Softw Syst Model
DOI 10.1007/s10270-015-0509-6

REGULAR PAPER

# Model checking multi-level and recursive nets

**Mirtha Lina Fernández Venero**[1] · **Flávio Soares Corrêa da Silva**[2]

Received: 18 March 2015 / Revised: 27 August 2015 / Accepted: 2 November 2015


**Abstract** With the increasing complexity of the problems and systems arising nowadays, the use of multi-level models is becoming more frequent in practice. However, there are still few reports in the literature concerning methods for analyzing such models without flattening the multi-level structure. For instance, several variants of multi-level Petri nets have been applied for modeling interaction protocols and mobility in multi-agent systems and coordination of cross-organizational workflows. But there are few automated tools for analyzing the behavior of these nets. In this paper we explain how to detect faults in models based on a representative class of multi-level nets: the nested Petri nets. We translate a nested net into a verifiable model that preserves its modular structure, a PROMELA program. This allows the use of SPIN model checker to verify properties related to termination, boundedness and reachability.

**Keywords** Multi-level modeling · Nested Petri nets · Model checking · SPIN

Communicated by Esther Guerra and Wil M. P. van der Aalst.

This is an extended and complete version of a preliminary work presented in [62].

✉ Mirtha Lina Fernández Venero
mirtha.lina@ufabc.edu.br

Flávio Soares Corrêa da Silva
fcs@ime.usp.br

[1] Center for Mathematics, Computation and Cognition, Federal University of ABC, Santo André, Brazil

[2] Department of Computer Science, University of São Paulo, São Paulo, Brazil

## 1 Introduction

Formal models are increasingly applied in software development due to the diversity and complexity of the systems (e.g., concurrent, distributed, real-time, mobile) arising in the present day. The models are used either at early stages of design to simulate and achieve understanding of the system behavior or at latter stages for validation or verification. The construction of a model is not a trivial task and even when they are designed to provide an abstract and reduced view of a system, their size may be large. A common approach to deal with a complex model is to organize it into layers having different levels of details. This multi-level approach produces models that are easier to understand, but it may also hide subtle faults that are harder to detect and correct. Nevertheless, there are few attempts to automatically verify the consistency of a multi-level model using multi-level aware methods.

The above issue is present in one of the most popular formalisms for analyzing properties of complex systems: the Petri nets (PNs) [45]. The PN framework is widely used for modeling concurrent, distributed, business processes and even biological systems [26,29,50,58]. Its success is due to the simplicity of its computation rule, its graphical representation and also to a large number of tools for simulation and verification. The formalism quickly evolved from a basic place/transition model into the high-level PNs [27]. A multi-layer representation was firstly introduced in the form of hierarchical colored PNs (HCPNs) by means of fusion places, substitution transitions and page instances. A more powerful step into the multi-level approach was the use of nesting and

Published online: 02 January 2016

recursion [3,17,28,30,32,38,56,59,60]. These two features have been applied in the context of distributed multi-agent, adaptive and dynamic systems [5,34,41], the coordination of inter-organizational workflows [47] and grid computing [43]. They are captured in a simple but effective way by the class of nested Petri nets (NPNs) in which the tokens may be multi-level and even recursive nets [38].

A NPN defines a set of net types; thus, the net tokens may be created, transported and consumed as ordinary ones. In addition, the net tokens may fire their own transitions autonomously or in synchrony with other net tokens at the same place (horizontal step) or in adjacent levels (vertical step). The behavior of the nets in lower levels may be influenced by the state of shared places at the topmost level. Furthermore, a transition may build new net tokens using operations such as the sequential, alternative or parallel composition [41]. These facilities provide a high degree of modularity and flexibility. NPNs, unlike HCPNs, are strictly more powerful than classical PNs. Therefore, some properties (e.g., reachability and boundedness) are undecidable [42]. However, for some important subclasses including the multi-level nets other properties such as termination can be decided [38].

Despite the benefits of using NPNs for multi-level modeling, the use of HCPNs prevails in practical applications. The main reason is the abundance of tools for simulating and verifying HCPNs while few of them support the features of the nets-within-nets approach. For example, one of the most popular tools for HCPNs, CPN Tools, provides an environment for editing and simulating a wide range of PNs. Besides, it includes options for generating the state space of the model and analyzing boundedness, liveness and CTL properties [48]. On the other hand, some multi-level models may be simulated using RENEW (for reference nets) [31] and PNTALK (for object-oriented Petri nets) [6]. However, these tools have no support for automated verification. CPN Tools was recently used to verify two-level bounded NPNs with vertical synchronization [9]. The translation into colored PNs is obtained by representing the net tokens as lists and unfolding all possible autonomous and synchronization steps. The resulting net can be analyzed using the state space tools. However, the flattening process removes the nesting structure producing a large number of transitions. This makes difficult the simulation of the NPN and the interpretation of the verification report. Furthermore, when counterexamples are found, the corresponding firing sequences in the NPN should be reconstructed by hand. Clearly, the extension of this approach to more levels would lead to rather large and obscure CPNs. To the best of our knowledge, no PN tool can be used to verify PNs with arbitrary nesting, several forms of synchronization and recursion. Therefore, this paper follows a different, but not new, path to the verification of a NPN: the use of a model checker.

### 1.1 Related work

Several variations of PNs and PN-like formalisms (e.g., workflows, business processes, UML diagrams) have been analyzed using model checking tools such as DVE [36], LTSA [49], NuSMV [11,57] and SPIN [1,15,16,20,33,52,55,64]. But few of these translations can be adapted to an arbitrary NPN. In most of the cases, either the tool language has no support for recursion (DVE, LTSA, NuSMV and STeP) or the translation avoids multiple levels or restricts the synchronization. For instance, in [12], two-level object nets are encoded as Prolog programs and verified using the XTL model checker [35,44]. Here, the synchronization was defined using a tree-like relation between transitions without locality constraints. However, the encoding of this general form of synchronization for the multi-level case was not provided. The semantics of recursive algebraic nets was defined in terms of a conditional rewriting theory in [2,18]. This allows the use of the LTL model checking tool of Maude [8,10]. But these nets do not include horizontal or transportation steps.

SPIN [21,23,54] is one of the most popular and successful model checkers for concurrent and distributed software systems. Therefore, it has been frequently used for verifying properties of PNs. SPIN uses a C-like language to specify the models called PROMELA (Process Meta Language) and efficient algorithms to deal with the state space explosion problem. The first work that addressed the synchronous firing of several transitions in a PN using SPIN was reported in [52,53]. This was achieved by considering all possible subsets of transitions that may fire simultaneously. Hence, in order to adapt this result to the multi-level case, a flattening process is required, similar to the one presented in [9]. A modular approach was described in [7] for two-level nested nets. However, in this translation the net tokens are not removed and the horizontal steps are not considered. Multi-level and recursive nets were firstly analyzed with SPIN in [62]. In this case, the horizontal steps were restricted to two net tokens and no transportation step was allowed. Besides, the conditions on the input arcs force the vertical steps to remove a single net token from each input place. An improved translation was outlined in [61], but due to space limitations few details were provided.

### 1.2 Contributions and outline of the paper

In this paper we get rid of the restrictions imposed in [62] and present a general and detailed translation of a NPN into PROMELA. One of the main features of this language is the dynamic creation of concurrent processes. Therefore, each net token is translated into a process and its marking is encoded in the state of the local variables. The process simulates not only the autonomous behavior of a child net but

also the interaction required for a synchronization. The latter is achieved by means of PROMELA asynchronous channels. We use these channels to store the messages that allow the vertical and horizontal steps as well as transporting or removing the net tokens. The resulting PROMELA program retains the modular and even recursive definition of a NPN.

This approach contrasts to the use of logic programming or conditional rewriting logic, where the marking of a net is represented using lists and terms and additional predicates/rules are required for simulating the behavior. We believe that the simulation of a NPN using directly the PROMELA model is more amenable than using a logic program or an equational/rewriting theory since no unification, matching or backtracking are involved. In addition, SPIN traces provide details not just about the transition but also the process (net token) and local variables (places) involved in each firing.

The paper also shows that SPIN can be very effective in detecting flaws of a NPN related to termination and boundedness as well as verifying LTL properties. Regarding verification, SPIN is faster than XTL model checker and can handle a larger number of properties and instances [13]. SPIN also outperforms Maude model checker in execution time and memory requirements [10]. Nevertheless, the paper discusses the factors that influence SPIN performance when verifying these nets.

The remaining sections are organized as follows. Section 2 introduces the basics of PNs and the formal definition of the NPNs we use in this paper. The translation of a NPN into PROMELA is presented in detail in Sect. 3 and the informal arguments of its correctness are provided in Sect. 4. Section 5 explains how to detect faults in a NPN-based model using SPIN and gives illustrative examples. Some issues about SPIN performance are discussed in Sect. 6. Section 7 draws some concluding remarks and future work.

## 2 Nested Petri nets

### 2.1 Preliminary notions of Petri nets

A Petri net [45] is a tuple $N = (P, T, A, W)$ where $P$ and $T$ are non-empty, finite and disjoint sets of places and transitions, resp. The elements in the set $A \subseteq (P \times T) \cup (T \times P)$ are called (input and output) arcs. The function $W$ is defined from $A$ to multisets of tokens; each multiset is called arc label or inscription. A *marking* of a PN is a function attaching a multiset of tokens to each place. The transitions change the marking of the net by means of events called *firings*. The firing of a transition may occur at a given marking if and only if the multiset on each input arc inscription is a subset of the multiset of tokens at the corresponding input place. If the event takes place, then the tokens in the input arc labels are removed from the input places and the tokens in the output arc labels are added to the output places.

The tokens in a PN have no structure or information. Therefore, they are called uncolored tokens and are usually represented as black dots. Since they have no difference, a multiset of black dots is usually replaced by its size. Thus, for a PN the function $W$ is usually defined from $A$ to $\mathbb{N}$ and a marking from $P$ to $\mathbb{N}$. In a colored Petri net (CPN) each place has a type; thus, it may host tokens with different data values, i.e., colors [25]. Each arc is labeled by a multiset that may contain constant values but also variables. The type of each element in the multiset should match type of the incident place. The firing of a transition is conditioned to the existence of a binding for the variables in the input arcs.

*Example 1* Figure 1 in the left shows an example of a PN. The places are drawn using circles and the transitions are represented using bars. In this paper, we write a marking $M$ as a sequence of pairs $p : M(p)$ enclosed by the symbols $\prec$ and $\succ$. Uncolored places will be marked with non-negative integers instead of multisets of black dots. Hence, the initial marking of the PN can be written as $\prec p_1 : 4, p_2 : 0, p_3 : 5, p_4 : 0\succ$. From this marking, the transition $t_1$ may fire twice: firstly leading to the marking $\prec p_1 : 2, p_2 : 2, p_3 : 3, p_4 : 2\succ$ and then to $\prec p_1 : 0, p_2 : 4, p_3 : 1, p_4 : 4\succ$. After that, $t_1$ is no longer enabled.

An example of a CPN is shown in the right of Fig. 1. Here, we assume that the type of the places and variables is the set of colors *red*, *green* and *blue*, denoted with the letters $R$, $G$ and $B$, resp. The initial marking of the CPN is $\prec p_5 : \{R, R, G, B\}, p_6 : \emptyset, p_7 : \{R, G, G, G, G\}, p_8 : \emptyset\succ$. In this case, the transition $t_2$ may fire once. Note that due to the marking of $p_7$ and the inscription on the input arc $(p_7, t_2)$, the variable $x$ must be bound to $G$. However, the firing may produce one of the six different markings because $y$ may be bound either to $R$ or to $B$ and we may bind $z$ to any color. Hence, after firing $t_2$ using, e.g., the binding $\{x = G, y = R, z = B\}$, the new marking of the CPN will be $\prec p_5 : \{R, B\}, p_6 : \{R, B\}, p_7 : \{R, G, G\}, p_8 : \{G, R\}\succ$. Regardless of the binding, the green token is removed from $p_5$. Therefore, no further firing can be performed.

A nested Petri net is a CPN in which tokens can also be Petri nets. More precisely, a NPN comprises several CPNs, $SN, EN_1, \ldots, EN_n$, one of them called *system net* ($SN$) [39]. As in CPNs, the definition of a NPN includes a set of basic types $\Sigma$ and a set of basic constants $\Sigma_c$ belonging to these types. In addition, each $EN_i$ (called *element net*) is also considered as a constant and a type. The set of values of an element net type consists of marked net tokens. We stress that $SN$ is not a type and it defines the structure of the uppermost level in the NPN.

The net tokens may fire their own internal transitions autonomously. The firing of a transition is performed accord-

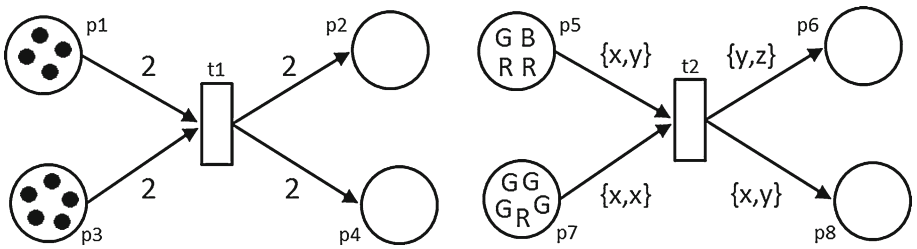



**Fig. 1** Examples of a classical PN (*in the left*) and a CPN (*in the right*)

ing to the classical CPN rules. Hence, child nets can be created, removed, copied or transported without changing the inner marking. The firing may also be synchronized with the firing of other net tokens at the same place (*horizontal synchronization step*) or with the child nets (*vertical synchronization step*). This is specified by means of two disjoint sets of labels that are attached to the transitions. The number of net tokens required for a horizontal synchronization is defined by associating a label with a natural number greater than 1 (*arity*) [61]. In some definitions of a NPN, the arities are related to places and tuples are used for grouping the net tokens [38,39].

### 2.2 A formal definition of a NPN

In the next, we assume the arc inscriptions are multisets over (basic or net) constants and variables of fixed type. We use the symbol _ as an anonymous variable to indicate a single token, regardless of its value. Each occurrence of _ represents a different variable. We denote as $Var(m)$ the set of non-anonymous variables occurring in the multiset $m$. For simplicity, we assume that the type of each net variable is the same as the type of the incident place and the arc inscriptions are well typed. We will use $\uplus$ to represent a disjoint union and $\mathbb{N}_2$ for the natural numbers greater than 1.

In this paper, the places in a NPN have either basic or net type (Definition 1, condition 3). As in [62], we allow some places of $SN$ to be shared by the element nets. This way the system net may influence the behavior of net tokens at any level. Besides, the net tokens may compete to get access to the shared resources. The shared places may host basic tokens or 1-level nets ($EN_1, \ldots, EN_b$ in Definition 1). In order to avoid conflicts in a synchronizing step, we require that the some labeled transitions have no shared place as input (Definition 1, condition 7). To this end, we separate the labels for vertical synchronization with the parent from those for synchronizing the child nets.

The definition of a NPN imposes some restrictions on the arc inscriptions because testing equality is not allowed [37,38,40,42]. Therefore, the input arcs of a transition must not contain net constants or multiple occurrences of the same variable (c.f. $t2$ in Fig. 1). The latter restriction avoids different interpretations of the behavior as those introduced by the value and reference semantics in the nets-within-nets paradigm [40,59]. These restrictions are formalized in Definition 1, conditions 8a, 8b and 8c. In addition, a constraint on output arc inscriptions is needed to obtain a finitely branching transition system [38]. Hence, each variable in an output arc must occur in some input arc of the same transition. This is stated in Definition 1, condition 8d. However, we note that this restriction is not necessary for variables of finite basic types.

**Definition 1** Let $N = (\Sigma, P_s, L, ar, (EN_0, EN_1, \ldots, EN_b, \ldots, EN_n))$ be a NPN s.t. $\Sigma$ is a finite set of basic types, $P_s$ is a finite set of shared places, $L = L_h \uplus L_v$ is a set of labels and $ar : L_h \to \mathbb{N}_2$. The set $L_v = L_v^+ \uplus L_v^-$ is s.t. $|L_v^+| = |L_v^-|$,

- for each $l \in L_v^+$, there is a complementary label $\bar{l} \in L_v^-$ and
- for all $l_1, l_2 \in L_v^+$, $l_1 \neq l_2$ implies $\bar{l}_1 \neq \bar{l}_2$.

For all $i = 0 \ldots n$, $EN_i = (P_i, V_i, C_i, I_i, T_i, \Lambda_i, A_i, W_i)$ is a colored Petri net (called net component) where

1. $P_i$ is a finite set of places s.t. if $i = 0$ then $P_s \subseteq P_i$ otherwise $P_i \cap P_s = \emptyset$;
2. $V_i$ is a set of variables;
3. $C_i$ is a type function s.t. if $0 < i \leq b$ then $C_i : P_i \cup V_i \to \Sigma$ otherwise $C_i : P_i \cup V_i \to \Sigma \cup \mathcal{P}(\{EN_1, \ldots, EN_n\})$. Besides, for all $p \in P_s$, $C_0(p) \in \Sigma \cup \mathcal{P}(\{EN_1, \ldots, EN_b\})$;
4. $I_i$, the function for the initial marking, is defined from $P_i$ into multisets over $\Sigma_c \cup \{EN_1, \ldots, EN_b\}$;
5. $T_i$ is a finite set of transitions s.t. $P_i \cap T_i = \emptyset$;
6. $\Lambda_i : T_i \to L \cup \{\epsilon\}$ is the labeling function. The symbol $\epsilon$ denotes the empty label and it is used for unlabeled transitions. For $SN$, we have that $\Lambda_0 : T_0 \to L_v^+ \cup \{\epsilon\}$;
7. $A_i$ is the set of arcs s.t. if $0 < i \leq b$ then $A_i \subseteq (P_i \times T_i) \cup (T_i \times (P_s^\Sigma \cup P_i))$ where $P_s^\Sigma$ is the subset of shared places with basic type. Otherwise $A_i \subseteq ((P_s \cup P_i) \times T_i) \cup (T_i \times (P_s \cup P_i))$ and satisfies that for all $(p, t) \in A_i$, if $\Lambda_i(t) \in L_h \cup L_v^-$ then $p \notin P_s$;
8. $W_i$ is defined from $A_i$ to multisets over $V_i \cup \Sigma_c \cup \{EN_1, \ldots, EN_n\}$ and s.t
   (a) there are no net constants in input arc inscriptions;
   (b) every variable has at most one occurrence in each input arc expression;

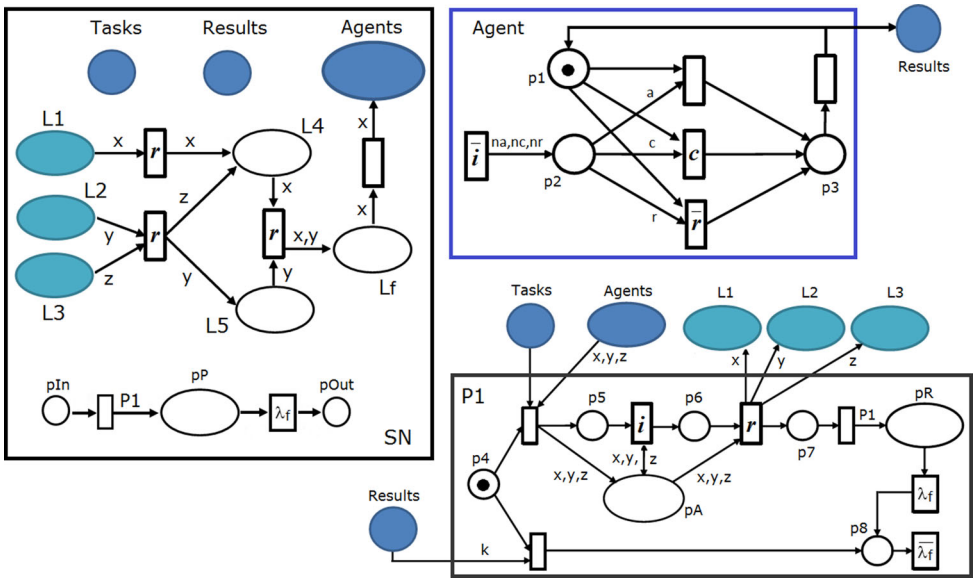



**Fig. 2** A NPN modeling a simple multi-agent environment

(c) given two different input arcs of the same transition $(p_1, t)$ and $(p_2, t)$, $Var(W_i(p_1, t)) \cap Var(W_i(p_2, t)) = \emptyset$;
(d) for each $x \in Var(W_i(t, q))$, there should be an input arc of $t$ s.t. $x \in Var(W_i(p, t))$. Besides, there are no anonymous variables in output arc expressions.

*Example 2* Figure 2 shows a simple NPN (adapted from [61]) modeling a multi-agent-based environment. The basic-typed places are drawn using circles but we use ellipses for net-typed ones. The shared places are colored in blue and the label of a transition appears inside the bar. For the sake of readability, we have omitted the braces of the multisets in the arc inscriptions. The input arcs with no inscription are assumed labeled as {_}, i.e., they remove a single token. The output arcs without inscription are assumed labeled as $\{\bullet\}$.

The system net of this NPN consists of an environment with locations $(L1, L2, L3, L4, L5, Lf)$ and places for agents, tasks, results and a call for a protocol net $(pP)$. The *Agent* element net specifies the behavior of an agent that may execute activities autonomously, in collaboration (horizontal synchronization) or in response to an external request (vertical synchronization). These activities are represented by the basic colors $a, c, r$, respectively. The number of activities, represented by the constants $na, nc, nr$, is defined when a net token is created. Hereafter, we denote a new *Agent* net as $A(na, nc, nr)$. The NPN also includes a recursive element net $P1$. This net defines a protocol for situating agents at specific locations in the environment. The arc inscription $k$ stands for a multiset of $k$ anonymous variables. Therefore, the recursion of the protocol terminates when the agents produce at least $k$ results.

### 2.3 State of a NPN

The state or configuration of a PN, called *marking*, is a distribution of tokens over the places. In a NPN, each net token has its own marking. Hence, a *marking of an element net* $EN_i$, $1 \leq i \leq n$, is defined using induction as follows.

1. A function $M$, mapping each place $p \in P_i$ to a finite multiset over $\Sigma$ is a marking of $EN_i$. The pair $(EN_i, M)$ is called a marked element net or net token of $EN_i$.
2. Let $\bar{\Sigma}$ be a set of marked element nets. Then, a function mapping each place $p \in P_i$ to a finite multiset over $\bar{\Sigma} \cup \Sigma$ is also a marking of $EN_i$.

A *marking of a NPN* $N$ is a function mapping each place $p \in P_0$ to a finite multiset over $\bar{\Sigma} \cup \Sigma$, i.e., a marking of $EN_0$. Any marking must match the type of the places. The *initial marking* of any net component is obtained from the initial function $I_i$. By definition, this function has no net token of type $EN_{b+1}, \ldots, EN_n$. The initial marking of $N$ is obtained from $I_0$. For all $i > 0$, $EN_i$ also represents a constant corresponding to the marked net $(EN_i, I_i)$.

Figure 3 shows a marking for the NPN in Example 2 with three agent nets $A(0, 1, 3)$, two tasks and a black dot at $pIn$. Note that the three agent tokens share the *Results* place in $SN$. To avoid confusion, it can be assumed that the places, transitions, variables and arcs of two net tokens of the same element net are different.

### 2.4 The behavior of a NPN

The behavior of a PN is defined by the firing sequences it may execute from an initial state. In a NPN each step of the sequence may involve the firing of several transitions.

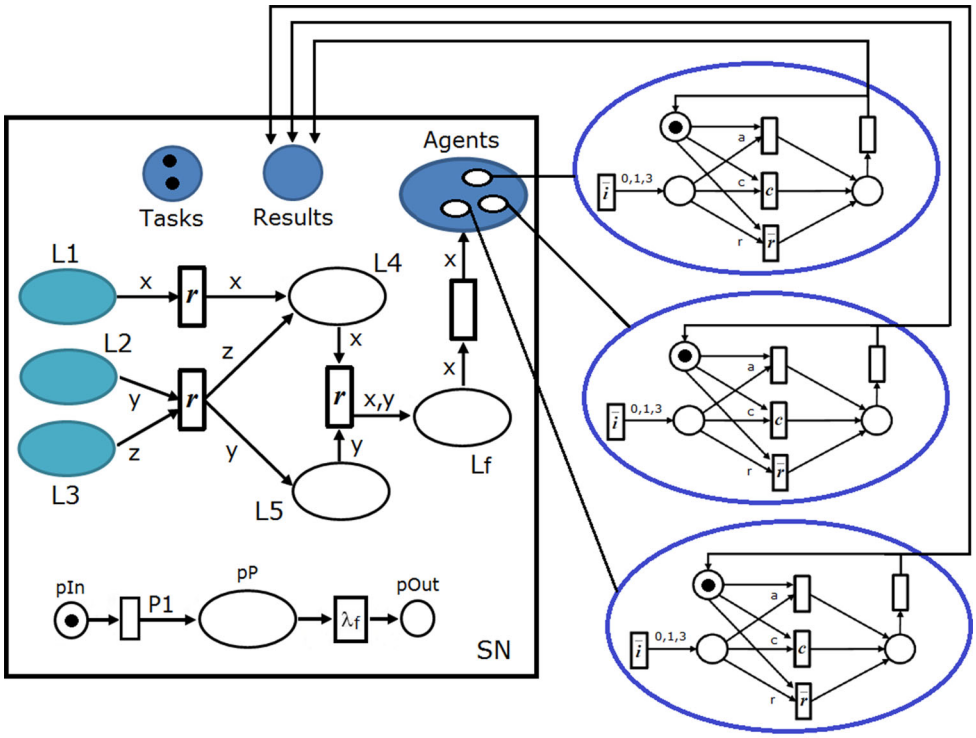



**Fig. 3** Initial marking for the NPN in Example 2

As in CPNs, each firing is conditioned to the binding of the variables in the input arcs. These concepts are defined in the next. We will say that a net token $nt = (EN_i, M')$ occurs in a marking $M$ of a NPN if there is a place $p \in P_0$ s.t. either $nt \in M(p)$ or there exists $(EN_j, M'') \in M(p)$ and $nt$ occurs in $M''$. The occurrence of two net tokens at the same place in a marking is defined analogously. The replacement of $nt$ in $M$ by a net token $nt_1 = (EN_i, M'_1)$ (denoted as $M[nt \rightarrow nt_1]$) is defined as the marking $M_1$ s.t. for each $p \in P_0$

- $M_1(p) = M(p) - \{nt\} \cup \{nt_1\}$ if $nt \in M(p)$;
- $M_1(p) = M(p) - \{(EN_j, M'')\} \cup \{(EN_j, M''[nt \rightarrow nt_1])\}$ if there exists $(EN_j, M'') \in M(p)$ and $nt$ occurs in $M''$;
- $M_1(p) = M(p)$ otherwise.

**Binding** Given a transition $t$, we write $Var^*(t)$ for the set of all variables occurring in the input arcs of $t$. Hereafter, we assume that $W(p, t)$ (respectively $W(t, p)$) is the empty set if and only if $(p, t) \notin A$ (respectively $(t, p) \notin A$). A *binding* for $t$ is a function $b$ assigning to each variable $x \in Var^*(t)$ a value from $\bar{\Sigma} \cup \Sigma$ (of the corresponding type). It is applied to multisets in a straightforward way. The elements in the multiset $\{b(x) \mid x \in Var^*(t) \wedge C(x) \notin \Sigma\}$ are the net tokens involved in $t$ w.r.t. the binding $b$.

**Firing** Let $M$ be a marking of a NPN $N$. A transition $t \in T_0$ is *enabled* in $M$ w.r.t. a binding $b$, if and only if for all $a = (p, t) \in A_0, b(W_0(a)) \subseteq M(p)$. In this case, $t$ may fire. After the firing, a new marking $M_n$ is obtained s.t. for any place $p \in P_0$, $M_n(p) = M(p) - b(W_0(p, t)) \cup b(W_0(t, p))$.

Let $(EN_i, M')$ be a net token occurring in $M$ at some place $p'$. A transition $t$ of $(EN_i, M')$ is *enabled* in $M$ w.r.t. a binding $b$, if for all $a = (p, t) \in A_i$, $b(W_i(a)) \subseteq M'(p)$. If $t$ fires, a new marking $M''$ is obtained from $M'$ s.t. for any place $p \in P_i$, $M''(p) = M'(p) - b(W_i(p, t)) \cup b(W_i(t, p))$. Furthermore, a new marking $M_n$ of $N$ is obtained from $M$ s.t.

- $M_n(p') = M[(EN_i, M') \rightarrow (EN_i, M'')]$;
- for any place $p \in P_s - \{p'\}$, $M_n(p) = M(p) \cup b(W_i(t, p))$ and
- for any place $p \notin P_s \cup \{p'\}$, $M_n(p) = M(p)$

**Step** A NPN may perform autonomous and synchronizing steps; the latter divided into horizontal and vertical. An *autonomous step* is the firing of a single unlabeled transition in $SN$ or in a net token of $N$. A *horizontal* step allows the synchronization of $k$ net tokens that occur in $M$ at the same place. In this case a transition labeled as $l$, with $l \in L_h$ and $ar(l) = k$, should be enabled in all the $k$ subnets. The horizontal step is the simultaneous firing of these transitions. A *vertical step* is the firing of an enabled transition $t$ in $SN$ or some net token that occurs in $M$ such that $l = \Lambda(t) \in L_v^+$, and the firing of a transition labeled as $\bar{l}$ in all net tokens involved in the binding of $t$.

We denote a step as $M[\rangle M_n$ where $M_n$ is the marking obtained after firing all involved transitions. We stress that the order for firing the transitions in a synchronizing step is

irrelevant because they do not share input places. We also note that, a vertical synchronization occurs at two adjacent levels of nesting: Transitions in $L_v^+$ are intended for *lower* synchronization (with child net tokens) while those in $L_v^-$ are for *upper* synchronization (with the parent). Without this separation, there is no clear distinction if a label is intended for synchronizing with the parent or with the children. Since both may be enabled, it would be possible to perform a vertical step involving several adjacent levels of nesting.

A marking $M$ is called *reachable* if there is a sequence of zero or more steps $I_0[\rangle M_1[\rangle \ldots [\rangle M_k$ such that $M_k = M$. This is denoted as $I_0[*\rangle M$. It is called *dead* if no step can be done from it. A NPN terminates if and only if it has no infinite firing sequence. An infinite firing sequence is a cycle if and only if there is a reachable marking $M$ such that $I_0[*\rangle M[\rangle M'[*\rangle M$.

*Example 3* In the next we illustrate some steps that can be performed by the NPN of Example 2, starting from the initial state in Fig. 3. We assume $ar(c) = 3$ and $k = 5$. A firing sequence may initiate with the following steps:

1. *SN* removes the black dot at *pIn* and creates a *P1* net token at $pP$.
2. The *P1* token executes an autonomous step that consumes a black dot from $p4$ and the shared place *Tasks* produces a black dot at $p5$ and transports the three agents to $pA$.
3. A vertical step is performed using the transition labeled as $i$ in the *P1* token and the transitions labeled as $\bar{i}$ in the agent nets. This step removes the token at $p5$, adds a black dot at $p6$ and the agent nets are put back at $pA$, all of them marked as $\ll p_1 : 1, p_2 : \{c, r, r, r\}, p_3 : 0\gg$.
4. The three agent nets at $pA$ perform a horizontal step: Each transition labeled as $c$ fires, leading to the inner marking $\ll p_1 : 0, p_2 : \{r, r, r\}, p_3 : 1\gg$.

5–7. Then, each agent net executes an autonomous step that adds a token at *Results* and produces the inner marking $\ll p_1 : 1, p_2 : \{r, r, r\}, p_3 : 0\gg$. After the three steps, the agents remain at $pA$ and $Results = 3$.

8. The $P1$ net performs a vertical step using the transition labeled as $r$ and the transitions labeled as $\bar{r}$ in the agents. This step removes the token at $p6$, adds a token at $p7$ and distributes the agent nets at $L1$, $L2$ and $L3$. Now, the inner marking of each agent is $\ll p_1 : 0, p_2 : \{r, r\}, p_3 : 1\gg$.

9–11. Similarly to steps 5–7, each agent net executes an autonomous step that adds a token at *Results* and produces the inner marking $\ll p_1 : 1, p_2 : \{r, r\}, p_3 : 0\gg$. After the three steps, each agent remains at its place and $Results = 6$.

12. An autonomous step by the protocol net removes the token at $p7$ and creates a nested $P1$ net token at $pR$.
13. The nested $P1$ token removes the token at $p4'$ and 5 tokens from $Results$, adding a black dot at $p8'$.
14. The two protocol nets perform a vertical step involving the transitions labeled as $\lambda$ and $\bar{\lambda}$. This step consumes the inner $P1$ net and produces a black dot at $p8$.
15. *SN* synchronizes with the net at $pP$ by a vertical step involving the transitions labeled as $\lambda$ and $\bar{\lambda}$. This removes the protocol net and produces a black dot at $pOut$.

At this point, the NPN may execute other steps, e.g., a vertical step at *SN* transporting the agent at $L1$ to $L4$ or the agents at $L2$ and $L3$ to $L5$ and $L4$, respectively. Note that, if we remove steps 4–7 and execute the latter vertical synchronization before step 12, we obtain a different sequence that leads to the removal of the protocol net at $pP$. An interesting question for this example is whether all agents return to the initial place after completing all activities and tasks. Currently, no PN tool can be used to automatically solve this question. We explain in Sect. 5.1 how to verify this property using SPIN.

## 3 Translating a nested Petri net into PROMELA

### 3.1 A brief introduction to PROMELA syntax and semantics

The language of SPIN model checker, called PROMELA, is a process-oriented language with a C-like syntax. The definition of a process template (`proctype`) uses parameters, variable declarations and imperative statements such as expressions, assignments, jumps, conditionals, loops and actions for communication. The declarations, expressions and assignments follow the standard syntax of C and other programming languages. However, the `do` and `if` statements consist of several branches (called options) that may be non-deterministically selected. Figure 4 shows a subset of the PROMELA syntax rules we use in this paper.[1] The non-terminal symbols are written in italics starting with an uppercase letter while the terminals are written in teletype. In the first rule, we have used the Kleene star to indicate zero or more occurrences of the components in a model.

In PROMELA, a process may create other process instances that are executed concurrently. To this end, the `run` operator is used which returns the process instantiation number. This number is stored in the predefined, local and read-only variable `_pid`. An initial process can be specified using the `init` keyword but no other instance of this main process can be created. The processes may communicate by

[1] See the complete grammar at http://spinroot.com/spin/Man/grammar.html.

**Fig. 4** A small subset of PROMELA grammar

$$
\begin{aligned}
Model &\rightarrow (\ VarDec \mid ProctypeDec \mid LTLDec \mid OtherDec\ )* \\
Proctype &\rightarrow \texttt{proctype}\ Name\ \texttt{(}\ Param\ \texttt{)}\ \{\ SeqStm\ \} \mid \texttt{init}\ \{\ SeqStm\ \} \\
Stm &\rightarrow BasicStm \mid NonDetStm \mid \texttt{atomic}\ \{\ SeqStm\ \} \mid Label\ \texttt{:}\ Stm \\
BasicStm &\rightarrow Exp \mid Var\ \texttt{=}\ Exp \mid \texttt{assert(}\ Exp\ \texttt{)} \mid \texttt{break} \mid \texttt{skip} \mid ComStm \\
ComStm &\rightarrow Var\ \texttt{!}\ Msg \mid Var\ \texttt{?}\ Msg \mid Var\ \texttt{!!}\ Msg \mid Var\ \texttt{??}\ Msg \\
NonDetStm &\rightarrow \texttt{if}\ SeqOption\ \texttt{fi} \mid \texttt{do}\ SeqOption\ \texttt{od} \\
Option &\rightarrow \texttt{::}\ SeqStm \mid \texttt{::else}\ SeqStm \\
Exp &\rightarrow IntExp \mid \texttt{run}\ Name\ \texttt{(}\ ExpList\ \texttt{)} \mid \texttt{set\_priority(}\ Expr\texttt{,}Expr\ \texttt{)} \mid \\
&\quad \texttt{len(}\ Var\ \texttt{)} \mid \texttt{empty(}\ Var\ \texttt{)} \mid Var\texttt{?[}\ Msg\ \texttt{]} \mid Var\texttt{??[}\ Msg\ \texttt{]}
\end{aligned}
$$

sending and receiving messages via channels that may be either asynchronous (*buffered*) or synchronous (*rendezvous*). The standard send and receive statements (`!` and `?` resp.) assume the buffered channels as *fifo* lists. PROMELA offers several predefined functions for buffered channels, e.g., `len` for the number of messages and `empty`. It also includes other variants of send and receive such as the sorted send (`!!`) and the random receive (`??`). There are also side effect-free operators for testing the presence of a message at a buffered channel. These operators, known as channel poll, are written as a receive statement with the message enclosed in square brackets (see the last two rules). For a rendezvous channel, the buffer size is zero. Since no message can be stored, the send and receive operations via such a channel should be executed simultaneously by two process instances.

A statement in a process can be executed if it is enabled; otherwise it is blocked. Declarations, assignments and jumps are always enabled. For the remaining statements there are conditions for their executability. For instance, an expression is enabled if it evaluates to a nonzero value.[2] A sequence of statements is enabled if the first statement is so. A conditional or loop statement is enabled if at least one of its options is enabled. In this case, an enabled option is nondeterministically chosen. The special option prefixed by `else` is enabled if and only if no other option is so. If the current statement is blocked, the execution of the process is suspended until the statement becomes enabled.

The concurrency in PROMELA has an asynchronous interleaving semantics. Therefore, statements of different processes are not executed simultaneously (except the rendezvous communication). Besides, the execution of a process may be interrupted by any other enabled process. Nevertheless, the statements in an `atomic` sequence are not interleaved, unless one of them blocks. The first statement of an `atomic` sequence or an option is called the *guard* and it is usually separated from the remaining statements using the symbol `->`. The scheduling of the enabled processes is non-deterministic. However, the processes may have priorities. In this case, an enabled statement/process is selected if there is no other enabled statement/process with higher priority.

The first tests of a PROMELA model may be done using SPIN random or interactive simulation. After that, a verifier can be generated for checking safety and liveness properties. The properties can be specified using linear-time temporal logic (LTL) formulae or a `never` claim. Besides, assertions (`assert`) and special labels for verification (`end`, `progress` and `accept`) can be placed as any other statement and label in the model. Any violation of a property (counterexample) is reported by the verifier as an error tail that can be inspected using a guided simulation. Very often the counterexamples are found using SPIN basic search algorithms. However, a full verification of the correctness claims may require advanced or approximation techniques to cope with the state space explosion, such as the partial-order reduction and the bitstate hashing [22,24]. For further details of PROMELA and SPIN see, e.g., [20,21,23,54].

### 3.2 Translation for the system net of a NPN

In this section we describe how to translate a NPN into a PROMELA program. Each element net of a NPN is encoded as a process definition; thus each net token is a process. The places are represented as local variables of the `proctype` definition, but the shared places are declared as global variables. The variables used in the arc inscriptions are also translated as local variables. The system net is translated as the `init` process of the PROMELA program. The first step of the program (Fig. 5, line `9`) sets the initial marking of *SN*, producing the tokens at the corresponding places as we will explain below.

Uncolored places are best represented as non-negative integer variables. We assume that the initial marking of these places is always defined as the initialization part of the variable declaration. Basic constants can be represented as integer values or symbolic names. We translate each colored place of basic type into an asynchronous channel that stores the tokens. For simplicity, we use a straightforward represen-

[2] The `skip` statement is equivalent to the constant `1`, i.e., it is always executable.

```
1  #define BasicPlace(bp, nTok) chan bp = [nTok] of {byte}
2  #define NetPlace(np,nMsg)  chan np = [nMsg] of {byte,byte,byte}
3  chan gbChan = [MaxMsg] of {byte, byte, byte, chan, bit};
4   /* Shared Places, Auxiliary Code, Element Nets */
5  init(){
6  /* Non-Shared Places and Basic Typed Arc Variables */
7  atomic{ set_priority(0, 2)  /* Initial Marking */  set_priority(0, 1) }
8  do :: atomic{ empty(gbChan) && enableTest_t ->
9           set_priority(_pid, 6);
10          consumeActions_t; transportActions_t; produceActions_t;
11          set_priority(_pid, 1) }
12    :: ...
13 od }
```

**Fig. 5** General structure of the PROMELA model for a NPN

tation; hence, the channel may contain multiple occurrences of the same token. An improved version may use an additional field for the multiplicity of the token or a C-data structure. As an alternative, a colored basic place may be unfolded into several uncolored ones.

A net-typed place is also represented using a buffered channel that stores the instantiation number of the processes corresponding to net tokens at the place. We translate a synchronization as a request–response communication. Therefore, we also use the asynchronous channel as the media where the net processes write their synchronization requests.[3] Each message in these channels consists of three fields: the `_pid` of the net token sending the message, and the label and identity of the enabled transition. We assume that $L \subseteq [1, 254]$ and each labeled transition has an identity number represented in a byte. The response messages are received via a global channel (`gbChan`) whose structure is explained in Sect. 3.3.3. All the channels are ordered using the *sorted send* (`!!`) to reduce the state space of the model.

A PN has been usually translated into PROMELA using a loop where each option simulates the firing of a transition [15, 16,20,64]. The firing is often enclosed in an `atomic` region to avoid the interleaving and has the following form: `atomic{enableTest_t -> consumeActions_t; produceActions_t}`. The expression `enableTest_t` is the conjunction of an enabling condition for each input arc $(p, t)$. This condition, here denoted as $\texttt{enableTest}_p$, depends on the arc inscription $W(p, t)$. Analogously, `consumeActions_t` is the sequence of instructions for removing the tokens on each input label $W(p, t)$. Finally, `produceActions_t` is the sequence of instructions for adding the tokens in $W(t, p)$ to each output place $p$.

In order to adapt the standard translation to the NPN framework we encode each transition $t$ in $SN$ as shown in Fig. 5, lines `10-13`. First note that a special code `transportActions` is added to distinguish the case when net tokens are transported. This code is not required for a basic token since it can be implemented in terms of `consumeActions` and `produceActions`. However, the removal of a net token entails the termination of the corresponding process. On the other hand, when producing a net token, a new process should be created with the initial marking of the element net. But when a child net is transported, its internal marking may not change or coincide with the initial one. In this section we do not deal with net copies, to keep the presentation simple. Therefore, we will assume that every non-anonymous net variable $x \in Var(W(p, t))$ has at most one occurrence in the output arcs of $t$. In Sect. 5.2, we illustrate how to adapt the translation for copying 1-level nets which are the most used in the literature.

Another difference in the usual translation is that each step of the NPN is only initiated if gbChan is empty. All child processes involved in the step are activated by a response message previously sent to `gbChan`. Therefore, the condition `empty(gbChan)` also ensures that no foreign firing is interleaved. As in [62], we use process priorities, but here they serve to keep track of the step progress during a simulation run. To this end, we define five levels of priorities from 6 to 2. The atomic region initiating the step is always executed with priority 6. The firing of transitions with labels in $L_v^-$ or $L_h$ is executed with priority 5 or 4 resp. This allows a clear distinction of a synchronization and its type. A process with priority 3 represents a net token that was transported or consumed without synchronization. Finally, a priority 2 is used for setting the initial marking of the new processes (net

[3] The rendezvous communication in PROMELA is not suitable for simulating the synchronizations in a NPN. A rendezvous statement can only be executed if a matching statement can be performed immediately; otherwise the process is blocked. This would prevent a net token from firing any other transition until the synchronization could be completed.

**Table 1** Consume and produce actions for basic-typed arc labels

| Arc label | consumeAction | produceAction |
|---|---|---|
| c | bp ?? c | bp !! c |
| x | bp ?* x | bp !! x |
| _ | bp ?* _ | if :: bp !! v1 ... :: bp !! vn fi |

tokens) created by the step. More details will be provided further in this section and in Sect. 4. As we will show in Sect. 6, this feature also helps to reduce the verification time.

### 3.3 Encoding the transitions in *SN*

#### *3.3.1 The enable test*

We assume that each incident arc to an uncolored place $up$ is labeled by a natural number. Hence, if $W(up, t) = n$ then $\texttt{enableTest}_{up}$ is the expression `up >= n`. An incident arc to a colored place is labeled by a multiset. Therefore, for a basic colored place $bp$, $\texttt{enableTest}_{bp}$ has the form `len(bp) >= n && E(c1) ... && E(cm)`, where `n` is the size of the multiset $W(bp, t)$ and `E(ci)` is an enabling expression for each basic constant in $W(bp, t)$. The expression `E(ci)` depends on the multiplicity ($k$) of the constant `ci`. When $k = 1$, the poll operator can be used to test the presence of a message matching a pattern at any position in the channel. In this case the pattern is the integer representation of the token. However, there is no predefined function in PROMELA for computing the number of such messages if $k > 1$. Hence, we implemented two functions `numMsg` and `numTok`, using SPIN facility for embedding C code into the PROMELA models. The definition and further details of the C function `numMsg` appear in "Appendix 1." The function `numTok` is defined analogously. By means of the latter, the expression `E(ci)` is defined as

$$\begin{cases} \texttt{bp ??[ci]} & \text{if } k = 1 \\ \texttt{c\_expr\{ numTok(qptr(PProcName -> bp - 1), ci) >= k \}} & \text{if } k > 1 \end{cases}$$

For a net place $np$, the multiset $W(np, t)$ consists of distinct variables. Hence, $\texttt{enableTest}_{np}$ must check the existence of at least $k = |W(np, t)|$ request messages at the channel, with the complementary label of $t$. It is defined as follows

$$\begin{cases} \texttt{np ??[\_, -Lv + (t), \_]} & \text{if } k = 1 \\ \texttt{c\_expr\{ numMsg(qptr(PProcName -> np - 1), -Lv + (t)) >= k \}} & \text{if } k > 1 \end{cases}$$

Here, we use `Lv+(t)` to indicate a label for lower synchronization and `-Lv+(t)` for its complementary. We extend this notation for transitions with labels in $L_v^- \cup L_h \cup \{\epsilon\}$, by defining `Lv+(t)=0` and `-Lv+(t)=MaxL` where `MaxL` is a fixed integer constant not belonging to $L$ (e.g., 255). Note that a net token $nt$ may be involved in the binding of $t$ only if a request message with the process identity of $nt$ was previously written at a channel.

#### *3.3.2 The firing instructions dealing with basic places*

For an uncolored place $up$, if $W(up, t) = n$, $\texttt{consumeActions}_{up}$ is the instruction `up = up - n`. If $W(t, up) = n$, then $\texttt{produceActions}_{up}$ is defined as `up = up + n`. For a colored place $p$, $\texttt{consumeActions}_{p}$ is a sequence of instructions for removing from $p$ the tokens in $b(W(p, t))$. Similarly, $\texttt{produceActions}_{p}$ is the sequence of instructions for adding the tokens in $b(W(t, p))$ to $p$. Table 1 shows these PROMELA instructions for basic-typed places. In order to consume a basic constant we use the random receive (`??`) that removes the first message matching a pattern, regardless of its position in the channel. The sorted send is used to produce a basic constant.

Without loss of generality we assume that all constants appear first in $W(p, t)$ since it makes easier the computation of the binding. We also assume that the name of an arc variable and the corresponding PROMELA variable are the same. Since in our translation the channels represent multisets, the token or message to bind a variable should be non-deterministically chosen. For the sake of readability, we use a notation similar to the standard SPIN statement for receiving messages, i.e., `chan ?* msg` (see "Appendix 1" for the inline definition in case of a net channel). The last row of the table allows the use of anonymous variables in input and output arcs. PROMELA also permits anonymous variables, and thus `consumeAction` is defined analogously. For the `produceAction` we assume that `v1...vn` are all the values belonging to the type of the output place. The semantics of the `if` construction ensures that one of the values will be randomly chosen in simulations, whereas all of them will be explored during verifications.

#### *3.3.3 The firing instructions dealing with net-typed places*

Table 2 shows the PROMELA instructions for dealing with net-typed arc inscriptions. For a net place $np$, the label of an output arc $W(t, np)$ includes just net constants or non-anonymous variables. In the latter case, due to the restriction on output arcs introduced in Sect. 3.2, the net tokens must be transported. In the former case, the instructions in

**Table 2** Consume, produce and transport actions for net-typed arc labels

| | |
|---|---|
| `consumeAction(_)` | `np ?* nt,-Lv+(t),_;`<br>`consNetTok(np,nt);`<br>`if :: Lv+(t)>0 -> pr_nt = 5`<br>`:: else -> pr_nt = 3`<br>`fi`<br>`gbChan !! 6-pr_nt,nt,-Lv+(t),np,1;`<br>`set_priority(nt, pr_nt);` |
| `transportAction(x)`<br>`np != onp` | `np ?* nt,-Lv+(t),_;`<br>`transpNetTok(np,onp,nt);`<br>`if :: Lv+(t)>0 -> pr_nt = 5`<br>`:: else -> pr_nt = 3; onp !! nt,MaxL,0`<br>`fi`<br>`gbChan !! 6-pr_nt,nt,-Lv+(t),onp,0;`<br>`set_priority(nt, pr_nt);` |
| `produceAction(EN_i)` | `nt = run EN_i();`<br>`np ! nt,MaxL,0;`<br>`gbChan ! 6-2,nt,MaxL,np,0;`<br>`set_priority(nt, 2);` |

`produceActions`$_{np}$ must create a process for each net constant. As shown in the last row, a child net token *nt* is produced at a place *np* by creating an instance of the corresponding `proctype` definition. The initial message sent to the channel (`nt,MaxL,0`) represents the net token at the place and stores the process instantiation number. The auxiliary label `MaxL` allows the removal of *nt* without synchronization, i.e., by an unlabeled transition or when the parent net is consumed. The last field of this message can be used to encode the type of the net token. However, we omit this information since by assumption each arc variable has the same type of its incident place. Note that a response message is sent to `gbChan` and the priority of the new process is changed to 2. This ensures that, by the end of the step, the values of the new local variables will match the initial marking of the element net.

As we explained in Sect. 3.3.1, a transition *t* with an arc (*np*, *t*) is enabled to consume or transport *nt* if the channel for *np* contains a message with the `_pid` of the process corresponding to *nt* and the complementary label of *t*. If *t* fires and consumes *nt*, then the code for `consumeActions`$_{np}$ removes the remaining request messages of *nt* from the channel (`consNetTok`). If *nt* is transported to a new[4] place *onp*, then `transportActions`$_{np}$ moves the messages to the new channel (`transpNetTok`). Besides, a response message is sent to the *nt* process via `gbChan` consisting of a precedence (between 1 and 4), the `_pid` of the receiver (*nt*), the synchronization label, the (perhaps new) parent place channel and a bit indicating whether or not *nt* was consumed. In addition, the priority of the *nt* process is changed. As we mentioned above, this improves the readability of the simulation traces and the performance during verification. The responses are ordered in `gbChan` by the precedence (computed using the new priority of the receiver) and the instantiation number. These two first fields of the message define the execution order of the processes involved in the step, once *t* completes the firing. The ordering is not relevant inside a synchronization, but it helps to reduce the state space of the model.

[4] In case $np = onp$, `transpNetTok` is not required. Besides, if *t* is unlabeled the entire code can be omitted. Also note that net-typed variables do not need a representation in the translation.

### 3.4 Translating the element nets

Each element net is translated into a `proctype` definition, as sketched in Fig. 6. The initial marking of the net token is assigned after receiving the first message from the parent net (last row in Table 2). The message contains the channel (place) where the token was created. However, if the initial marking of the element net can be encoded in the variable declaration, then the parent channel should be declared as an input parameter. This way, there is no need to write the initial response message at `gbChan` and the net token can be created with priority 1. We assume that this translation is always applied for the element nets $EN_1, \ldots, EN_b$.

The behavior of a child net is also simulated using a `do`-loop, but the options have different forms, depending on the labels of the transitions. The translation of a transition *t* with

**Fig. 6** PROMELA encoding for an element net

```
1  proctype EN_i(){
2  chan ppChan;  /* Non-Shared Places and Basic Typed Arc Variables */
3  atomic{ gbChan ? 6-2,eval(_pid),MaxL,ppChan,0;
4          /* Initial Marking */;
5          set_priority(_pid, 1) }
6  do :: atomic{/* autonomous firing          - O1 */}
7     :: atomic{/* synchronization request    - O2 */}
8     :: atomic{/* horizontal synchronization - O3 */}
9     :: ...
10    :: atomic{ gbChan ? _,eval(_pid),lt,ppChan,rm  ->
11        if :: lt == L(t) && enableTest_t ->    /* sync firing */
12              consumeActions_t; transportActions_t; produceActions_t;
13              rmConf_t;
14           :: ...
15           :: lt == MaxL -> skip
16        fi;
17        if :: rm   -> consNetsAtPlace(np1); ...; break
18           :: else -> set_priority(_pid,1)
19        fi }
20 od }
```

label in $L_v^+ \cup \{\epsilon\}$ is similar to the one provided for the transitions in *SN* (cf. `O1` in Table 3). But when $t$ is in conflict with another transition with label in $L_h \cup L_v^-$, the code should include an additional instruction `rmConf_t` that we explain further in this section. In this paper, we have avoided a nested loop and the `unless` construction proposed in [61,62]. This way, we preserve to atomicity of the autonomous firings that create net tokens and reduce size of the state space.

The transitions labeled for upper or horizontal synchronization depend on other net tokens to fire. Therefore, its translation is divided into two parts. The first part is performed once the transition is enabled and it simply writes a request message at the parent channel (`O2` in Table 3). The second part (the firing) is performed once the process receives the response message, as shown in Fig. 6, lines `10-19`. At this point, any enabled transition with the same label may be chosen for firing. The empty option in line `15` corresponds to a transportation or removal without synchronization. If the net was consumed by the parent, then the child nets that may still be active at some place are removed without synchronization (line `17`). After that, the loop and the process terminate.

A horizontal synchronization can be arranged by any of the possible participants, since all of them are situated at the same place and will remain there after the firing. Therefore, the loop also includes an option for each horizontal label `Lh` occurring in the element net (see `O3` in Table 3). This option checks if the number of requests at `ppChan` is greater than or equal to the arity of the label. If so, the remaining participants are non-deterministically chosen and the response messages are sent. Each process involved in the horizontal synchronization executes the firing with priority 4. The translation is easily adapted in case the arities are attached to places instead of labels. In this respect, a local variable should be added to the `proctype` definition for holding the arity of the place where the net token is hosted. As the parent channel, it should be updated whenever the net token is created or transported to a new place.

We remark that the rules for generating the `enableTest`, `consumeActions` and `produceActions` are the same for all transitions, regardless of the label. This is because when a transition $t1$ with label in $L_h \cup L_v^-$ consumes or transports net tokens, the label used for `enable` and `consume` is the one for lower synchronization, i.e., `Lv+(t1)=0` as for unlabeled transitions. As we mentioned above, for these transitions an additional situation should be considered, i.e., when they may have a conflict with another transition, say $t$. A conflict occurs if there is a reachable marking such that both transitions are enabled and the firing of $t$ disables $t1$. In this case, if there is a synchronizing request from $t1$ at the parent channel, then the message should be removed after the firing of $t$. Note that, transitions $t$ and $t1$ must belong to

**Table 3** Options in the loop of an element net

| | |
|---|---|
| 01 | `empty(gbChan) && enableTest_t ->`<br>`set_priority(_pid,6);`<br>`consumeActions_t; transportActions_t; produceActions_t; rmConf_t;`<br>`set_priority(_pid,1);` |
| 02 | `empty(gbChan) && enableTest_t && ! ppChan??[eval(_pid),L(t),_] ->`<br>`ppChan !! _pid,L(t),t` |
| 03 | `empty(gbChan) && ppChan??[eval(_pid),Lh,_] &&`<br>`c_expr { numMsg(qptr(PEN_i->ppChan-1), Lh) >= ar(Lh) } ->`<br>`set_priority(_pid,6);`<br>`ppChan ?? _pid,Lh,_;`<br>`ppChan ?* nt,Lh,_; gbChan !! 6-4,nt,Lh,ppChan,0; set_priority(nt,4);`<br>`/* repeat the above line ar(Lh)-2 times */`<br>`gbChan !! 6-4,_pid,Lh,ppChan,0; set_priority(_pid,4);` |

the same net token because $t1$ has no shared place as input. Hence, to deal with such a situation, each transition $t$ with a common input place (or output place if, e.g., inhibitor arcs[5] are allowed) with $t1$ must include the next code as part of `rmConf_t`.

```
if :: ppChan ?? [eval(_pid),_,t1]
 -> ppChan ?? eval(_pid),_,t1
  :: else fi
```

For simplicity, this code just removes the request message without checking whether or not $t1$ was disabled. The request can be rewritten later, in case $t1$ (or any other transition with the same label) remains enabled after firing $t$.

We point out that this is the only purpose of the field for the identity of the transition. Therefore, it is not required when the NPN has no conflict involving these labels or if the labels have a single occurrence in each element net. In addition, we can get rid of this field if in the above code we check and remove the message with the label of $t1$, regardless of the transition that sent the request. Other subtle refinements may also help to reduce the size of model and the verification time. For example, the `enableTest` in Fig. 6, line `11` can be omitted if the label occurs only once in the element net. If no net token is transported, then the channel and bit fields in the response messages can be omitted. Tight bounds for the channels and data types such as `unsigned` and `bit` instead of `byte` for representing places and labels are also helpful for reducing the state space. Some of these refinements have been applied in the translation of the NPN of Fig. 2 that is included in "Appendix 2."

[5] An inhibitor arc is enabled when the input place is empty.

## 4 Outline for the correctness proof

In this section we provide the arguments for proving the correctness of the above translation schema. Due to the length of a formal proof, the discussion we present here is mostly informal. We refer the reader interested in the formal semantics of PROMELA to [14,46,51,63]. In particular, the work in [51] includes a formal correctness proof of a transformation from object-based graph grammars into the language. The translation deals with processes and asynchronous messages but does not allow object deletion or synchronization. Nevertheless, their proof may serve as the basis for the formal proof for our translation.

We start our informal discussion by noting that all net components, basic tokens, labels, variables and places are properly encoded as PROMELA `proctype` definitions, integer constants and integer or channel variables. Each net token is represented by an active process and a specific message `[_pid,MaxL]` at a channel corresponding to the place where it is situated. Each transition is represented by an option in the loop (or the inner conditional) with the code implementing its firing. The input/output arcs and their inscriptions are embedded in the firing code.

Roughly speaking, the state of a PROMELA program holds the information about the active process instances, the channels and the values of the global variables. The state of each process includes the values of the local variables and its unexecuted code. For our translation, each active process represents either *SN* or a net token; thus from some states of the PROMELA program, a marking of the NPN can be constructed.

The basic PROMELA statements in our translation are all enclosed in atomic regions. The first region executed by the program (hereafter denoted as $A_0$) is the one before the

loop in the `init` process (Fig. 5, lines `9-11`). Its aim is to initialize the system net using just basic constants or 1-level nets of type $EN_1, \ldots, EN_b$. It may be empty if the initialization can be performed as part of the variable declaration. In any case, after completing $A_0$, the PROMELA program will be in a state such that: 1) `gbChan` is empty and 2) all active processes are properly initialized/marked and have priority 1. We can define these states as *valid* or *well-formed states* of the PROMELA model. It can be shown that from these states, we can always construct a marking of the NPN by discarding the request messages at the channels. In particular we can prove that after $A_0$, the underlying marking of the program matches the initial marking $I_0$ of the NPN. Furthermore, if a transition (in *SN* or an initial net token) is enabled in $I_0$ then an option in the loop corresponding to this transition (in `init` or an active process) will also be enabled after $A_0$. Depending on the label of the transition, the option may be either the encoding of a firing ($l \in L_v^+ \cup \{\epsilon\}$) or a synchronization request ($l \in L_h \cup L_v^-$).

A step $I_0[\rangle M_1$ in the NPN will correspond to a sequence $A_1$ of atomic regions $A_{10}A_{11}\ldots A_{1i}$ ($i \geq 0$) that are executed with priorities greater than 1. The first region of every step is always (and the only) executed with priority 6. It corresponds to the firing of a transition with label in $L_v^+ \cup \{\epsilon\}$ or an arrangement for horizontal synchronization. In a vertical step, the sequence continues with priority 5 `atomic` regions, corresponding to the firing of the upper transitions ($l \in L_v^-$) involved in the synchronization. In a horizontal step, the firing of each labeled transition ($l \in L_h$) is executed with priority 4. We have chosen a different priority for these transitions to easily recognize the type of synchronization during simulations.[6] The sequence may continue with a number of regions with priority 3, corresponding to net tokens that are transported or consumed without synchronization. It may finish with regions with priority 2 that initialize the new processes (net tokens) created in the step. The execution of all these processes with priorities from 5 to 2 is due to response messages written along the sequence. By the end of $A_1$, `gbChan` will be empty and any process not consumed by the step will continue in the loop with priority 1. The latter holds because all the processes involved in $A_1$ finish its execution either with a statement `set_priority(_pid,1)` or a `break` that terminates the loop (and the process). Therefore, the sequence $A_1$ leads to a valid state.

The sequence $A_1$ may be preceded by a finite number of regions $R_{11}\ldots R_{1m}$ ($m \geq 0$) with priority 1, corresponding to synchronization requests. Note that $m$ is greater than zero if the first step of the NPN is a synchronization. Any request is executed once before the synchronization, unless it is removed by another firing (`rmConf_t`). Besides, at each channel there will be a single request message per net token and label (see the guard of `O2` in Table 3). These regions modify the state of the PROMELA program, in particular the content of the channels, but do not affect the underlying marking of the NPN. The same applies to other PROMELA steps for the control flow of the program.

By choosing the appropriate processes, options in the loops and messages in the channels, the state of the program after completing the sequence $R_{11}, \ldots R_{1m}, A_{10}, A_{11}, \ldots A_{1i}$ will match the marking $M_1$. Clearly, several execution sequences may correspond to the same step in the NPN. Note that, e.g., the number and order of the requests may vary and the arrangement of a horizontal synchronization can be done by any of the participants. Since concurrent processes having the same priority (in this case 1) are interleaved in all possible ways by SPIN, it is guaranteed that all possible steps $I_0[\rangle M_1$ can be reproduced by the translation. A further step $M_1[\rangle M_2$ will also correspond to a sequence $R_{21}\ldots R_{2n}A_{20}A_{21}\ldots A_{2j}$ with $n \geq 0$, $j \geq 0$. Here we stress that the request message of an enabled transition may have been sent in a previous step; thus the case $n = 0$ and $j = 0$ is possible.

Using induction on the length of a sequence $I_0[*\rangle M_k$, we may conclude that any finite firing sequence in the NPN can be simulated using the PROMELA translation, assuming an infinite number of processes and unbounded channels. Using structural induction we can prove that, under this assumption, the translation allows simulating an infinite sequence as well. In addition, we can define the *valid* or *well-formed paths* of the model as those ending at a valid state. It can be shown that these paths have the form

$$A_0 R_{11}\ldots R_{1m} \underbrace{A_{10}\ldots A_{1i}}_{A_1} R_{21}\ldots R_{2n} \underbrace{A_{20}\ldots A_{2j}}_{A_2}$$
$$R_{31}\ldots R_{3p} \underbrace{A_{30}\ldots A_{3q}}_{A_3} R_{41}\ldots$$

as explained above. The path may end after a step or a request. Therefore, from a valid path in the PROMELA translation, we can construct a firing sequence in the NPN. The markings involved in the firing sequence are constructed from the final state of each sequence $A_k$. To see more clearly the transitions involved in the firing sequence, we may identify each transition in the NPN with a unique integer. Then, in the PROMELA program we may use a global variable to store the identity transition number at the beginning of each firing encoding. This way, the variable will keep track of all transitions that fire along the execution path.

## 5 Detecting faults in a NPN-based model using SPIN

PROMELA models are intended to specify finite states transition systems. Therefore, the data types are restricted and

[6] Any sequence $A_k$ may contain regions with priorities 4 or 5, not both.

the size of channels and the number of active processes are limited to 255. As a result, a firing sequence exceeding these bounds cannot be reproduced by the translation. In general, what we can conclude is that for any firing sequence in the NPN there is an execution path in the PROMELA model corresponding to a prefix of the sequence. Conversely, from an execution path of the PROMELA program we can always construct a finite firing sequence of the NPN. Using this fact, we may discover some sequences violating properties related to termination, reachability and boundedness of NPN. A counterexample may be found quickly with SPIN even using a basic verification. It may end in an invalid state, i.e., with an incomplete `atomic` region. But the firing sequence can be reproduced till the last valid state.

In this section we explain how to detect the next four types of faults in a NPN-based model using SPIN.

1. *Existence of infinite recursive sequences*. The dead markings of a NPN correspond to the states of the PROMELA model in which every active process is blocked at the loop. These states can be marked as valid `end` states for SPIN by adding an `end` label in front of the loops. This way, an infinite recursive sequence will lead to an invalid state (in the middle of an atomic region), because the number of active processes or channels will be exceeded. Actually, any long sequence going beyond SPIN limits will produce an invalid state. The invalid states can be found using SPIN safety verification. We can get insights about the fault (whether or not it is due to an infinite recursion) using SPIN guided simulation or advanced verification options.
2. *Violations of place invariants.* Invariants for places can be translated as assertions on the corresponding variables. Auxiliary variables should be used to simplify the assertions on net-typed places. Note that the number of net tokens at a place may not be equal to the length of the channel because the channel may also contain several request messages. These auxiliary variables should be updated each time the marking of the place is modified by the firing of a transition.
   An assertion can also be used to enforce the conditions for executing a firing. Such conditions must prevent errors such as send operations on full channels or assignments of values outside the domain. This is important because these errors are not detected by SPIN during verifications. A send operation on a full buffer either blocks the process or loses the message (depending on the SPIN option `-m`). Besides, an integer overflow is only reported during simulations because SPIN automatically truncates the result of an expression to the domain. Therefore, these assertions as well as the place invariant should be inserted at the beginning of each option of interest. The violations are also found using the safety verification.

It is important to note that when a safety verification is completed without errors, the verifier may report some unreachable states in the PROMELA model. One of these states is the end of the `init` process because the system net is never be consumed. The remaining unreachable states may be due to net tokens that were not removed or transitions that never fired. This information may help to detect errors or refine the model.

3. *Existence of infinite cycles.* An infinite cycle in the NPN will lead to a cycle in the state space of the PROMELA model. These cycles can be detected using SPIN search for acceptance cycles. In this case, the verifier will report any infinite run that visits an acceptance state infinitely often. A state of a PROMELA model is an acceptance state if any of the active processes is at a statement labeled with an `accept` label. For our translation, it is enough to add an `accept` label in front of the loop of `init` process. Since *SN* is never consumed, this will mark the beginning of any step.
4. *Violations of LTL properties.* Properties of a NPN related to reachability can be encoded as LTL formulas or `never` claims in the PROMELA model. To this end, the involved places of *SN* should be declared as global variables. Conditions involving the marking of net tokens should be specified as a `never` claim, in order to access the local places through remote references. This is only possible if the total number of net tokens is known in advance.

We remark that the properties of a NPN with infinite cycles may be hard to verify with SPIN. This is due to the fact that SPIN provides support just for weak fairness at the process level. Hence, a firing sequence where a net token fires infinitely often but it is also infinitely often disabled may not be covered by the verification. Furthermore, no fairness is applied to non-deterministic statements. Therefore, some liveness properties of a PN may not be directly verified with SPIN (see, e.g., [53]). The strong fairness may be encoded inside the model and the claim. But this is not easily achieved for an arbitrary NPN because it may have an arbitrary number of net tokens.

### 5.1 Analyzing the running example with SPIN

In this section we analyze the NPN of our running example using the above approach. Our goal is to answer the question raised at the end of Example 3, i.e., if, from the initial marking in Fig. 3, all agents return to the initial place after completing all activities and tasks. First, we performed a basic safety verification of the model to prove the absence of infinite sequences in the NPN. This search was completed in 4.02s using SPIN version 6.4.3 on a notebook Intel Core I3,

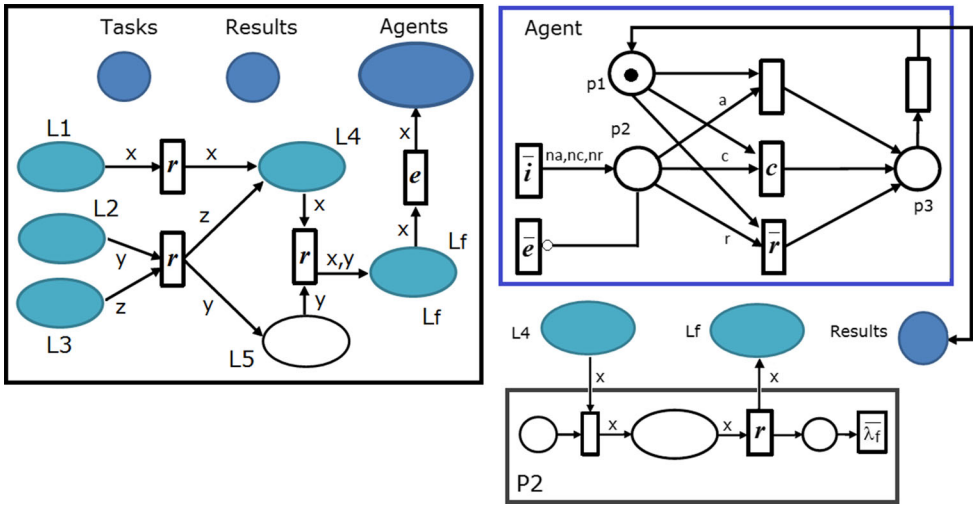



**Fig. 7** Improvements on the NPN in Fig. 2

2.4GHz, 4Gb RAM. The previous question can be specified in PROMELA using the next LTL formula:

```
ltl p {<>[]( Tasks==0 &&  pOut==1 &&  len(Agents)==3 &&
len(L1)==0 &&  len(L2)==0 && len(L3)==0 &&
len(L4)==0 && len(L5)==0  && len(Lf)==0 ) }
```

Note that, the condition `len(Agents)==3` does not guarantee that the three net tokens are situated at *Agents*. Therefore, we may ensure that the remaining places that may host the agent nets are empty. The symbols <> and `[]` represent the temporal operators *eventually* and *always* in PROMELA resp. SPIN found a violation for this property in 0.01s. The last part of the error trail with the final state of the global variables is shown below. The labels are encoded using integers as follows: $\bar{i} = 1, \bar{r} = 3, c = 5$ and `MaxL`=10.

```
spin: trail ends after 852 steps
#processes: 4
       queue 3 (gbChan):
       Tasks = 1
       Results = 8
       queue 2 (Agents): [2,1][2,5][2,10][3,1][3,5][3,10]
       queue 7 (L1):
       queue 8 (L2):
       queue 9 (L3):
       queue 12 (L4): [1,1][1,3][1,5][1,10]
       queue 14 (Lf):
       queue 13 (L5):
       pOut = 1
852:    proc  3 (agentNet:5) exNPNMAS.pml:116 (state 141)
852:    proc  2 (agentNet:1) exNPNMAS.pml:116 (state 141)
852:    proc  1 (agentNet:1) exNPNMAS.pml:116 (state 141)
852:    proc  0 (:init::1) exNPNMAS.pml:253 (state 348)
852:    proc  - (p:1) _spin_nvr.tmp:7 (state 10)
6 processes created
```

The counterexample shows a sequence ending in a dead marking with an agent stuck at $L4$. This `agentNet` token, represented by the message `[1,10]`, has all labeled transitions enabled; therefore, it could not perform at two different activities ($r$ and $c$). The trail also reveals that the other agents return home without completing the activities intended for collaboration. Note that the channel for the *Agents* place has request messages for horizontal synchronization (`[2,5],[3,5]`). Besides, `Tasks = 1` and `pOut = 1` indicate that the minimum number of results ($k = 5$) should be increased in order to accomplish all tasks.

The above problems can be solved by adding a new element net $P2$ that moves an agent from $L4$ to $Lf$ and forces it to perform a request task (see Fig. 7). A net token of $P2$ is created along with the recursive call to $P1$; thus, the input arc of the place $pR$ in $P1$ should be labeled as $P1, P2$. We also added a sink transition to the *Agent* element net, labeled for vertical synchronization ($\bar{e}$). This transition is linked to $p2$ by an inhibitor arc (represented using a circle instead of an arrow on the transition side). Now, the last transition in *SN* is labeled as $e$, ensuring that the agent returns home after completing all the activities. Finally, we define $k = 15$.

In the improved model,[7] we used a simpler LTL formula for the same property: `<> [](Tasks==0 && pOut==1 && netsAtAgents==3)`. Here, the auxiliary variable `netsAtAgents` keeps track of the number of net tokens at *Agents*. The verification of the property took 112s using the extra option `-DCOLLAPSE`. The last part of the verification report is shown below.

```
unreached in proctype agentNet
       exNPNMASimpExt.pml:134, state 5, "set_priority(_pid, 6)"
       exNPNMASimpExt.pml:135, state 6, "p1 = (p1-1)"
       exNPNMASimpExt.pml:55, state 9, "pc??eval(_pid),5"
       exNPNMASimpExt.pml:55, state 13, "(pc??[eval(_pid),5])"
       exNPNMASimpExt.pml:55, state 13, "else"
       exNPNMASimpExt.pml:55, state 15, "pc??eval(_pid),3"
       exNPNMASimpExt.pml:136, state 21, "set_priority(_pid, 1)"
       exNPNMASimpExt.pml:176, state 129, "-end-"
```

[7] See http://www.dropbox.com/s/uloj2xmgwomimqv/exNPNMASimpExt.pml?dl=0.

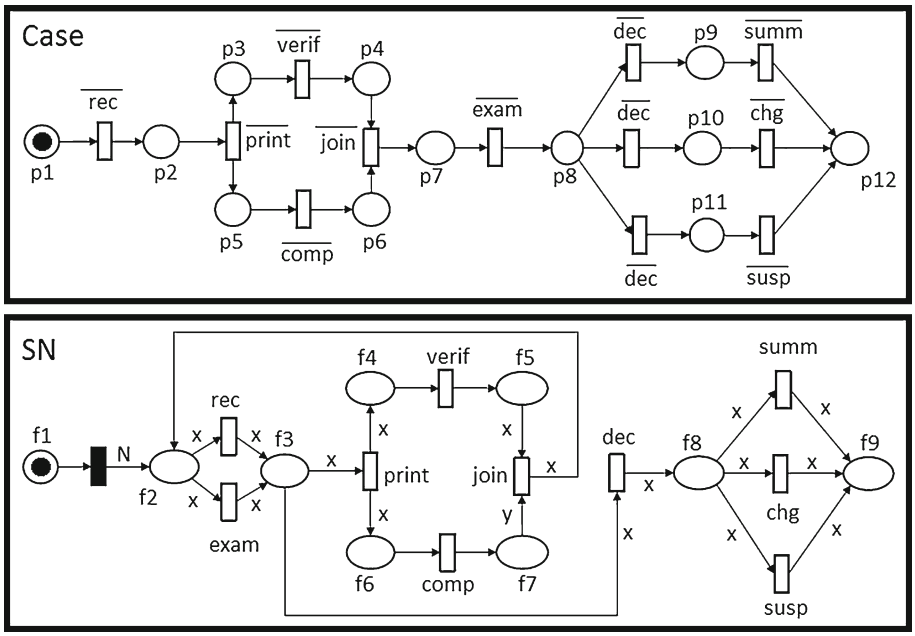



**Fig. 8** A NPN variant of the prosecution Example from [12,59]

**Fig. 9** PROMELA code for copying a net token (transition *print* in Fig. 8)

```
1  chan copyMsg = [MaxNumCopies] of {byte, chan }; chan copyPCh;
2  proctype case(chan pc; bit p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12){
3  do  ...
4   if :: rm   -> break
5      :: else -> do ::copyMsg ?? [eval(_pid),_] ->      /* copy */
6                       copyMsg ?? eval(_pid),copyPCh;
7                       nt = run case(copyPCh,p1,p2,...,p11,p12);
8                       copyPCh ! nt,15;
9                     ::else -> break
10                 od;
11                 set_priority(_pid, 1)
12   fi  }
13 od }
14 NetPlace(f2); NetPlace(f3);  ... ; NetPlace(f9);
15 init{ bit f1=1;
16 do ...
17 :: atomic{ empty(gbChan) && f3 ?? [_,2] ->          /* print */
18       set_priority(_pid, 6);
19       recMsg(f3, nt,2); transpNetTok(f3,f4,nt);      /* transport x */
20       gbChan !! 6-5, nt,2,f4,0; set_priority(nt, 5);
21       copyMsg !! nt,f6;                              /* copy x */
22       set_priority(_pid, 1) }
23 od }
```

```
       (7 of 129 states)
unreached in proctype P1
       exNPNMASimpExt.pml:282, state 504, "(1)"
       exNPNMASimpExt.pml:284, state 507, "set_priority(_pid, 1)"
       (2 of 512 states)
unreached in proctype P2
       (0 of 110 states)
unreached in init
       exNPNMASimpExt.pml:415, state 350, "-end-"
       (1 of 350 states)
unreached in claim p
       _spin_nvr.tmp:10, state 13, "-end-"
       (1 of 13 states)
pan: elapsed time 112 seconds
pan: rate 68279.609 states/second
```

The verifier found some unreachable states in the `proctype` definition of `agentNet`. The states `5,6,9, 13,15,21` are all related to the transition $u$ that never fired. The last unreachable state (`129`) was reported for the end of the `proctype` definition (as for `init`—state `350`) because no `agentNet` token was consumed. The unreachable states in $P2$ are due to the fact that its net tokens were not transported. They are not reported for $P2$ because the code was refined as explained in Sect. 3.4.

### 5.2 Dealing with copies of a net token

In this section we use an example from [59] due to W.M.P. van der Aalst, the *workflow of the Dutch Justice Department*. In [12], it was modeled using an object net with reference semantics and verified using XTL and Maude. Here, we use the NPN approach as depicted in Fig. 8: The element net models a case and the system net its life cycle. In $SN$, the black bar represents an unlabeled transition and $N$ is the initial number of $Case$ net tokens. Instead of synchronous channels as in [12], we use labels attached to the transitions. The labels (written outside the transitions) represent the internal tasks to be performed for each case. We briefly describe them as follows:

- *rec*: the case is recorded by an official;
- *print*: two copies of the case are printed and sent to verify (*verif*) and complete (*comp*). In [12], the places contain references to the net tokens and this transition creates a new reference of the input net token. Here, a new net token is created which is a copy of the input net after the firing;
- *join*: puts together the information obtained from *verif* and *comp*. In [12], the same variable (net token reference) is used on both input arcs. But this is forbidden in a NPN (Definition 1, condition 8c). Hence, here this transition has two different net tokens as input;
- *exam*: the case is examined and a prosecutor decides (*dec*) whether to summon (*summ*), charge (*chg*) or suspend (*susp*).

The PROMELA translation of Sect. 3 can be easily extended to deal with copies of 1-level nets. To this end, an additional global channel should be used for holding the instantiation number of the net token to be copied and the parent channel where the new instance should be situated. The code for sending these messages should be inserted after transporting the original child net (see Fig. 9, line `23`). The code for receiving the messages should be executed once the transportation is completed. Therefore, the `proctype` definition of an element net that may be copied requires two modifications. First, we add a `do`-loop to the `else` branch of the last conditional statement, as in Fig. 9, lines `6-11`. The new loop is intended for reading the copy messages and producing the new instances at the corresponding channels. Besides, the `proctype` declaration should use parameters instead of local variables for the places. Since the basic places can be represented using integer variables, the response message and the priority change are not required for initializing the copy. See "Appendix 3" for further details.

A safety verification of the model with $N = 1$ was completed in 0.044s, but several unreachable states were reported. To analyze the source of a possible flaw, we verified the proper termination of the system net by means of the LTL property `<> [](len(f9)==1)`. An error trail was found by SPIN in 0.021s. The last part of the trail is shown below, omitting the empty channels. Note that a new *case* process instance was created with `_pid = 2`. The messages `[1,15]` and `[2,15]` indicate that the net tokens are stuck at $f5$ (the original case) and $f7$ (the copy), resp. Besides, the net at $f5$ has the transition labeled as $\overline{comp}$ enabled while in its copy, the enabled transition is $\overline{verif}$ (request messages `[1,4]` and `[2,3]`, resp.).

```
spin: trail ends after 172 steps
#processes: 3
       queue 9 (f5): [1,4][1,15]
       queue 10 (f7): [2,3][2,15]
172:    proc  2 (case:1) prosecutionEx.pml:118 (state 90)
172:    proc  1 (case:1) prosecutionEx.pml:118 (state 90)
172:    proc  0 (:init::1) prosecutionEx.pml:174 (state 532)
172:    proc  - (p:1) _spin_nvr.tmp:7 (state 10)
3 processes created
```

The flaw is due to the fact that the example was not properly adapted to the NPN setting. Since the *verif* and *comp* tasks are assigned to different subnets, a net token cannot execute both. Therefore, as pointed out in [40], the branching in element net must be replaced by a choice, e.g., by joining the places as depicted below. With this modification the property is verified in 0.023$s$. The proper termination was also verified for $N = 2$ in 0.454s. Nevertheless, advanced options should be used for a larger $N$; for example, in case $N = 5$ the search was completed using bitstate hashing in 9.01s.

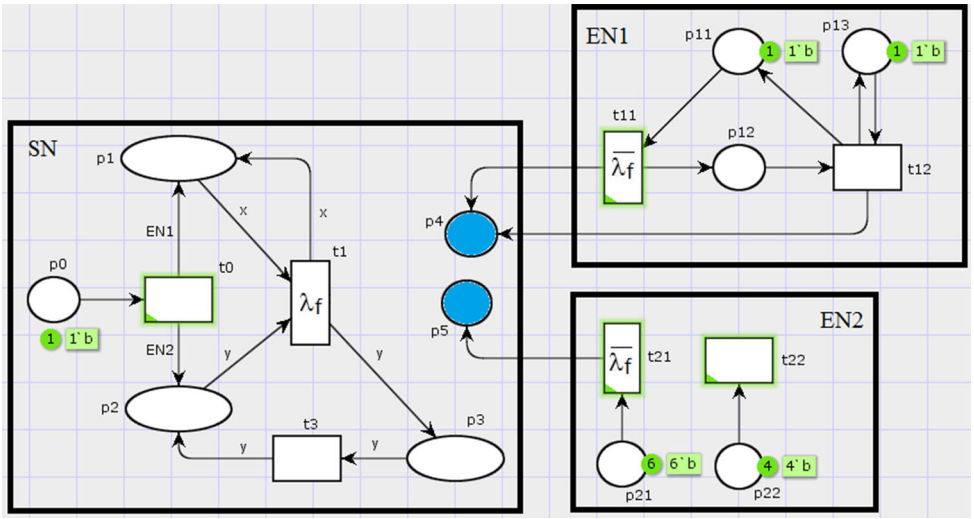
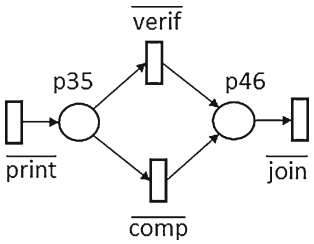



**Fig. 10** A NPN mostly drawn using the CPN Tools editor




In [12], a similar property `<>(len(f9)==5)` was verified for the object net using XTL in 115.65s (with a precompiled program) and Maude in 218.61s. Using SPIN the formula is verified in 6.96s but again using bitstate hashing with a small hash factor (1.84177). This indicates that the search coverage was not high. However, the state space of the NPN is larger than the one of the object net because the former reaches a marking with 10 `case` net tokens. Besides, the *join* transition in *SN* may combine an original case with any enabled copy at $f7$. Nonetheless, a counterexample for an invalid property with $N = 5$, `[](len(f9)<5)`, was found in 0.016s and depth 2383 using the basic search. A similar trace was found for the object net with XTL in 0.091s.

### 5.3 SPIN versus CPN Tools?

A two-level NPN without horizontal synchronization can be translated into a CPN using the approach in [9]. This translation spreads out the structure of the element nets all over the system net. To this end, the net tokens are encoded as ML integer lists, the autonomous transitions are moved outward and the vertical steps are flattened. The enable tests and the firings in the net tokens are moved into the guards and actions of the new transitions, resp.

Figures 10 and 11 (adapted from [9]) illustrate the transformation. We have used the CPN Tools editor to draw a simple NPN. Therefore, in Fig. 10, ellipses represent net-typed places and the blue circles are shared uncolored places. The black dot is denoted as the letter `b` and the green labels define the initial marking of each net component. Note that, the places at *SN* coincide with the places in the CPN in Fig. 11, but the net-typed ones are transformed into places of `INTlist` type.

An unlabeled transition in an element net $EN_i$ may be encoded using several loop transitions in the CPN (c.f. $t22$ in $EN2$ and transitions $t22p2$ and $t22p3$ in Fig. 11). This is because an autonomous step may occur at any place of *SN* typed as $EN_i$. In order to keep the example simple, we have used a single labeled transition in each net component. These transitions ($t1$, $t11$ and $t12$) are combined into a single transition $t1$ in the CPN. However, if an element net has several transitions with the same label, then the CPN should include a transition for each possible combination in a vertical step. The guards of the transitions in the CPN use the ML function `leq` for the enable test of the net tokens. The firing of a net token is embedded in the action of the transition using the ML function `sum`.

CPN Tools and SPIN can be used to analyze this kind of NPN.[8] The two reports cannot be compared since the state spaces are different: the reachability graph of the CPN and the state space of the PROMELA program. For the above example, CPN Tools (version 4.0.1) reported 126 nodes and 251 arcs while SPIN generated 851 states and 2142 transitions. Both reports were obtained in less than a second. CPN Tools provided information such as the number of home/dead markings, live/dead transitions and the bounds for the places; e.g., the integer upper bound for $p4$ and $p5$ is 6 and 12, resp. SPIN also computes the bounds for the integer variables of each `proctype` using the `-DVAR_RANGES` option. But more importantly we verified the validity of the LTL for-

[8] See http://www.dropbox.com/s/28j8x7fhyx7ucdg/npn2cpn.rar?dl=0 for the models.

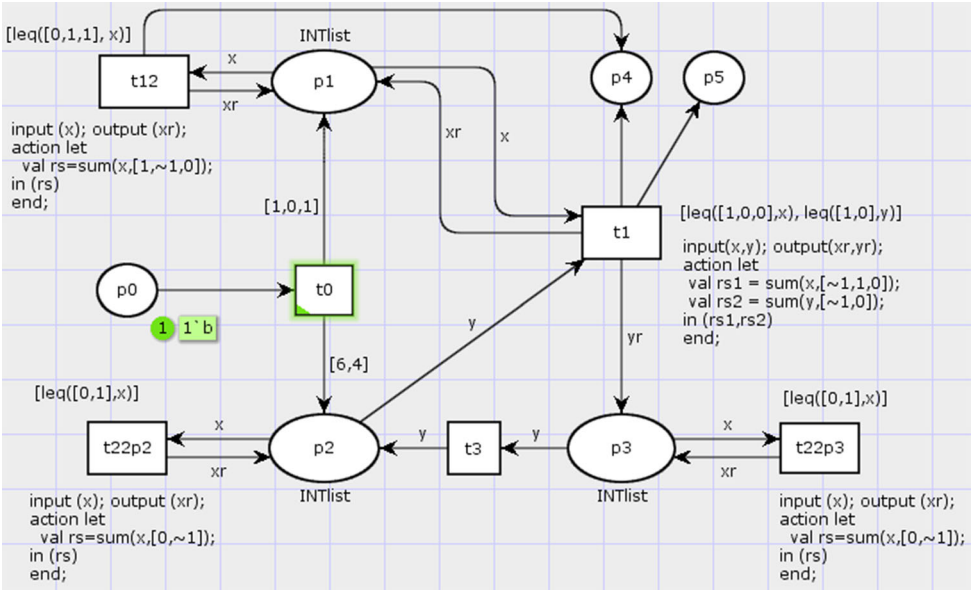



**Fig. 11** A CPN obtained from the NPN in Fig. 10 using the approach in [9]

mula <> [](p4 == 2 * p5) in 0.1s, a property that cannot be expressed in CTL. Therefore, both approaches are not competing and actually they may complement each other.

Nevertheless, the encoding of the correctness claims in CPN Tools is harder than in PROMELA because it requires some knowledge of the ML language. Besides, since our translation preserves the structure of the NPN, the interpretation of the counterexamples provided by SPIN is easier than those obtained with CPN Tools. Furthermore, SPIN performs on-the-fly verifications; thus it may discover a flaw without building the whole state space. For instance, consider the version of the NPN in Fig. 10 with an additional output arc ($t12$, $p12$). Now the resulting NPN is not bounded. This modification entails a slight change on both translations: In the CPN, the list used in the action of transition $t12$ is replaced by `[1,0,0]`; in the PROMELA model, we may simply remove the assignment `p12-` or add `p12++` in the code of the transition (see Fig. 12, line 6).

CPN Tools failed to generate any report for the new CPN after three consecutive attempts to calculate the state space (5min each). The basic safety verification of the PROMELA model took 1.12s, but the search could not be completed because the maximum depth parameter was too small (`pan -m10000`). By increasing the search depth by a factor of ten, the search was completed in 1.17s. Since no place invariant was added to the model, SPIN did not find any error. Nonetheless, a counterexample to the LTL property was found in 0.098s and depth 23653. The resulting trail includes several overflow messages[9] involving the increment in $p11$ and $p4$ when firing $t12$. These errors are not reported in verifications because, by default, SPIN truncates any value outside the domain. However, this is a strong indication that both places are unbounded. After adding a place invariant (e.g., `assert(p11 < 255)` in Fig. 10, line 6), the basic safety verification detected the assertion violation in 0.037s and depth 1367.

## 6 A note on priorities and SPIN performance

As we explained in Sects. 3.2 and 4, the processes involved in a step of the NPN are executed with priority greater than 1. However, the execution order is fixed by the order of the messages stored at `gbChan`. Therefore, unlike in [62], the behavior of the PROMELA model with and without priorities is almost the same; the difference is in the `set_priority` expressions that are removed in the latter case. The advantage of using this feature is that it speeds up the finding of counterexamples to invalid properties. This is due to the fact that SPIN schedules for execution only the enabled processes with highest priority. Since our translation uses several levels of priorities, the number of such processes is usually small. Besides, just one process will receive the first message at `gbChan`; thus no interleaving is required. Hence, the time for choosing the next executable process is reduced.

For full verification, the use of priorities (hereafter denoted as **+PRT**) requires compilation without the partial-order reduction (POR). The latter is an optimization technique applied to PROMELA steps that do not depend on the order of execution. In this case, the analyzer chooses an ordering and ignores the remaining combinations. In our translation, this optimization can be applied, e.g., when two different processes write request messages at different channels. Therefore, disabling priorities and using POR may help to decrease the memory and time needed for verifying valid

[9] `Error: value (256->0 (8)) truncated in assignment.`

**Fig. 12** PROMELA translation for the element net $EN1$ in Fig. 10

```
1  proctype EN1(chan pc){
2  byte p11=1; byte p12; byte p13=1;
3  endl: do
4        :: atomic{ empty(gbChan) &&  p12>0 &&  p13>0 -> /* t12 */
5                   set_priority(_pid,6)
6                   p12--;    /* Remove to add the arc (t12, p12) */
7                   p11++; p4++;
8                   set_priority(_pid,1) }
9        :: atomic{ empty(gbChan) &&  p11>0 && ! pc ?? [eval(_pid),1] ->
10                  pc!!_pid,1 }                          /* t11 req */
11       :: atomic{ gbChan ? _,eval(_pid),lt,pc ->
12                  if :: lt==1  -> p11--; p12++; p4++;  /* t11 */
13                     :: lt==10 -> skip
14                  fi;
15                  set_priority(_pid,1)  }
16 od }
```

properties. For instance, for the refined model of our running example (Fig. 7) the verification time is slightly reduced as the next table shows. But it doubles if none of these options are used (-**PRT** -**POR**).

| Option | States | Time (s) | Memory (Mb) | Depth |
|---|---|---|---|---|
| +**POR** | 3568553 | 93.2 | 329.683 | 1934 |
| +**PRT** | 3789929 | 112 | 342.087 | 2184 |
| −**PRT** − **POR** | 7479529 | 232 | 595.384 | 1934 |

To illustrate SPIN performance using both approaches we use a smaller variant of our running example, as depicted in Fig. 13. Here, we investigate whether given an initial configuration of the agents, the NPN always reaches a dead marking where the number of agents coincides with the results. An example of such initial configuration (hereafter called *sound*) is the marking $I_0^1$ where $I_0^1(L1) = \{Agent(1,0,1)\}$, $I_0^1(L2) = \{Agent(1,1,2)\}$, $I_0^1(L3) = \{Agent(1,1,2),\ Agent(2,1,2)\}$, the remaining places in $SN$ are empty and $ar(c) = 3$. Modifying the number of autonomous steps does not affect the soundness of an initial marking. For example, $I_0^1$ is still sound if we define $na = 2$ for all net tokens ($I_0^2$). However, $I_0^1$ turns up unsound if $ar(c) = 2$ (denoted as $I_0^3$) or we situate an identical agent at $L1$ ($I_0^4$). This soundness property can be specified using the LTL formula <> [](`Results == NumberAgents`).

Table 4 summarizes the results of verifying the soundness of these markings using the PROMELA translation.[10]

We have also included two more demanding markings $I_0^5$ (sound) and $I_0^6$ (unsound), both having two more agents at $L4$ and $L5$. The verification for $I_0^5$ was completed using the `-DCOLLAPSE` option. Note that the POR is a bit more efficient for sound markings while priorities are better for unsound ones. We also think that the verification with priorities scales better than the POR since the effectiveness of the latter decreases as the number of processes and their dependencies grow.

To conclude we point out that the performance of the translation is highly influenced by the poor support of PROMELA to channel-based operations. The language lacks for predefined functions such as the non-deterministic choice,[11] counting, removing or moving several elements matching a pattern in a channel. This increases the complexity in verifications and produces larger error trails. An alternative implementation using embedded C code affects the readability in simulations with no significant gain in verification time.

## 7 Conclusions and future work

This paper presented a general translation from NPNs (with an arbitrary number of levels and recursion) into PROMELA. The translation respects the modular definition of a NPN and allows the addition and removal of places, transitions and element nets in an easy way. Furthermore, it can be extended to other features of CPNs such as guards for transitions, actions

[10] See http://www.dropbox.com/s/n7y5ib76ym9x59p/exNPNMASsmall.pml?dl=0.

[11] The first verification of the model may use the standard PROMELA receive statement, instead of the non-deterministic version.

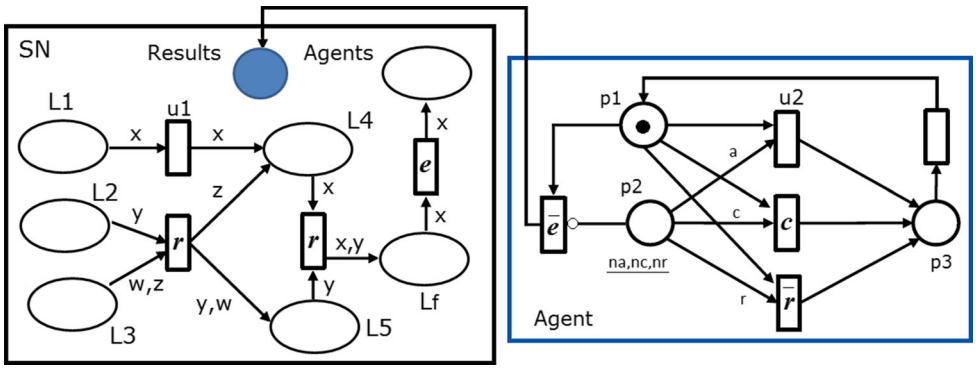



**Fig. 13** A simpler NPN for the multi-agent-based scenario

**Table 4** SPIN performance using priorities or the partial-order reduction

| Initial marking | States | Time (s) | Memory (Mb) | Depth |
|---|---|---|---|---|
| $I_0^1$ + **PRT** | 303331 | 9.05 | 132.797 | 736 |
| $I_0^1$ + **POR** | 316463 | 9.01 | 138.483 | 632 |
| $I_0^2$ + **PRT** | 1042645 | 32.8 | 455.868 | 775 |
| $I_0^2$ + **POR** | 1093273 | 32.4 | 478.017 | 653 |
| $I_0^3$ + **PRT** | 63 | 0.017 | 0.240 | 617 |
| $I_0^3$ + **POR** | 67 | 0.02 | 0.241 | 522 |
| $I_0^4$ + **PRT** | 76 | 0.011 | 0.239 | 814 |
| $I_0^4$ + **POR** | 81 | 0.012 | 0.241 | 701 |
| $I_0^5$ + **PRT** | 11965231 | 356 | 646.631 | 1004 |
| $I_0^5$ + **POR** | 12080327 | 331 | 652.883 | 882 |
| $I_0^6$ + **PRT** | 83 | 0.013 | 0.239 | 864 |
| $I_0^6$ + **POR** | 89 | 0.047 | 0.240 | 750 |

(e.g., printing information about the marking and binding) or more complex arc types (e.g., inhibitor and reset arcs).

The PROMELA translation can be used to simulate the behavior of a NPN without flattening its multi-level structure. To this end, SPIN model checker includes options for random and interactive simulation. In addition, SPIN provides a variety of searching algorithms that allow a fast detection of faults related to termination, boundedness and the violation of LTL properties. A full verification of valid properties may be achieved using advanced options, compression methods, approximation techniques or the swarm search on multi-core systems. Some techniques do not perform an exhaustive analysis but they may achieve a significant coverage of the state space, increasing the confidence in the correctness of a model.

We are currently working on automating the translation of NPNs specified using the Petri net markup language into PROMELA [19]. Besides, we would like to implement a plug-in for integrating SPIN with the simulation tool Renew. Two issues should be addressed in order to deal with the references nets in Renew instead of NPNs. Firstly, the translation should deal with several references to the same net object. But the reference may be the instantiation number of the process which can be easily transported, replicated and even moved between adjacent levels. The net processes may send the synchronization requests to a global channel. The number of references to a net token may also be stored in that channel in order to terminate the process once all its references have been consumed. The second difference is that Renew uses synchronous channels for vertical communication. But this mechanism can be easily implemented by means of asynchronous channels as we did for the vertical steps. Additional fields in the request–response messages can be used for exchanging data between the parent and the child nets.

The main limitations in using SPIN are its weak fairness model and the restriction to LTL properties. Therefore, unlike CPN tools, information concerning home markings, liveness, fairness and CTL properties is not easily obtained. However, the ideas behind the translation may provide guidelines for using other model checkers, e.g., XTL and JPF2 [4]. Research is still needed in order to develop effective techniques and reduction strategies for model checking multi-level, recursive and other variants of nested nets.

**Acknowledgments** We are grateful to Gerard J. Holzmann for his prompt replies to several questions concerning the use of process priorities in SPIN. We also thank the anonymous reviewers for their comments and suggestions that helped to improve the presentation of this paper.

## Appendix 1: The code for dealing with channels representing net-typed places

The next embedded C code is used to count the number of request messages at a net place channel whose second field coincides with a given label. A similar C code can be used to compute the number of occurrences a given token at a channel representing a basic colored place(`numTok`). A call to the function `numMsg` should have the form `numMsg(qptr(PProcName->c-1),i)` where `c` is a channel and `i` is an integer. The prefix `PProcName->` (e.g., `Pinit->`) is required to refer a local channel inside a `c_expr`. The prefix `now.` must be used instead for a global channel.

```
c_code{
  typedef struct QNP {
            uchar Qlen; /* q_size */
            uchar _t;   /* q_type */
            struct {
                uchar fld0, fld1, fld2;
            } contents[MaxMsg]; } QNP;

  int numMsg(uchar *z, int lab){
            int n = ((Q0 *)z)->Qlen, c = 0;
            for (int k = 0; k<n; k++)
              if ( ((QNP *)z)->contents[k].fld1 == lab )   c++;
            return c;
  }
};
```

The inline definition `recMsg` below is used for receiving a request message from a channel in a non-deterministic way (denoted in Sect. 3 by the operator `?*`). The definitions `transpNetTok` and `consNetTok` implement the operations for moving and removing all request messages of a given net token process at a net place, respectively. All net token processes at a place are terminated using `consNetsAtPlace`.

```
  hidden byte nt,lt,it,nt1,i;
  chan cha =[MaxMsg] of {byte,byte};

  /* ch ?* f0,f1,f2 */
  inline recMsg(ch,f0,f1,f2) {
  ch ! 0,f1,f2;
  do:: ch ?? f0,f1,f2;
      if:: f0>0 ->
           ch ! f0,f1,f2;
           cha ! len(cha)+1,f0;
        :: else -> break
      fi
  od
  select (i : 1 .. len(cha) );
  cha ?? <eval(i),f0>;
  do :: cha?_,nt1;
       ch ?? eval(nt1),f1,it;
       ch !! nt1,f1,it;
    :: empty(cha) -> break
  od;
  ch ?? eval(f0),f1,f2;  }
```

```
 inline transpNetTok(ch, och, p){
 do :: ch ?? [eval(p),_,_] ->
       ch ?? eval(p),lt,it;
        och !! p,lt,it;
     :: else -> break
 od; skip }

inline consNetTok(ch, p){
do:: ch ?? [eval(p),_,_] ->
     ch ?? eval(p),_,_;
  :: else -> break
od; skip }

 inline consNetsAtPlace(ch){
 do:: ch ?? [_,MaxL,0] ->
      ch ?? nt,MaxL,0;
     consNetTok(ch, nt);
     gbChan !! 6-3,nt,MaxL,ch,1;
     set_priority(nt, 3);
   :: else -> break
 od; skip }
```

## Appendix 2: PROMELA program for the NPN in Fig. 2

This appendix includes the PROMELA translation for the net components in Fig. 2, Sect. 2.2. The `proctype` definition corresponding to the *Agent* element net uses input parameters for *na*, *nr* and *nc*. Besides, the place *p*2 is unfolded into three uncolored places *p*2*a*, *p*2*r* and *p*2*c* [61]. Some non-shared places of *SN* have been declared as global variables because they are used to specify the LTL property `p` in Sect. 5.1. The complete model can be found at http://www.dropbox.com/s/et6mhll7ze17j6t/exNPNMAS.pml?dl=0.

```
unsigned Tasks:2 = 2 ; byte Results; bit pOut; NetPlace(Agents); NetPlace(L1);
NetPlace(L2); NetPlace(L3); NetPlace(L4); NetPlace(L5); NetPlace(Lf);

proctype agentNet(chan pc; bit na,nc; byte nr){
bit p1=1; unsigned p2a:2, p2r:2, p2c:2; bit p3;
do
:: atomic{ empty(gbChan) &&  p1>0 && p2a>0 ->               /* unlab (left) */
          set_priority(_pid, 6);
          p1--; p2a--; rmConf(5); rmConf(3); p3++;
          set_priority(_pid, 1) }
:: atomic{ empty(gbChan) &&  p3>0 ->                        /* unlab (right) */
          set_priority(_pid, 6);
          p3--; p1++; Results++;
          set_priority(_pid, 1) }
:: atomic{ empty(gbChan) &&  ! pc??[eval(_pid),1] ->             /* rq -i */
          pc!!_pid,1 }
:: atomic{ empty(gbChan) &&  p1>0 && p2r>0                       /* rq -r */
                      && ! pc??[eval(_pid),3] -> pc!!_pid,3 }
:: atomic{ empty(gbChan) &&  p1>0 && p2c>0                       /* rq  c */
                      && ! pc??[eval(_pid),5] -> pc!!_pid,5 }
:: atomic{ empty(gbChan) &&  pc??[eval(_pid),5]  &&              /* hor sync */
                c_expr{ numMsg(qptr(PagentNet->pc-1), 5)>=3 } ->
          set_priority(_pid, 6);
          pc??eval(_pid),5;
          recMsg(pc, nt,5); gbChan !! 6-4,nt,5,pc; set_priority(nt, 4);
          recMsg(pc, nt,5); gbChan !! 6-4,nt,5,pc; set_priority(nt, 4);
          gbChan !! 6-4,_pid,5,pc; set_priority(_pid, 4); }
:: atomic{ gbChan ? _,eval(_pid),lt,pc ->
          if   :: lt==1 ->                                       /* -i */
                  p2a = p2a+na; p2r = p2r+nr; p2c = p2c+nc;
               :: lt==3 && p1>0 && p2r>0 ->                      /* -r */
                  p1--; p2r--; rmConf(5); p3++
               :: lt==5 && p1>0 && p2c>0 ->                      /* c */
                  p1--; p2c--; rmConf(3); p3++
               :: lt==10 -> skip
           fi;
           set_priority(_pid, 1) }
od }

proctype P1(){
chan pc; NetPlace(pA); NetPlace(pR); bit p4=1, p8, p5,p6,p7;
atomic{ gbChan ? _,eval(_pid),10,pc; set_priority(_pid,1); }
do
::  atomic{ empty(gbChan) &&  p4>0 && Results >= 5 ->
           set_priority(_pid, 6);
           p4--; p8++;                                 /* unlab (left-down) */
           set_priority(_pid, 1) }
:: atomic{ empty(gbChan) &&  p4>0 && Tasks>0 &&
                c_expr { numMsg(qptr(now.Agents-1),10)>=3 } ->
          set_priority(_pid, 6);
          p4--; Tasks--; p5++;                          /* unlab (left-up) */
          /* transport x without sync */
          recMsg(Agents, nt,10); transpNetTok(Agents,pA,nt); pA !! nt,10;
          gbChan !! 6-3, nt,10,pA; set_priority(nt, 3);
          /* transport y without sync */
          recMsg(Agents, nt,10); transpNetTok(Agents,pA,nt);pA !! nt,10;
          gbChan !! 6-3, nt,10,pA; set_priority(nt, 3);
          /* transport z without sync */
          recMsg(Agents, nt,10); transpNetTok(Agents,pA,nt); pA !! nt,10;
          gbChan !! 6-3, nt,10,pA; set_priority(nt, 3);
          set_priority(_pid, 1) }
:: atomic{ empty(gbChan) &&  p5>0 && c_expr { numMsg(qptr(PP1->pA-1),1)>=3 } ->
          set_priority(_pid, 6);                                  /* i */
          p5--; p6++;
          /* transport x - sync & move to the same place */
          recMsg(pA, nt,1); gbChan !! 6-5, nt,1,pA; set_priority(nt, 5);
          /* transport y - sync & move to the same place */
          recMsg(pA, nt,1); gbChan !! 6-5, nt,1,pA; set_priority(nt, 5);
          /* transport z - sync & move to the same place */
          recMsg(pA, nt,1); gbChan !! 6-5, nt,1,pA; set_priority(nt, 5);
          set_priority(_pid, 1) }
:: atomic{ empty(gbChan) &&  p6>0 && c_expr { numMsg(qptr(PP1->pA-1),3)>=3 } ->
          set_priority(_pid, 6);                                  /* r */
          p6--; p7++;
          recMsg(pA, nt,3); transpNetTok(pA,L1,nt);        /* transport x */
          gbChan !! 6-5, nt,3,L1; set_priority(nt, 5);
          recMsg(pA, nt,3); transpNetTok(pA,L2,nt);        /* transport y */
          gbChan !! 6-5, nt,3,L2; set_priority(nt, 5);
          recMsg(pA, nt,3); transpNetTok(pA,L3,nt);        /* transport z */
          gbChan !! 6-5, nt,3,L3; set_priority(nt, 5);
          set_priority(_pid, 1) }
```

```
:: atomic{ empty(gbChan) &&  p7>0 ->                      /* unlab (right) */
          set_priority(_pid,6);
          p7--;
          nt = run P1(); pR !! nt,10;
          gbChan !! 6-2,nt,10,pR, set_priority(nt,2); }
:: atomic{ empty(gbChan) &&  pR ?? [_,4] ->                         /* lf */
          set_priority(_pid, 6);
          /* consume */
          recMsg(pR, nt,4); consNetTok(pR,nt);
          gbChan !! 6-5,nt,4,pR; set_priority(nt, 5);
          p8++;
          set_priority(_pid, 1) }
:: atomic{ empty(gbChan) &&  p8>0 &&                          /* rq  -lf */
               ! pc ?? [eval(_pid),4] -> pc !! _pid,4; }
:: atomic{ gbChan ? _,eval(_pid),lt,pc ->
          if  :: lt==4 ->  p8--; break                          /* -lf */
              :: lt==10 -> skip      /* This option can be removed since  */
          fi;                        /* the net is never transported      */
          set_priority(_pid,1) }
od }

init{
NetPlace(pw1); bit pIn=1;
atomic{ set_priority(_pid, 2);                          /* Initial Marking */
       nt = run agentNet(Agents, 0, 1, 3); Agents !! nt,10;
       nt = run agentNet(Agents, 0, 1, 3); Agents !! nt,10;
       nt = run agentNet(Agents, 0, 1, 3); Agents !! nt,10;
       set_priority(_pid, 1) }
do
:: atomic{ empty(gbChan) &&  L1 ?? [_,3] ->                 /* r (left-up) */
           set_priority(_pid, 6);
           recMsg(L1, nt,3); transpNetTok(L1,L4,nt);       /* transport x */
           gbChan !! 6-5, nt,3,L4; set_priority(nt, 5);
           set_priority(_pid, 1) }
:: atomic{ empty(gbChan) &&  L2 ?? [_,3] && L3 ?? [_,3] ->  /* r (left-down) */
           set_priority(_pid, 6);
           recMsg(L2, nt,3); transpNetTok(L2,L5,nt);       /* transport y */
           gbChan !! 6-5, nt,3,L5; set_priority(nt, 5);
           recMsg(L3, nt,3); transpNetTok(L3,L4,nt);       /* transport z */
           gbChan !! 6-5, nt,3,L4; set_priority(nt, 5);
           set_priority(_pid, 1) }
:: atomic{ empty(gbChan) &&  L4 ?? [_,3] && L5 ?? [_,3] ->    /* r (right) */
           set_priority(_pid, 6);
           recMsg(L4, nt,3); transpNetTok(L4,Lf,nt);         /* transport x */
           gbChan !! 6-5,nt,3,Lf; set_priority(nt, 5);
           recMsg(L5, nt,3); transpNetTok(L5,Lf,nt);         /* transport y */
           gbChan !! 6-5,nt,3,Lf; set_priority(nt, 5);
           set_priority(_pid, 1) }
:: atomic{ empty(gbChan) &&  Lf ?? [_,10] ->                /* unlab (left) */
           set_priority(_pid, 6);
           /* transport x without sync */
           recMsg(Lf, nt,10); transpNetTok(Lf,Agents,nt); Agents !! nt,10;
           gbChan !! 6-3,nt,10,Agents; set_priority(nt, 3);
           set_priority(_pid, 1) }
:: atomic{ empty(gbChan) &&  pIn>0 ->                       /* unlab (right) */
           set_priority(_pid, 6);
           pIn--;
           nt = run P1(); pw1 !! nt,10;
           gbChan !! 6-2,nt,10,pw1; set_priority(nt,2);
           set_priority(_pid, 1) }
:: atomic{ empty(gbChan) &&  pw1 ?? [_,4] ->                        /* lf */
           set_priority(_pid, 6);
           /* consume with sync */
           recMsg(pw1, nt,4); consNetTok(pw1,nt);
           gbChan !! 6-5,nt,4,pw1; set_priority(nt, 5);
           pOut++;
           set_priority(_pid, 1) }
od }
```

## Appendix 3: Outline for the translation of the NPN in Fig. 8

This section outlines the `proctype` definitions corresponding to the net components of the NPN in Fig. 8, Sect. 5.2. We have omitted several branches in the loops that are similar to those provided in this and previous examples. Here we note that the labels in $L_v^+$ are not required by the translation. Hence, in the model the labels in $L_v^-$ are numbered from 1 to 10 and `MaxL` is defined as 15. See the entire model at http://www.dropbox.com/s/dhxo1bg2d196lf0/prosecutionEx.rar?dl=0. The logic program implementing the object net can be found at http://edoc.sub.uni-hamburg.de/informatik/volltexte/2009/60/.

```
proctype case(chan pc; bit p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12){
do ...
 :: atomic{ empty(gbChan) &&  p4>0 && p6>0 &&
                        ! pc??[eval(_pid),5] ->  pc!!_pid, 5}  /* rq join */
 :: atomic{ gbChan ? _,eval(_pid),lt,pc,rm ->
            if  :: lt==1  &&  p1>0 -> p1--; p2++                /* -rec */
                :: lt==2  &&  p2>0 -> p2--; p3++; p5++          /* -print */
                :: lt==3  &&  p3>0 -> p3--; p4++;               /* -verif */
                :: lt==4  &&  p5>0 -> p5--; p6++;               /* -comp */
                :: lt==5  &&  p4>0 && p6>0 -> p4--; p6--; p7++; /* -join */
                :: lt==6  &&  p7>0 -> p7--; p8++;               /* -exam */
                :: lt==7  &&  p8>0 -> p8--; p9++;               /* -dec */
                :: lt==7  &&  p8>0 -> p8--; p10++;              /* -dec */
                :: lt==7  &&  p8>0 -> p8--; p11++;              /* -dec */
                ...
                :: lt==15 -> skip
            fi;
            if :: rm   -> break
               :: else -> do::copyMsg ?? [eval(_pid),_] ->      /* copy */
                             copyMsg ?? eval(_pid),copyPCh;
                             nt = run case(copyPCh,p1,p2,p3,p4,p5,...,p11,p12);
                             copyPCh !! nt,15;
                           ::else -> break
                         od;
                         set_priority(_pid, 1)
            fi  }
od }

NetPlace(f2); NetPlace(f3);  ... ; NetPlace(f9);

init{ bit f1=1;
do ...
:: atomic{ empty(gbChan) &&  f1>0 ->
   set_priority(_pid,6)
   f1--;
   nt = run case(f2,1,0,0,0,0,0,0,0,0,0,0); f2 !! nt,15;
   ...
   nt = run case(f2,1,0,0,0,0,0,0,0,0,0,0); f2 !! nt,15;
   set_priority(_pid,1)
  }
:: atomic{ empty(gbChan) && f3 ?? [_,2] ->                    /* print */
           set_priority(_pid, 6);
           recMsg(f3, nt,2); transpNetTok(f3,f4,nt);         /* transport x */
           gbChan !! 6-5, nt,2,f4,0; set_priority(nt, 5);
           copyMsg !! nt,f6;                                 /* copy x */
           set_priority(_pid, 1) }
:: atomic{ empty(gbChan) &&  f5 ?? [_,5] && f7 ?? [_,5] ->    /* join */
           set_priority(_pid, 6);
           recMsg(f7, nt,5); consNetTok(f7,nt);              /* consume y */
           gbChan !! 6-5,nt,5,f7,1; set_priority(nt, 5);
           recMsg(f5, nt,5); transpNetTok(f5,f2,nt);         /* transport x */
           gbChan !! 6-5,nt,5,f2,0; set_priority(nt, 5);
           set_priority(_pid, 1) }
od }
```

## References


1. Augusto, J., Butler, M., Ferreira, C., Craig, S.: Using SPIN and STeP to verify business processes specifications. Perspect. Syst. Inform. LNCS **2890**, 207–213 (2003)
2. Barkaoui, K., Hicheur, A.: Towards analysis of flexible and collaborative workflow using recursive ECATN ets. In: ter Hofstede, A., Benatallah, B., Paik, H.-Y. (eds.) Business Process Management Workshops, LNCS, vol. 4928, pp. 232–244. Springer, Berlin, Heidelberg (2008)
3. Bednarczyk, M.A., Bernardinello, L., Pawlowski, W., Pomello, L.: Modelling mobility with Petri hypernets. In: Proceedings of 17th International Conference on Recent Trends in Algebraic Development Techniques, WADT'04, pp. 28–44 (2005)
4. Brat, G., Havelund, K., Park, S., Visser, W.: Java PathFinder-second generation of a Java model checker. In: Workshop on Advances in Verification (2000)
5. Cabac, L., Duvigneau, M., Moldt, D., Rölke, H.: Modeling dynamic architectures using nets-within-nets. In: Proceedings of Interna-

tional Conference on Applications and Theory of Petri Nets, LNCS, vol. 3536, pp. 148–167 (2005)
6. Ceška, M., Janoušek, V., Vojnar, T.: PNtalk—a computerized tool for object oriented Petri nets modelling. In: Computer Aided Systems Theory (EUROCAST'97), LNCS, vol. 1333, pp. 591–610. Springer, Berlin (1997)
7. Chang, L., He, X.: A model transformation approach for verifying multi-agent systems using SPIN. In: Proceedings ACM Symposium on Applied Computing, pp. 37–42 (2011)
8. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The maude 2.0 system. In: Proceedings of 14th International Conference on Rewriting Techniques and Applications, LNCS, vol. 2706, pp. 76–87 (2003)
9. Dworzański, L., Lomazova, I.: CPN tools-assisted simulation and verification of nested Petri nets. Autom. Control Comput. Sci. **47**(7), 393–402 (2013)
10. Eker, S., Meseguer, J., Sridharanarayanan, A.: The maude LTL model checker. In: Proceedings Workshop on Rewriting Logic and Its Applications, ENTCS, vol. 71, pp. 162–187 (2002)
11. Eshuis, R.: Symbolic model checking of UML activity diagrams. ACM Trans. Softw. Eng. Methodol. **15**(1), 1–38 (2006)
12. Farwer, B., Leuschel, M.: Model checking object Petri nets in Prolog. In: Proceedings 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, pp. 20–31 (2004)
13. Frappier, M., Fraikin, B., Chossart, R., Chane-Yack-Fa, R., Ouenzar, M.: Comparison of model checking tools for information systems. In: Proceedings 12th International Conference on Formal Engineering Methods and Software Engineering, pp. 581–596 (2010)
14. Gallardo, M.M., Merino, P., Pimentel, E.: A generalized semantics of PROMELA for abstract model checking. Form. Asp. Comput. **16**(3), 166–193 (2004)
15. Gannod, G.C., Gupta, S.: An automated tool for analyzing Petri nets using SPIN. In: Proceedings of 16th IEEE International Conference on Automated Software Engineering, pp. 404–407. IEEE Computer Society (2001)
16. Grahlmann, B., Pohl, C.: Profiting from SPIN in PEP. In: SPIN Workshop (1998)
17. Haddad, S., Poitrenaud, D.: Recursive Petri nets-theory and application to discrete event systems. Acta Inform. **44**(7–8), 463–508 (2007)
18. Hicheur, A., Ben Dhieb, A., Barkaoui, K.: Modelling and analysis of flexible healthcare processes based on algebraic and recursive Petri nets. In: Weber, J., Perseil, I. (eds.) Foundations of Health Information Engineering and Systems, LNCS, vol. 7789, pp. 1–18. Springer, Berlin, Heidelberg (2013)
19. Hillah, L., Kordon, F., Petrucci, L., Trèves, N.: PNML framework: an extendable reference implementation of the Petri net markup language. In: Proceedings of International Conference on Applications and Theory of Petri Nets, LNCS, vol. 6128, pp. 318–327 (2010)
20. Holzmann, G.J.: Tutorial: design and validation of protocols. Tutor. Comput. Netw. ISDN Syst. **25**, 981–1017 (1991)
21. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)
22. Holzmann, G.J.: An analysis of bitstate hashing. Form. Methods Syst. Des. **13**(3), 289–307 (1998)
23. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Boston (2003)
24. Holzmann, G.J., Peled, D.: An improvement in formal verification. In: Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques, pp. 197–211 (1995)
25. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Springer, Berlin (1992)
26. Jensen, K., Kristensen Rozenberg, L. (eds.): Coloured Petri Nets—Modeling and Validation of Concurrent Systems. Springer, Berlin (2009)
27. Jensen, K., Rozenberg, G. (eds.): High-Level Petri Nets: Theory and Application. Springer, Berlin (1991)
28. Kissoum, Y., Sahnoun, Z.: A recursive colored Petri nets semantics for AUML as base of test case generation. In: Proceedings IEEE/ACS International Conference on Computer Systems and Applications, pp. 785–792 (2008)
29. Koch, I.: Petri nets in systems biology. Softw. Syst. Model. **14**(2), 703–710 (2015)
30. Köhler, M., Moldt, D., Rölke, H.: Modelling mobility and mobile agents using nets within nets. ICATPN, LNCS **2679**, 121–139 (2003)
31. Kummer, O., Wienberg, F., Duvigneau, M., Schumacher, J., Köhler, M., Moldt, D., Rölke, H., Valk, R.: An extensible editor and simulation engine for Petri nets: Renew. In: Proceedings of International Conference on Applications and Theory of Petri Nets, LNCS, vol. 3099, pp. 484–493 (2004)
32. Lakos, C.: From coloured Petri nets to object Petri nets. In: ICATPN, LNCS, pp. 278–297. Springer, Berlin (1995)
33. Latella, D., Majzik, I., Massink, M.: Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. Form. Asp. Comput. **11**(6), 637–664 (1999)
34. Lehmann K., Cabac, L., Moldt, D., Rölke H.: Towards a distributed tool platform based on mobile agents. In: Eymann, T., Klügl, F., Lamersdorf, W., Klusch, M., Huhns, M.N. (eds.) Multiagent System Technologies. LNCS, vol. 3550, pp. 179–190. Springer, Berlin, Heidelberg (2005)
35. Leuschel, M., Massart, T.: Logic programming and partial deduction for the verification of reactive systems: an experimental evaluation. In: Proceedings 2nd Workshop on Automated Verification of Critical Systems, pp. 143–150 (2002)
36. Leyla, N., Mashiyat, A.S., Wang, H., MacCaull, W.: Towards workflow verification. In: Proceedings Conference of the Center for Advanced Studies on Collaborative Research, pp. 253–267 (2010)
37. Lomazova, I.A.: Nested Petri nets—a formalism for specification and verification of multi-agent distributed systems. Fundam. Inf. **43**(1–4), 195–214 (2000)
38. Lomazova, I.A.: Nested Petri nets: multilevel and recursive systems. Fundam. Inf. **47**, 283–293 (2001)
39. Lomazova, I.A.: Recursive nested Petri nets: analysis of semantic properties and expressibility. Program. Comput. Softw. **27**(4), 183–193 (2001)
40. Lomazova, I.A.: Modeling dynamic objects in distributed systems with nested Petri nets. Fundam. Inf. **51**(1–2), 121–133 (2002)
41. Lomazova, I.A.: Nested Petri nets for adaptive process modeling. In: Avron, A., Dershowitz, N., Rabinovich, A. (eds.) Pillars of Computer Science, LNCS, vol. 4800, pp. 460–474. Springer, Berlin, Heidelberg (2008)
42. Lomazova, I.A., Schnoebelen, P.: Some decidability results for nested Petri nets. In: 3rd International Andrei Ershov Memorial Conference Perspectives of System Informatics'99, LNCS, vol. 1755, pp. 208–220 (2000)
43. Mascheroni, M., Farina, F.: Nets-within-nets paradigm and grid computing. In: Jensen, K., Donatelli, S., Kleijn, J. (eds.) Transactions on Petri Nets and Other Models of Concurrency V, LNCS, vol. 6900, pp. 201–220. Springer, Berlin, Heidelberg (2012)
44. Mateescu, R., Garavel, H.: XTL: a meta-language and tool for temporal logic model-checking. In: Proceedings of International Workshop on Software Tools for Technology Transfer, BRICS, pp. 33–42 (1998)
45. Murata, T.: Petri nets: properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989)
46. Natarajan, V., Holzmann, G.J.: Outline for an operational semantics of PROMELA. In: The SPIN Verification System. Proceedings of

the 2nd SPIN Workshop 1996, DIMACS-Discrete Mathematics and Theoretical Computer Science, vol. 32 (1997)
47. Prisecaru, O., Jucan, T.: Interorganizational workflow nets: a Petri net based approach for modelling and analyzing interorganizational workflows. In: EOMAS, pp. 64–78 (2008)
48. Ratzer, A., Wells, L., Lassen, H., Laursen, M., Qvortrup, J., Stissing, M., Westergaard, M., Christensen, S., Jensen, K.: CPN tools for editing, simulating, and analysing coloured Petri nets. In: Proceedings of the International Conference on Applications and Theory of Petri Nets, LNCS, vol. 2679, pp. 450–462 (2003)
49. Regis, G., Ricci, N., Aguirre, N., Maibaum, T.: Specifying and verifying declarative fluent temporal logic properties of workflows. In: 15th Brazilian Symposium on Formal Methods, LNCS, vol. 7498, pp. 147–162 (2012)
50. Reisig, W. (ed.): Elements of Distributed Algorithms: Modeling and Analysis with Petri nets. Springer, Berlin (1998)
51. Ribeiro, L., dos Santos, O., Dotti, F., Foss, L.: Correct transformation: from object-based graph grammars to PROMELA. Sci. Comput. Program. **77**(3), 214–246 (2012)
52. Ribeiro, O., Fernandes, J.: Translating synchronous Petri nets into PROMELA for verifying behavioural properties. In: International Symposium on Industrial Embedded Systems, pp. 266–273 (2007)
53. Ribeiro, O., Fernandes, J., Pinto, L.: Model checking embedded systems with PROMELA. In: IEEE International Conference Engineering of Computer-Based Systems, pp. 378–385 (2005)
54. Ruys, T.C., Holzmann, G.J.: Advanced SPIN tutorial. In: 11th International SPIN Workshop Model Checking Software, pp. 304–305 (2004)
55. Sbai, Z., Missaoui, A., Barkaoui, K., Ben Ayed, R.: On the verification of business processes by model checking techniques. In: Proceedings of the 2nd International Conference on Software Technology and Engineering, vol. 1 (2010)
56. Seghrouchni, A.F., Haddad, S.: A recursive model for distributed planning. In: Proceedings of International Conference on Multi-Agent Systems, pp. 307–314 (1996)
57. Szpyrka, M., Biernacka, A., Biernacki, J.: Methods of translation of Petri nets to NuSMV language. In: Proceedings of 23rd Workshop on Concurrency, Specification and Programming, pp. 245–256 (2014)
58. van der Aalst, W.M.P.: Business process management as the Killer App for Petri nets. Softw. Syst. Model. **14**(2), 685–691 (2015)
59. Valk, R.: Petri nets as token objects: an introduction to elementary object nets. In: ICATPN, vol. 1420, pp. 1–25 (1998)
60. Valk, R.: Object Petri nets: using the nets-within-nets paradigm. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets, LNCS, vol. 3098, pp. 819–848. Springer, Berlin, Heidelberg (2004)
61. Venero, M.: Verifying cross-organizational workflows over multi-agent based environments. In: Barjis, J., Pergl, R. (eds.) Enterprise and Organizational Modeling and Simulation, LNBIP, vol. 191, pp. 38–58. Springer, Berlin, Heidelberg (2014)
62. Venero, M.L.F., da Silva, F.S.C.: On the use of SPIN for studying the behavior of Nested Petri nets. In: Iyoda, J., de Moura, L. (eds.) Formal Methods: Foundations and Applications. LNCS, vol. 8195, pp. 83–98. Springer, Berlin, Heidelberg (2013)
63. Weise, C.: An incremental formal semantics for PROMELA. In: Proceedings of 3rd International SPIN Workshop (1997)
64. Yamaguchi, S., Yamaguchi, M., Tanaka, M.: A soundness verification tool based on the SPIN model checker for acyclic workflow nets. In: Proceedings of 23rd International Conference on Circuits/Systems, Computers and Communications, pp. 285–288 (2008)

**Mirtha Lina Fernández Venero** received her B.Sc. in Computer Science (1994) from the University of Oriente and her M.Sc. in Applied Computing (1996) from the Central University "Marta Abreu" of Las Villas, Cuba. From 1996 until early 2009, she was a full-time professor at the Computer Science Department of the University of Oriente. She received her Ph.D. in Software (2007) from the Technical University of Catalonia, Spain. She held a postdoc position at the University of São Paulo and she is currently a visiting professor at the Federal University of ABC, Brazil. Her research interests include formal methods, software verification and automated reasoning.

**Flávio Soares Corrêa da Silva** is Associate Professor at University of Sao Paulo, Brazil. Flavio completed his PhD in Artificial Intelligence at the University of Edinburgh in 1992 and has been a full-time faculty member at the University of São Paulo since then. He has published over 90 articles in international journals and conferences, and participated in several funded research projects, in many of them as Principal Investigator. His research interests have focused on a variety of topics, such as Knowledge Representation, Intelligent Interactive Systems, Electronic Government, Digital Democracy, Digital Entertainment and Gamification. Flavio has authored eight technical books, one of which has been awarded the prestigious Jabuti Literary Prize in Brazil in 2007.