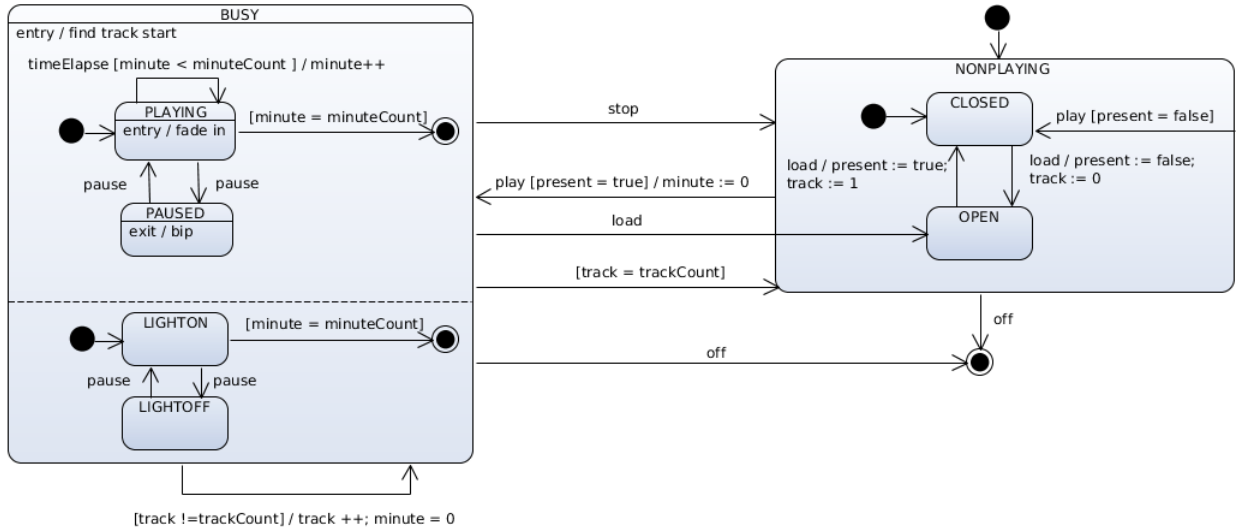


Fig. 20. The CD-Player example (revised version)

#### 6.4. Verifying Properties

We generate the CPN corresponding to the modified SMD model. We give the resulting CPN in Fig. 21; for sake of readability, the CPNML code segments are left out on the graphics. The top-most place is the events place, initially containing all events. All other places represent simple states (including final states) of the SMD. The fused transitions can be recognized by their numerous incoming and outgoing arcs. Our complete CPNTools model including all code segments, and together with some commands to perform the verification of the properties (see below), is available online.<sup>6</sup>

We use CPN Tools 4.0 [Wes13] to verify properties on our modified CD player.

CPN Tools is not an on-the-fly model-checker, and requires the generation of the entire state space before verifying properties. The first step of the verification phase is to specify the property in the CPNML language (the language used in CPN Tools both in the code segments and to specify properties). Second, we generate the state space of the net using the state space generator of CPN Tools. Third, we ask CPN Tools to evaluate the property on the generated state space.

We fix the value of `trackCount` to 5 and the value of `minuteCount` to 3. The state space generated by CPN Tools is made of 159 symbolic states and 1551 edges (computed in 4.0s on an Intel core i5-4570 CPU 3.2GHz with 8 GiB RAM). After generating the state space, we can now formally verify the properties formulated in Section 3.

**Property 1 (“the CD player cannot be both closed and open”)** It suffices to verify that, in any marking of the state space, we cannot have a token in `closed` and `open` at the same time. This property is proven valid by CPN Tools almost instantaneously, as the state space was generated beforehand and is of a reasonable size (the same applies to the other properties).

**Property 2 (“whenever the CD player is in state PLAYING, there is a CD in the player”)** Checking this property reduces to checking that, whenever there is a token in `playing`, then the value of the global variable `present` is `true`. This property is proven valid by CPN Tools.

**Property 3 (“whenever the player is paused, the light is off”)** Checking this property reduces to checking that, whenever there is a token in `paused`, then there is also a token in place `lightoff`. This property is proven valid by CPN Tools.

<sup>6</sup> <http://www.lipn.fr/~benmoussa/SMD2CPN/>

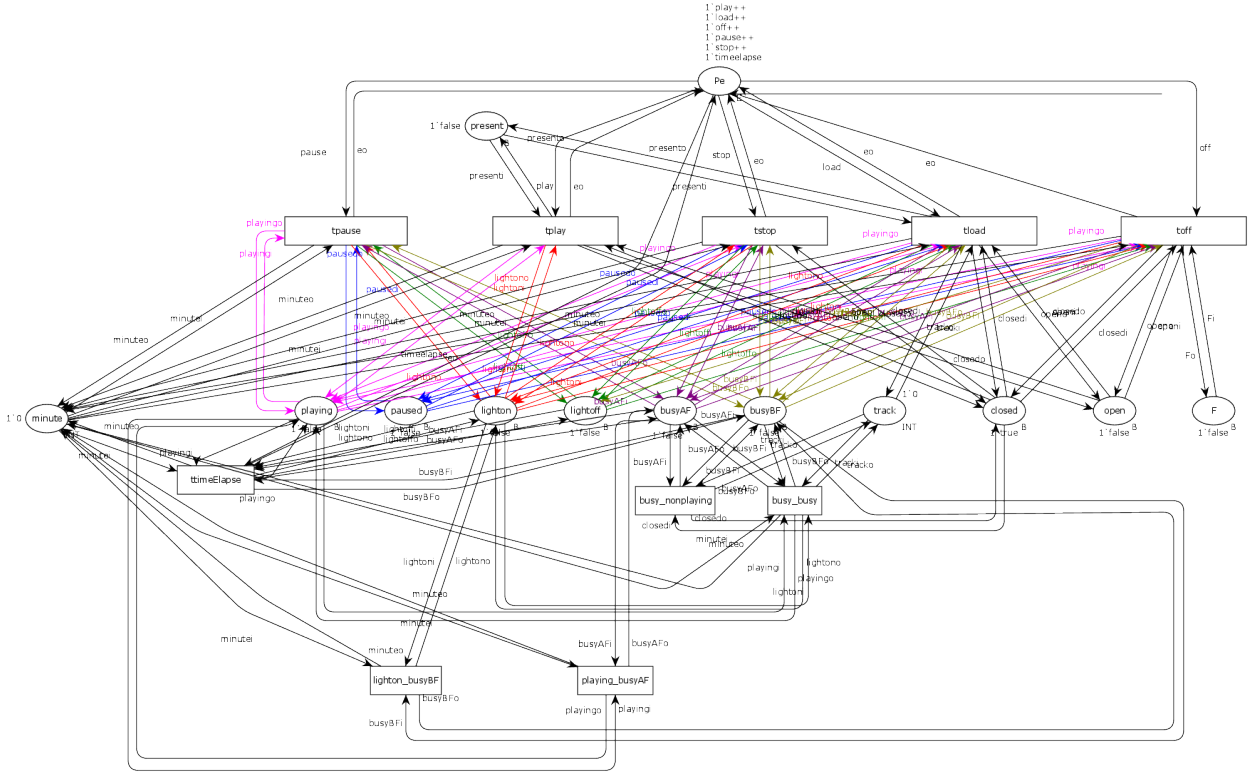


Fig. 21. Translation of the revised CD player into a CPN

**Property 4 (“the value of track never exceeds trackCount”)** Checking this property reduces to checking that, in any state of the state space, we have that  $\text{track} \leq \text{trackCount}$ . Again, this property is proven valid by CPN Tools.

**Remark** Our verification has been performed for a fixed value of `trackCount` and `minuteCount`. Performing the verification for *any* value of `trackCount` and `minuteCount` would require parametrised verification techniques (which goes beyond the scope of this work).

## 6.5. Discussion

We believe that 4.0 seconds to build a state space for this (relatively) simple example is not very impressive. However, we would like to make the following remarks. First, scaling might not entirely be an issue, in the sense that most of the UML state machine diagrams we found in the literature (including industrial case studies reported in published papers) are usually relatively small in terms of number of states, transitions or variables. Being small does not necessarily mean simple; even in the case of our (simple) CD player, it is not clear at all to be convinced without formal automated methods that our properties hold.

Second, we chose CPN Tools as it is the most famous state-of-the-art model checker for CPNs, and hence its use is natural in our context to serve as a proof of concept. However, CPN Tools is notoriously slow (either due to the intrinsic complexity of CPNs, or due to internal implementation issues). In fact, as noted in Section 4.1, our CPNs are not exactly coloured, in the sense that all tokens are of type  $\bullet$  (or of type  $\mathbb{B}$ , which can be easily mapped to untyped tokens). Colours appear in guards and transition updates, where global variables are updated. Hence a solution that we would like to investigate in the future is to use a model checker that supports Petri nets extended with global variables, and time as well, as future works include the extension to time (see Section 7.2). Model checkers for time Petri nets include Tina [BV06] (that does not

support such global variables) and Roméo [LRST09] (that does support such global variables that can be used in functions). As a consequence, investigating an alternative destination model checker (together with the introduction of time) is in our agenda. Note that changing the destination model checker would have no consequence on our translation, and almost no consequence on our ongoing implementation (only the “printer”, mapping the internal representation of our translation tool to the input syntax of the destination model checker, needs to be changed).

## 7. Conclusion and Future Works

We presented here a formalisation of UML concurrent state machine diagrams by translating them into coloured Petri nets. We take into account a set of syntactic elements in our translation: simple, composite and orthogonal states, most kinds of transitions (local, external, inter-level), most important pseudostates (fork, join, shallow history, initial), concurrency, behaviours (entry, exit and do), variables, hierarchy of states and behaviours. We also added an extension to encode the transition selection mechanism of UML.

We entirely revised the translation mechanism of [ABC14a], leading to what we think is a significantly clearer solution. This also leads to a simpler translation algorithm. In particular, in [ABC14a] we used a fixed structure in CPNs to represent entry and exit behaviours of SMDs, and this yielded an unwanted complexity. In addition, in order to clarify our approach, we presented here in a systematic way the different cases to be taken into account in order to translate a transition (triggered or not by an event) between two states, that may be simple, composite, orthogonal etc., with/without entry/do/exit behaviours. We released the constraint we had in [ABC14a] that was requesting all states to have entry and exit behaviours. We released another constraint that was requesting all composite states (and state machines) to have a final state. In addition to the syntax taken into account in [ABC14a], we added history pseudostates and improved the different cases of fork/join pseudostates taken into account. We also adapted the translation to handle run-to-completion steps and transition selection.

Our translation was applied to various examples, including a CD player, on which we could successfully verify several properties using CPN Tools. We presented in a detailed way our case study, and showed how the associated CPN can be used to achieve some property verification.

**Summary of the syntax** Recall that Table 2 summarises the syntactic elements we consider in our translation. Deep history pseudostates, submachine states and implicit forks/joins were discarded for sake of simplicity but can be added in a very straightforward manner. We did not consider entry/exit points and junction pseudostates, but we believe that there would be no difficulty for adding them to our scheme. Choice pseudostates might be more tricky due to the dynamic evaluation of guards. 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.

### 7.1. Discussion on Complexity

Let us briefly discuss the complexity of the resulting CPN w.r.t. the size of the source SMD. First, let us consider the CPN output by Section 4. Concerning CPN places: each SMD simple state (including final states) and each initial pseudostate is encoded into one CPN place. This gives a number of places linear in the number of simple states and initial pseudostates of the source SMD. Concerning CPN transitions: first, each “do” behaviour is translated into a CPN transition. Second, consider an SMD transition from a composite state to another composite state labelled by an event (which is the worst situation): note that there is a single set of target states (basically the simple states target of the initial pseudostates in the target composite states). Concerning the source states however, there may be a combinatorial explosion due to the orthogonal regions: a rough upper bound (*i.e.* in the worst case) on the maximal number of CPN transitions due to one such SMD transition is  $n^m$ , where  $n$  is the maximal number of direct substates in a region, and  $m$  is the maximum number of orthogonal regions (including indirect subregions) in a composite state (*i.e.* this is the maximal number of simple states to be active at the same time in the SMD). For example, in Fig. 9(a),  $n = 3$  (there are three states in the top-level region) and  $m = 4$  (there are at most four active states at a time).

This exponential complexity might seem prohibitive, but we believe it is not for the following reasons.

First, we did not find huge case studies of UML state machines for which an exponential blow-up would clearly prevent formal verification. Second, although our destination translation explicitly represents these combinations, not representing them explicitly would not spare the model checker from considering them in the verification: let us assume we use a model (yet to be defined) of hierarchical coloured Petri nets that would allow the representation of SMD transitions, the destination model would be much more compact, but the verification would (most probably) be performed in an identical manner. That is, each of the combinations should still be considered during the model checking phase to assert the system safety.

In fact,  $n^m$  is (in the worst case) the number of transitions in the destination CPN, but the number of cases to be considered by the model checker may be larger, due to the use of the *interleave* function. Recall that *interleave* considers the interleaving of all possible entry/exit behaviours (following the semantics of SMDs); this might yield an additional factorial in the number of behaviours to be executed on a single transition, *i.e.* depending on the maximal depth of the state hierarchy and the maximal number of active states at a time. However, once more, we believe that the number of such transitions (with their respective behaviours) does not fundamentally depend on the choice of the target formalism and model-checker – as all possible behaviours must be considered by model-checking anyway.

However, the solution extending transition selection (Section 5) has a much smaller complexity in terms of number of transitions: there is only one transition per completion transition in the source UML SMD, plus one (fused) transition per different event in the SMD. Note however that the code segment attached to these transitions is more complex.

## 7.2. Future Work

**Implementation** Our first main future work is to develop a tool automating the translation so as to be able to perform formal verification of larger-scale SMDs, study the complexity of our translation and of the verification, and propose optimisations if needed.

We implemented a first prototype using Acceleo<sup>7</sup>, but this technology turned out to be slightly inaccurate for our framework (as reported in [ABC14b]). Although we could translate toy examples using this prototype, we started the development on a new standalone translation framework, the implementation of which is currently in progress.

Several tools such as Rhapsody [HG97], MagicDraw<sup>8</sup> and Papyrus [Ger15] allow the modelling of systems using UML and its diagrams. They also feature simulation and analysis of the diagrams. We choose Papyrus as a tool to model our state machine diagrams for two reasons: (i) Papyrus is a plugin for Eclipse and it conforms to the specification of the OMG, and (ii) Papyrus outputs files in an XMI syntax that is compatible with the SAX parser used in our ongoing prototype implementation.

We use Papyrus 1.1.0 [Ger15] to model our UML state machine diagram, the SAX parser 2.0.1<sup>9</sup> to parse the UML state machine diagram; then, our translation algorithm and the converter to the input syntax of CPN Tools are being implemented in Java.

**Model checking** Beyond the natural applications of model-checking to the resulting CPN (safety, boundedness of some variables, liveness, etc.), an interesting application of model-checking is to detect whether a UML state machine is ill-formed, for some definitions of ill-formedness. Checking proper initialization of variables, conformance of the UML state machine to the UML or to our assumptions (*e.g.* no “do” behaviour on composite states) would be very straightforward. More interesting and challenging issues would be to address more elaborate definitions of ill-formedness. For example, can it happen that none of the guards of a choice pseudostate (dynamically) evaluates to true at runtime? This cannot be statically checked, and model-checking seems to be a good technique as it explores all possible situations that can arise in a given state machine.

**Comparison with other tools** A very interesting future work (once we will have an automated translation tool) will be to compare the output of our translation with other tools, on a set of UML state machines benchmarks. Performing such a comparison (syntactic aspects considered, semantics defined by the tool or an

<sup>7</sup> <https://www.eclipse.org/acceleo>

<sup>8</sup> <http://www.nomagic.com/products/magicdraw.html>

<sup>9</sup> <https://docs.oracle.com/javase/tutorial/jaxp/sax/parsing.html>

underlying translation, comparison of the performances both in translation and model-checking, comparison of the kinds of verifications that can be performed) will be of high interest, and this is something that we aim at addressing in a next step.

**About hierarchy** Our translation can be seen as a “flattening” of the state machine. In contrast, previous approaches *did* preserve the hierarchy in the destination formalism, using hierarchical coloured Petri nets (HCPNs). This is the case of [CKZ11], which was made possible because no entry/exit behaviours were considered in [CKZ11]. However, we do not see how to use HCPNs in our approach: indeed, the transitions labelled with an event imply to leave a composite state (and hence the corresponding hierarchical place) suddenly, though after executing the appropriate behaviours. This does not seem to be feasible using HCPNs. In fact, a challenging future work would be to propose an extension of HCPNs to allow such “exception-like” transitions, with some behaviours on exit/entry, similar to UML state machines.

**Extensions** We aim at extending our work to the case of multiple UML classes, so as to have communicating state machines (as in, *e.g.* [PG06]).

Most other syntactic aspects not considered in our work (see Table 2) could be added in a rather straightforward manner – except for timing aspects. Adding timing aspects to our translation is an interesting future work that we are interested to tackle in a near future. This will allow us to consider more expressive models with timing constraints (such as timed protocols, real-time systems, etc.). Such an extension could benefit from previous attempts to equip UML (or a part of its syntax) with time, *e.g.* [MGT09, OMG11, ACN14].

Natural destination formalisms for our translation are (extensions of) Petri nets extended with time. This includes timed Petri nets, time Petri nets (with a dense time semantics) [Mer74], but also more expressive models such as coloured Petri nets extended with time. These formalisms are supported by model checking tools: CPNTools supports CPNs extended with discrete time, Tina [BV06] supports time Petri nets, and Roméo [LRST09] supports parametric time Petri nets [TLR09] extended with global variables.

**Integration with other diagrams** Another important future work is to integrate activity diagrams in the translation of UML state machines. The combination of the two diagrams allows us to model more aspects of the system at the same time. We could get inspired by previous works formalising activity diagrams, *e.g.* using coloured Petri nets [ACR13]. Also note that a subset of the syntax of SysML state machines and activity diagrams is translated in a homogeneous manner using CSP in [JS15].

**Semantics equivalence** Finally, although it goes beyond of the scope of this paper, a challenging future work will be to formally prove the equivalence between the original SMD and the resulting CPN. Of course, a problem is that the OMG does not define a fully formal semantics for SMDs. However, we could reuse the operational semantics that we recently proposed for SMDs [LLA<sup>+</sup>13], and define a trace equivalence taking into account active states, behaviours and events.

## Acknowledgements

We are grateful to the anonymous reviewers for very useful comments. We would also like to thank Sami Evangelista for his help when using CPN Tools.

## References

- [ABC14a] Étienne André, Mohamed Mahdi Benmoussa, and Christine Choppy. Formalising concurrent UML state machines using coloured Petri nets. In *Proceedings of the 6th International Conference on Knowledge and Systems Engineering (KSE'14)*, volume 326 of *Advances in Intelligent Systems and Computing*, pages 473–486. Springer, 2014.
- [ABC14b] Étienne André, Mohamed Mahdi Benmoussa, and Christine Choppy. Translating UML state machines to coloured Petri nets using Acceleo: A report. In *Proceedings of the 3rd International Workshop on Engineering Safety and Security Systems (ESSS 2014)*. EPTCS, 2014.
- [ACK12] Étienne André, Christine Choppy, and Kais Klai. Formalizing non-concurrent UML state machines using colored Petri nets. *ACM SIGSOFT Software Engineering Notes*, 37(4):1–8, 2012.
- [ACN14] Étienne André, Christine Choppy, and Thierry Noulamo. Modelling timed concurrent systems using activity diagram patterns. In Viet-Ha Nguyen, Anh-Cuong Le, and Van-Nam Huynh, editors, *Proceedings of the 6th*

- International Conference on Knowledge and Systems Engineering (KSE'14)*, volume 326 of *Advances in Intelligent Systems and Computing*, pages 339–351. Springer, 2014.
- [ACR13] Étienne André, Christine Choppy, and Gianna Reggio. Activity diagrams patterns for modeling business processes. In Roger Lee, editor, *11th International Conference on Software Engineering Research, Management and Applications (SERA'13)*, volume 496 of *Studies in Computational Intelligence*, pages 197–213. Springer, 2013.
- [Bee02] Michael von der Beeck. A structured operational semantics for UML-statecharts. *Software and Systems Modeling*, 1(2):130–141, 2002.
- [BP01] Luciano Baresi and Mauro Pezzè. On formalizing UML with high-level Petri nets. In Gul Agha, Fiorella de Cindio, and Grzegorz Rozenberg, editors, *Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*, pages 276–304. Springer, 2001.
- [BRS00] Egon Börger, Elvinia Riccobene, and Joachim Schmid. Capturing requirements by abstract state machines: The light control case study. *Journal of Universal Computer Science*, 6(7):597–620, 2000.
- [BV06] Bernard Berthomieu and François Vernadat. Time Petri nets analysis with TINA. In *Proceedings of the Third International Conference on the Quantitative Evaluation of Systems (QEST 2006)*, pages 123–124. IEEE Computer Society, 2006.
- [CJ09] Mats Carlsson and Lars Johansson. Formal verification of UML-RT capsules using model checking. Master's thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, 2009.
- [CKZ11] Christine Choppy, Kais Klai, and Hacene Zidani. Formal verification of UML state diagrams: a Petri net based approach. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011.
- [FS07] Harald Fecher and Jens Schönborn. UML 2.0 state machines: Complete formal semantics via core state machine. In *Proceedings of the 11th International Workshop on Formal Methods: Applications and Technology (FMICS 2006)*, volume 4346 of *Lecture Notes in Computer Science*, pages 244–260. Springer, 2007.
- [Ger15] Sébastien Gerard. Papyrus UML Modeling tool 1.1.2. <https://www.eclipse.org/papyrus/>, September 2015.
- [GLM02] Stefania Gnesi, Diego Latella, and Mieke Massink. Modular semantics for a UML statechart diagrams kernel and its extension to multicharts and branching time model-checking. *Journal of Logic and Algebraic Programming*, 51(1):43–75, 2002.
- [GP98] Martin Gogolla and Francesco Parisi Presicce. State diagrams in UML: A formal semantics using graph transformations - or diagrams are nice, but graphs are worth their price. In *ICSE Workshop on Precise Semantics of Modelling Techniques*, pages 55–72, 1998.
- [HG97] David Harel and Eran Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, 1997.
- [Hol03] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 2003.
- [JDJ+06] Toni Jussila, Jori Dubrovin, Tommi Junttila, Timo Latvala, and Ivan Porres. Model checking dynamic and hierarchical UML state machines. In *MDV*, 2006.
- [JK09] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009.
- [JS15] Jaco Jacobs and Andrew Simpson. A formal model of SysML blocks using CSP for assured systems engineering. In *Proceedings of the 3rd International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS 2014)*, volume 476 of *Communications in Computer and Information Science*. Springer, 2015. To appear.
- [KMR02] Alexander Knapp, Stephan Merz, and Christopher Rauh. Model checking - timed UML state machines and collaborations. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, 7th International Symposium, FTRTFT*, volume 2469 of *Lecture Notes in Computer Science*, pages 395–416. Springer, 2002.
- [LBC99] Gerald Lüttgen, Michael von der Beeck, and Rance Cleaveland. Statecharts via process algebra. In *10th International Conference on Concurrency Theory CONCUR*, volume 1664 of *Lecture Notes in Computer Science*, pages 399–414. Springer, 1999.
- [LHS08] Jiexin Lian, Zhaoxia Hu, and Sol M. Shatz. Simulation-based analysis of UML statechart diagrams: methods and case studies. *Software Quality Journal*, 16(1):45–78, 2008.
- [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 *Proceedings of the 10th International Conference on Integrated Formal Methods (iFM'13)*, volume 7940 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2013.
- [LMM99] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, V11(6):637–664, 1999.
- [LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.
- [LRST09] Didier Lime, Olivier H. Roux, Charlotte Seidner, and Louis-Marie Traonouez. Romeo: A parametric model-checker for Petri nets with stopwatches. In Stefan Kowalewski and Anna Philippou, editors, *15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, volume 5505 of *LNCS*, pages 54–57. Springer, March 2009.
- [Mer74] Philip Meir Merlin. *A study of the recoverability of computing systems*. PhD thesis, University of California, Irvine, CA, USA, 1974.
- [MGT09] Ahmed Mekki, Mohamed Ghazel, and Armand Toguyeni. Validating time-constrained systems using UML statecharts patterns and timed automata observers. In *VECoS*, pages 112–124. British Computer Society, 2009.
- [MPT03] Andrea Maggiolo-Schettini, Adriano Peron, and Simone Tini. A comparison of statecharts step semantics. *Theory of Computing Science*, 290(1):465–498, 2003.
- [NB02] Muan Yong Ng and Michael Butler. Tool support for visualizing CSP in UML. In *Proceedings of the 4th International Conference on Formal Engineering Methods (ICFEM 2002)*, volume 2495 of *Lecture Notes in Computer Science*, pages 287–298. Springer, 2002.



- [NB03] Muan Yong Ng and Michael Butler. Towards formalizing UML state diagrams in CSP. In *Proceedings of the 1st International Conference on Software Engineering and Formal Methods (SEFM 2003)*, pages 138–147. IEEE Computer Society, 2003.
- [OMG11] OMG. UML profile for modeling and analysis of real-time and embedded systems (MARTE), Version 1.1. <http://www.omg.org/spec/MARTE/1.1/PDF/>, June 2011.
- [OMG15] OMG. Unified Modeling Language Superstructure, Version 2.5. <http://www.omg.org/spec/UML/2.5/>, March 2015.
- [Per95] Adriano Peron. Statecharts, transition structures and transformations. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *6th International Joint Conference CAAP/FASETAPSOFT'95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 454–468. Springer, 1995.
- [PG00] Robert G. Pettit IV and Hassan Gomaa. Validation of dynamic behavior in UML using colored Petri nets. In *Proceedings of UML'2000 Workshop - Dynamic behaviour in UML models: Semantic Questions*, volume 1939 of *Lecture Notes in Computer Science*, pages 295–302. Springer Verlag, 2000.
- [PG01] Robert G. Pettit IV and Hassan Gomaa. Modeling state-dependent objects using colored Petri nets. In *Proceedings of Workshop on Modelling of Objects, Components, and Agents*, pages 105–120, 2001.
- [PG06] Robert G. Pettit IV and Hassan Gomaa. Modeling behavioral patterns of concurrent objects using Petri nets. In *9th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC*, pages 303–312. IEEE Computer Society, 2006.
- [Sam09] Miro Samek. *A crash course in UML state machines*. Quantum Leaps, LLC, 2009.
- [Sch05] Jens Schönborn. Formal semantics of UML 2.0 behavioral state machines. Technical report, Institute of Computer Science and Applied Mathematics, Christian-Albrechts-University of Kiel, 2005.
- [SLDP09] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. PAT: Towards flexible verification under fairness. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009)*, volume 5643 of *Lecture Notes in Computer Science*. Springer, 2009.
- [TH08] Yann Thierry-Mieg and Lom-Messan Hillah. UML behavioral consistency checking using instantiable Petri nets. *Innovations in Systems and Software Engineering (ISSE)*, 4(3):293–300, 2008.
- [TLR09] Louis-Marie Traonouez, Didier Lime, and Olivier H. Roux. Parametric model-checking of stopwatch Petri nets. *Journal of Universal Computer Science*, 15(17):3273–3304, 2009.
- [TZ05] Jan Trowitzsch and Armin Zimmermann. Real-time UML state machines: An analysis approach. In *Workshop on Object Oriented Software Design for Real Time and Embedded Computer Systems (Net.ObjectDays 2005)*, 2005.
- [Wes13] Michael Westergaard. CPN Tools 4: Multi-formalism and extensibility. In *Proceedings of the 34th International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2013)*, volume 7927 of *Lecture Notes in Computer Science*, pages 400–409. Springer, 2013.
- [ZL10] Shaojie Zhang and Yang Liu. An automatic approach to model checking UML state machines. In *SSIRI-C*, pages 1–6. IEEE, 2010.