On the Computation of Stubborn Sets of Colored Petri Nets

Sami Evangelista, Jean-François Pradat-Peyre

▶ **To cite this version:**

Sami Evangelista, Jean-François Pradat-Peyre. On the Computation of Stubborn Sets of Colored Petri Nets. 2006, pp.146-165. hal-00145906

HAL Id: hal-00145906

https://hal.science/hal-00145906

Submitted on 12 May 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On the Computation of Stubborn Sets of Colored Petri Nets

Sami Evangelista and Jean-François Pradat-Peyre

CEDRIC - CNAM Paris
292, rue St Martin, 75003 Paris
{evangeli, peyre}@cnam.fr

Abstract. Valmari's Stubborn Sets method is a member of the so-called *partial order methods*. These techniques are usually based on a selective search algorithm: at each state processed during the search, a stubborn set is calculated and only the enabled transitions of this set are used to generate the successors of the state. The computation of stubborn sets requires to detect dependencies between transitions in terms of conflict and causality. In colored Petri nets these dependencies are difficult to detect because of the color mappings present on the arcs: conflicts and causality connections depend on the structure of the net but also on these mappings. Thus, tools that implement this technique usually unfold the net before exploring the state space, an operation that is often untractable in practice. We present in this work an alternative method which avoids the cost of unfolding the net. To allow this, we use a syntactically restricted class of colored nets. Note that this class still enables wide modeling facilities since it is the one used in our model checker Helena which has been designed to support the verification of software specifications. The algorithm presented has been implemented and several experiments which show the benefits of our approach are reported. For several models we obtain a reduction close or even equal to the one obtained after an unfolding of the net. We were also able to efficiently reduce the state spaces of several models obtained by an automatic translation of concurrent software.

1 Introduction

State space analysis is a powerful formal method for proving that finite systems match their specification. It consists of enumerating all the possible configurations (or states) of the system to track the erroneous ones. A major obstacle to the application of this technique to industrial systems is the famous state explosion problem: the number of reachable states can be far too large to fit in memory or even on disk.

The state explosion has been the subject of many researches in the last two decades and techniques that alleviate this problem have been introduced. This includes the family of partial order methods which tackle one of the main sources of the combinatorial explosion: the concurrent execution of several components. These are based on the following observation: due to the interleaving semantic

S. Donatelli and P.S. Thiagarajan (Eds.): ICATPN 2006, LNCS 4024, pp. 146–165, 2006.
© Springer-Verlag Berlin Heidelberg 2006

of concurrent systems, a set of different executions can have exactly the same effect on the system and be only a permutation of the same sequence. Thus, an efficient way to reduce the state explosion would be to explore only a single or some representative executions and ignore all the others permutations that are equivalent to the chosen ones. This is why the term of *model checking using representatives* [14] seems more appropriate than the one of *partial order model checking*.

Valmari's stubborn sets method [17, 16, 18] is a member of this family. It is based on a selective search algorithm: at each state processed, a stubborn set of transitions is computed and only the enabled transitions of this set are used to generate the successors of the state. The elimination of some transitions from the stubborn set can cause some states not to be explored and can thus greatly reduce the number of visited states.

The problem of deciding if a set of transitions is stubborn at a state is at least as hard as the reachability problem [5]. Thus, selective search algorithms exploit the structure of the modeled system to build sets for which the stubbornness conditions are guaranteed to hold. Construction algorithms are thus tightly linked to the formalism of the model, e.g., Promela [10], Petri nets, but, whatever the formalism, they always rely on the notion of dependency between transitions. When adding a transition to the stubborn set, one has to find the transitions that could disable or enable the considered transition. For ordinary PT-nets, these dependencies can be directly deduced from the structure of the net. For this reason, stubborn sets have been widely studied in the field of ordinary Petri nets and many algorithms were introduced to solve the problem. Examples include the *deletion algorithm* [17, 22], and the *incremental algorithm* [17, 22, 16].

The computation of stubborn sets for colored Petri nets is more problematic. Indeed, the detection of dependencies between transitions can not solely rely on the structure of the net but must also consider the color mappings that label arcs between places and transitions. A brute force approach consists then in unfolding the net, i.e., building the equivalent ordinary Petri net, in order to apply traditional stubborn sets algorithm for PT-nets. However, this unfolding step is not always possible because of large color domains. A possible way to avoid this unfolding is to work at the "colored" level, and to detect the dependencies between transitions symbolically. As a counterpart, we can not expect to obtain as good results as if the net was unfolded since the analysis of dependencies can not be as precise.

We investigate in this paper a solution for the computation of stubborn sets of colored Petri nets that does not rely on the unfolding of the net nor works at the transition binding level when constructing a stubborn set. Our algorithm detects dependencies directly on the colored Petri net. A keypoint is to use a slightly restricted class of colored Petri net that enables a symbolic detection of the dependencies. However, this class still allows wide modeling facilities since it is the one used in Helena [7] an explicit state model checker based on high level Petri nets and aimed at software verification.

Related works. Stubborn sets of colored Petri nets were initially introduced by Valmari in [19, 20]. His algorithm performs an implicit unfolding of the net: the net is not explicitly unfolded but the algorithm systematically enumerates transition bindings. The possibility to ignore the colors of the net, i.e., treat each colored transition as if it was an agglomeration of all its unfolded transitions, was also mentioned but not recommended, since it usually leads to unnecessary large stubborn sets.

Brgan and Poitrenaud proposed in [1] an optimized version of Valmari's algorithm for well formed Petri nets which exploits the good structuring of the color domains and mappings of this class. The idea of their algorithm is to translate some structural relations into equivalent constraints systems before the search. These systems express dependencies between transition bindings and are thus repeatedly solved during the search to compute stubborn sets. Though their method can efficiently speed up the detection of dependencies, their construction of stubborn sets still works at the transition binding level. Consequently, their approach is not much different from unfolding the net.

To our best knowledge, the only algorithm proposed so far, that does not work at the binding level is [13]. In this paper, the authors suggest that a possible way to obtain "good" stubborn sets for a colored Petri net without unfolding it is to add some extra informations on top of the structure of the net. The model designer must thus supply to the model checking tool some additional inputs, such as the type of the places (e.g., communication buffer, shared resource), that give crucial hints on the structure and the dynamic of the unfolded net. These informations are then used during the search in the stubborn sets construction. Their method seems to provide good reductions for a very low cost as acknowledged by the experimental results. The author argue that it is reasonable to assume that the type of information needed can be provided by the user. An analogy is made to the fact of typing variables in a program. However, there is no other possibility for the tool for validating these informations than unfolding the net or generating the complete state space, which is, by hypothesis, infeasible. Thus, in case of "typing error", the tool could produce wrong results. Another interesting contribution of [13] is a theoretical result about the complexity of the problem: the size of the unfolded PT-net is the worst-case time complexity of any algorithm that computes non-trivial stubborn sets. A trivial stubborn set include all the transition bindings of the net.

This paper is organized as follows. The next section recalls some basic definitions and notations on colored Petri nets and stubborn sets. Our algorithm is presented in Section 3. Section 4 presents a set of experimental results. Lastly we conclude in Section 5.

2 Formal Background

We recall in this section the basic definitions and notions on colored Petri nets [11] and stubborn sets that are needed for the comprehension of this paper. We assume that the reader is familiar with PT-nets and their dynamic behavior.

$\mathbb{N}$ will denote the set of positive integers, $\mathbb{N}^+$ the set $\mathbb{N} \setminus \{0\}$ and $\mathbb{B} = \{false, true\}$ the set of booleans.

The definition of colored Petri nets is based on multi-sets. A multi-set over a set S is a mapping from S to $\mathbb{N}$. The set of multi-set over a set S is denoted by $Bag(S)$. The addition, substraction, and comparison of multi-sets are defined in the usual way. $\emptyset$ denotes the empty multi-set. If m is a multi-set then $e \in m \Leftrightarrow m(e) > 0$.

If S is a set, S^* is the set of finite words over S and S^∞ is the set of infinite words over S. The "." operator shall be used to denote the concatenation of two sequences. ϵ shall denote the empty sequence.

We shall denote by $\mathcal{P}(S)$ the powerset of a set S, i.e., the set of its sub-sets.

2.1 Colored Petri Nets

Definition 1 (Colored Petri net). *A colored Petri net (or CPN) is a tuple $\langle P, T, C, W^-, W^+, \phi, m_0 \rangle$ where P is a finite set of* **places***; T is a finite set of* **transitions** *such that $P \cap T = \emptyset$; C a* **color function** *is a mapping from $P \cup T$ to Σ, a set of finite and non empty sets; W^- and W^+ the* **forward and backward incidence matrixes** *associate to each pair (p,t) of $P \times T$ a mapping from $C(t)$ to $Bag(C(p))$; ϕ a* **guard function** *associates to each $t \in T$ a mapping from $C(t)$ to $\mathbb{B}$; and m_0 an* **initial marking** *is an element of $\mathbb{M}$ the set of mappings which associate to each $p \in P$ an element of $Bag(C(p))$.*

From now on a CPN N will implicitly define the tuple $\langle P, T, C, W^-, W^+, \phi, m_0 \rangle$. Given a node n of $P \cup T$, $C(n)$ will be called the color domain of n. The set of inputs (resp. outputs) of a place p is the set ${}^\bullet p$ (resp. $p^\bullet$) defined by: ${}^\bullet p = \{t | W^+(p,t) \neq 0\}$ [1] (resp. $p^\bullet = \{t | W^-(p,t) \neq 0\}$). The same sets can be defined for a transition t : ${}^\bullet t = \{p | W^-(p,t) \neq 0\}$ and $t^\bullet = \{p | W^+(p,t) \neq 0\}$. These notations are extended to set of nodes as usual.

The firing rule defines the dynamic of the net.

Definition 2 (Firing rule). *Let N be a CPN and $m \in \mathbb{M}$. The instance c of transition t is firable (or enabled) at m (denoted by $m[(t,c)\rangle$) if and only if $\phi(t)(c) \wedge m(p) \geq W^-(p,t)(c)$. The firing of the instance (t,c) at m leads to a marking m' (denoted by $m[(t,c)\rangle m'$) defined by: $\forall p \in P, m'(p) = m(p) - W^-(p,t)(c) + W^+(p,t)(c)$.*

The state space of a CPN is the set of all the markings of the net which can be reached from the initial marking by a sequence of firings.

Definition 3 (State space). *Let N be a CPN. The state space (or reachability set) of N is the minimal set $S \subseteq \mathbb{M}$ such that $m_0 \in S$ and if $\exists m \in S, t \in T, c \in C(t)$ such that $m[(t,c)\rangle m'$ then $m' \in S$.*

In the remainder we will often refer to the unfolded net of a CPN. Such a net is an ordinary PT-net obtained from the colored one by creating a node for each place or transition instance of the net. The flow relation of the unfolded net can

[1] 0 denotes here the empty mapping from $C(t)$ to $Bag(C(p))$, i.e., $\forall c \in C(t), 0(c) = \emptyset$.

then be derived by a direct application of the color mappings which label the arcs of the CPN.

2.2 Stubborn Sets

The stubborn sets method can be used to build a reduced state space of the system. The construction of this reduced state space can be done by introducing a little modification into a standard search algorithm: at each state processed, a stubborn set of transitions is computed and only the enabled transitions of this set are used to generate the immediate successors of the processed state. The set is said to be stubborn because the transitions outside it can not affect its behavior: it remains stubborn after the firing of any sequence of non stubborn transitions. Consequently, some transitions may never be executed, reducing the number of explored states. In the best case the reduction is exponential. Such a modified algorithm is usually called a selective search algorithm.

The computation of the stubborn set must respect certain rules in order to preserve the desired property in the reduced state space. This has led to the introduction of several versions of the stubborn sets method. However, the notion of *dynamic stubbornness* is a common basis of many of these versions.

Definition 4 (Dynamic Stubbornness). *Let N be a Petri net, m be a marking of N and $S \subseteq T$. The set S is dynamically stubborn at m if conditions* **D1** *and* **D2** *hold where:*

D1 $\forall t \in S, \sigma \in (T \setminus S)^*, m[\sigma.t\rangle \Rightarrow m[t.\sigma\rangle$
D2 $(\exists t \in T \mid m[t\rangle) \Rightarrow \exists t \in S \mid \forall \sigma \in (T \setminus S)^*, m[\sigma\rangle \Rightarrow m[\sigma.t\rangle$

Condition D1 states that the firing of a sequence of transitions that are outside the stubborn set can not enable the firing of a disabled transition of S while D2 ensures that if the marking is not dead, then there is an enabled transition in the stubborn set (called a *key transition* in the stubborn set terminology) which remains firable after the firing of any sequence of non stubborn transitions.

It is well known that a selective search algorithm which computes stubborn sets having the dynamic stubbornness property builds a reduced state space which contains all the dead markings of the initial state space and at least one infinite sequence if such a sequence exists. Because of the *ignoring phenomenon* [19], that's basically all that this definition preserves: a firable transition may be infinitely forgotten in the stubborn set computation. In order to verify more elaborated properties, such as liveness properties, it is crucial to prevent this ignoring phenomenon and to add additional constraints in the computation of the stubborn set, possibly leading to less reductions. For instance it is usually needed that along each cycle of the reduced reachability graph, a transition enabled at a marking of the cycle belongs to at least one of the stubborn sets computed for the markings of the cycle. However, this subject is beyond the scope of this work and we will focus our attention on the dynamic stubbornness from definition 4. Thus in the remainder of the paper, we will call a stubborn set a set of transitions which respect the dynamic stubbornness property.

3 Symbolic Computation of Stubborn Sets

We present in this section an algorithm to compute stubborn sets of colored Petri nets. The proposed method does not rely on an explicit or implicit unfolding of the net but rather on the two following ideas:

- We treat instances classes instead of explicitly enumerating instances. We thus obtain an algorithm which complexity is independent from the size of the unfolded net.
- We define a class of colored Petri nets allowing an efficient detection of the dependencies between the transitions of the net.

Our method is therefore inspired by two works [1, 13] on the same subject which have been mentioned in the previous section.

In the remainder of the section we will proceed as follows. Firstly we give a static stubbornness definition for PT-nets on which our algorithm is based. Then we make an informal presentation of our algorithm with the help of an example which illustrates several difficulties that arise when directly handling colored nets instead of unfolding. A syntactically restricted class of CPN is introduced in the third part of the section along with additional definitions needed to handle bindings classes. The last part of the section is more technical and deals with a concrete implementation of the algorithm for our CPN class.

3.1 Static Stubbornness for PT-Nets

The dynamic stubbornness definition is based on the semantic of the net, i.e., its reachability graph, and it is thus difficult to obtain an "implementation" of this property. Moreover, it has been proved that the problem of deciding if a set is stubborn is at least as hard as checking reachability for the full reachability graph [5] which is exactly what we want to avoid. Consequently, rather than checking the dynamic stubbornness property for an arbitrary set of transitions, algorithms usually exploit the structure of the modeled system to produce sets of transitions for which the dynamic stubbornness is guaranteed to hold. Such algorithms have been proposed for various description languages, e.g., Promela [10], Variable/Transition systems [19], and Petri nets [17, 16, 22].

We now introduce a stubbornness definition for Petri nets. It is far from being optimal and could be refined, e.g., see [22], but we will use it for its simplicity.

Definition 5 (Stubbornness). *Let N be a Petri net, m be a marking and $S \subseteq T$. S is stubborn at m if the three following conditions are satisfied:*

1. $(\exists t \in T \mid m[t\rangle) \Rightarrow \exists t \in S \mid m[t\rangle$
2. **if** $t \in S$ **and** $m[t\rangle$ **then** $({}^\bullet t)^\bullet \subseteq S$
3. **if** $t \in S$ **and** $\neg m[t\rangle$ **then** $\exists p \in {}^\bullet t$ **such that** $m(p) < W^-(p,t)$ **and** ${}^\bullet p \subseteq S$

Item 1 prevents from picking an empty set if there are enabled transitions. Item 2 ensures that a firable transition t remains firable after the firing of a sequence which only includes transitions outside the stubborn set. Indeed, all the transitions which could prevent the firing of t, i.e., the set $({}^\bullet t)^\bullet$, are also in the

stubborn set. Let us note that all the enabled transitions selected in the stubborn sets are thus key transitions since they remain firable. At last item 3 ensures that a disabled transition t remains disabled after the firing of a sequence of non stubborn transitions since place p prevents the firing of t and transitions that could put tokens in it are also stubborn. The place p of this item is usually called a *scapegoat* in the stubborn sets terminology.

These three items are thus sufficient to prove the following proposition.

Proposition 1. *Let N be a Petri net, m be a marking and $S \subseteq T$. If S is stubborn at m then it is dynamically stubborn at m.*

The previous definition can be used to define a stubborn sets computation algorithm for PT-nets. This algorithm initiates the construction with an enabled transition and successively applies item 2 and 3 until reaching a fix point. If the marking does not enable any transition then it directly returns an empty set.

The goal of our work if to present a "colored" version of this algorithm. We will see that difficulty arise with the introduction of color mappings in the net and how to deal with this additional complexity.

3.2 Informal Presentation of the Algorithm

At first we informally present our algorithm with the help of the net of figure 1. This net models the part of a distributed system in which different client and server process interact. Clients compete for acquiring access to a pool of shared objects. To each object of the pool is an associated lock guaranteeing an exclusive access to the object. An idle client C can evolve in two different ways. First he can try to grab the lock of object O (transition *takeLock*). Alternatively, if a server S has acknowledged his request (transition *sendAck*) he can choose to receive and treat this acknowledgment (transition *receiveAck*). We denote $\{c_1, \ldots, c_n\}$ the set of clients, $\{s_1, \ldots, s_m\}$ the set of servers and $\{o_1, \ldots, o_l\}$ the set of shared objects.

We propose to detail a few steps of the construction of the stubborn set $\mathcal{S}$ at the marking m depicted.Binding $tb_1 = (takeLock, \langle C = c_1, O = o_1 \rangle)$ is enabled at m and we choose it to initialize the stubborn set $\mathcal{S}$.

Binding $tb_1 = (takeLock, \langle C = c_1, O = o_1 \rangle$ Since tb_1 is enabled we must apply the second item of definition 5 and identify the instances in conflict with it to

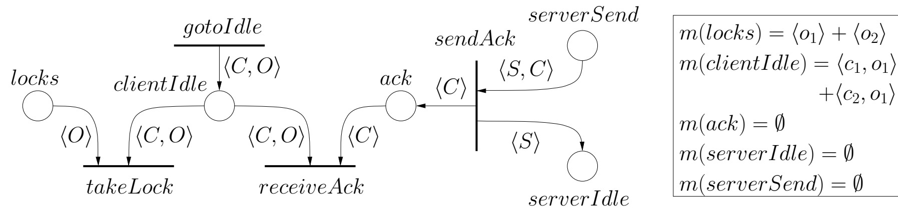


Fig. 1. An illustrative CPN example

include them in $\mathcal{S}$. It appears that binding $tb_2 = (receiveAck, \langle C = c_1, O = o_1 \rangle)$ is clearly in conflict with tb_1. Indeed, they both withdraw the token $\langle c_1, o_1 \rangle$ from place *clientIdle*. We therefore include tb_2 in $\mathcal{S}$. What seems less clear from a PT-net point of view is that tb_1 is also in conflict with other bindings of transition *takeLock*. An analysis of the unfolded net would indeed reveal that bindings of set $\cup_{c \in \{c_2,\ldots,c_n\}} \{(takeLock, \langle C = c, O = o_1 \rangle)\}$ also withdraw token $\langle o_1 \rangle$ from place *locks* and must thus be included in $\mathcal{S}$. Algorithms of [20] and [1] would directly insert these $n - 1$ bindings. The size of the stubborn set would then quickly increase as color domains grow. This may even fail with infinite color domains. Our approach is different since we choose to not explicitely enumerate these $n - 1$ bindings but rather to treat this set as a single unit. For this purpose we introduce the $\star$ symbol which will be used to denote "any item" of a color. Thus we will say that tb_1 is in conflict with binding $tb_3 = (takeLock, \langle C = \star, O = o_1 \rangle)$, i.e., all the clients who wishes to acquire the same lock. Since tb_3 is a compact representation of a set of bindings we will rather use the term bindings class or more simply class.

Binding $tb_2 = (receiveAck, \langle C = c_1, O = o_1 \rangle)$ To be enabled tb_2 needs a token $\langle c_1 \rangle$ in place *ack*. The instance $(ack, \langle c_1 \rangle)$ is therefore chosen as a scapegoat. The set $\cup_{s \in \{s_1,\ldots,s_m\}} \{(sendAck, \langle S = s, C = c_1 \rangle)\}$ includes all the bindings which can produce this token. We thus add class $tb_4 = (sendAck, \langle S = \star, C = c_1 \rangle)$ to $\mathcal{S}$. In other words, the only event that can cause the reception by c_1 of an acknowledgment is the sending of this acknowledgment whatever the sending server.

Binding $tb_3 = (takeLock, \langle C = \star, O = o_1 \rangle)$ The introduction of the concept of bindings class raises here a difficulty. Indeed, since we want to treat all the bindings of this class as a unit without differentiating them we have to assign the same status (disabled / enabled) to all these bindings in order to apply the second or third item of definition 5. In practice, the number of enabled bindings of a CPN always remains relatively low even when the color domains are large. We thus choose to proceed as follows. We treat each class as if all the bindings within it were disabled and we extract from it the enabled bindings which are inserted into the stubborn and treated separately. Thus an enabled binding added by this way to the stubborn set will be both considered as disabled and enabled. Though this will produce larger stubborn sets, this does not affect the validity of our algorithm. For instance, the two enabled bindings $tb_1 = (takeLock, \langle C = c_1, O = o_1 \rangle)$ and $tb_5 = (takeLock, \langle C = c_2, O = o_1 \rangle)$ both belong to the class tb_3 and must therefore be included in $\mathcal{S}$.

Let us come back to tb_3. Each disabled binding $b = (takeLock, \langle C = c_i, O = o_1 \rangle)$ of this class needs a token $\langle o_1 \rangle$ in *locks* and a token $\langle c_i, o_1 \rangle$ in *clientIdle*. Since $m(locks)(\langle o_1 \rangle) = 1$, the absence of token $\langle c_i, o_1 \rangle$ is the reason why b is disabled. Once again we avoid an explicit enumeration of these scapegoats by using the $\star$ symbol: the scapegoat of class tb_3 is $(clientIdle, \langle \star, o_1 \rangle)$. Such an approximation is valid since any disabled binding of tb_3 has a scapegoat place in this class. Item 3 requires that we insert in $\mathcal{S}$ the bindings which put in *clientIdle* tokens belonging to class $\langle \star, o_1 \rangle$. The single input of *clientIdle* is *gotoIdle*. By looking at the color

mapping which labels the arc from *gotoIdle* to *clientIdle* we notice that a binding $(gotoIdle, \langle C = c, O = o \rangle)$ can produce a token of class $\langle \star, o_1 \rangle$ if and only if $o = o_1$. Consequently, we must add $(gotoIdle, \langle C = \star, O = o_1 \rangle)$ to $\mathcal{S}$.

We mentioned in the introduction of this section that the time complexity of this algorithm is not related to the size of the unfolded net. As a matter of fact, the worst time complexity is indeed related to this size since each enabled binding can be processed and the number of enabled bindings is bounded by the number of transitions in the unfolded net. However, in practice, we observe that, even when the types of places and transitions are very large, the number of enabled bindings remains reasonable. So the worst time complexity of our algorithm is rarely reached.

This example highlighted the necessity to have CPNs with a well structured syntax. Indeed, each time we processed a binding we exploited the structuring of the arc functions to detect dependencies between transitions. This approach would typically fail with CPNs having arbitrarily complex arc functions or color domains. We introduce in the next sub-section a class of CPNs inspired by Well Formed colored nets [3] which, on one hand, allows us a symbolic detection of dependencies and, on the other hand, still enables large modeling facilities.

3.3 A Class of Colored Petri Nets

Color domains of our colored nets are Cartesian products of finite and non empty sets called basic types.

Definition 6 (Color domain). *The set of basic types Δ is a finite set of finite and non empty sets. A color domain C is a product $C_1 \times \cdots \times C_{s(C)}$ where $C_i \in \Delta$ and $s(C) \in \mathbb{N}$ is the size of C. The set of color domains is noted $\mathcal{C}$.*

A color domain item will be noted as a tuple, e.g., $\langle 2, true \rangle$. In addition, for each transition t, we will assume a bijective mapping associating each element of its color domain to a variable which will appear in the tuple, e.g., $\langle X = 2, B = true \rangle$.

Color mappings of the net are built with the help of elementary expressions. Two types of elementary expressions are allowed: the projection (or variable) and the functional expression. The first one is used to choose a specific element in an item of a color domain. Functional expressions are provided to enable complex operations on basic types. All the valid expressions from a color domain to a basic type can be put in this family. Projections are thus particular cases of functional expressions.

Definition 7 (Elementary expression). *Let $C \in \mathcal{C}$ and $\delta \in \Delta$. $\mathcal{E}_{C,\delta}$, the set of elementary expressions from C to δ is the set $\mathcal{E}_{C,\delta} = \mathcal{V}_{C,\delta} \cup \mathcal{F}_{C,\delta}$ where*

- $\mathcal{V}_{C,\delta} = \{V_i \mid i \in [1..s(C)] \wedge C_i = \delta\}$
  is the set of **projections** *from C to δ. V_i is defined by:*
  $\forall c = \langle c_1, \ldots, c_{s(C)} \rangle \in C, V_i(c) = c_i$.
- $\mathcal{F}_{C,\delta} = \{(f, (e_1, \ldots, e_n)) \mid f \in \delta_1 \times \cdots \times \delta_n \to \delta \wedge \forall i \in [1..n], e_i \in \mathcal{E}_{C,\delta_i}\}$
  is the set of **functional expressions** *from C to δ. $(f, (e_1, \ldots, e_n))$ is defined by:* $\forall c \in C, (f, (e_1, \ldots, e_n))(c) = f(e_1(c), \ldots, e_n(c))$

In our examples, instead of using the formal notation V_i we will prefer to use the variable of the transition at position i. For example V_2 will be directly noted B if the second element of the transition domain is associated to this variable.

Expressions tuples are basic components of color mappings. They can be guarded by a boolean expression which condition the tokens production. The syntax of guards will appear later in this section.

Definition 8 (Elementary expressions tuple). *Let $C, C' \in \mathcal{C}$. The set of elementary expressions tuples (or tuples) from C to $Bag(C')$, is noted $Tup_{C,C'}$ and is the set of triplets (γ, α, E) such that: $\gamma \in \mathcal{G}_C$ is the guard of the tuple; $\alpha \in \mathbb{N}^+$ is the factor of the tuple; $E = \langle e_1, \ldots, e_{s(C')} \rangle$, with $\forall i \in [1..s(C')], e_i \in \mathcal{E}_{C,C'_i}$, is the expressions list of the tuple. $tup = (\gamma, \alpha, \langle e_1, \ldots, e_n \rangle)$ is defined by: $\forall c \in C, tup(c) =$ **if** $\gamma(c)$ **then** $\alpha \cdot \langle e_1(c), \ldots, e_n(c) \rangle$ **else** $\emptyset$.*

A tuple $(\gamma, \alpha, \langle e_1, \ldots, e_n \rangle)$ will also be noted $[\gamma]\ \alpha \cdot \langle e_1, \ldots, e_n \rangle$. For instance $[X > Y]\ 2 \cdot \langle X, 0, f(Y) \rangle$ is a valid example tuple. Given an instantiation of the variables X and Y of the transition it produces 2 items of type $\langle X, 0, f(Y) \rangle$ if $X > Y$. Otherwise, it produces the empty multi-set.

At last, color mappings that label the arcs of the net are sums of elementary expressions tuples.

Definition 9 (Color mapping). *Let $C, C' \in \mathcal{C}$. A color mapping f from C to $Bag(C')$ is a sum $f = \sum_{i=1}^{k} tup_i$ (with $\forall i \in [1..k], tup_i \in Tup_{C,C'}$) defined naturally. The set of color mappings from C to $Bag(C')$ is noted $Map_{C,C'}$.*

A color mapping will sometimes be considered as the union of the tuples which constitute it, i.e., $tup \in map \Leftrightarrow map = tup + map'$.

Expressions tuples and transitions can be guarded by a boolean expression which states under which conditions the tuple produces items or the transition if firable. We do not impose special constraints on these guards. Any boolean expression on the color domain of the transition is a valid guard.

Definition 10 (Guard). *Let $C \in \mathcal{C}$. $\mathcal{G}_C = \mathcal{E}_{C,\mathbb{B}}$ is the set of guards over C.*

Dealing with instances classes. We first "extend" basic types by including to them the $\star$ symbol. The same extension is done for color domains. The definition assumes that this symbol does not belong to any basic type.

Definition 11 (Extended color domain). *Let $\delta \in \Delta$. The extended type $\delta^\star$ is the set $\delta \cup \{\star\}$. The set of extended types $\Delta^\star$ is the set $\{\delta^\star | \delta \in \Delta\}$. Let $C \in \mathcal{C}$. The extended color domain $C^\star$ is the Cartesian product ${C_1}^\star \times \cdots \times {C_{s(C)}}^\star$. The set of extended color domains is noted $\mathcal{C}^\star$.*

We must also modify the semantics of the elementary expressions to take into account this extension. The value of a projection is unchanged. If a sub-expression of a functional expression e is evaluated to $\star$ so is e. Otherwise its value does not change.

Definition 12 (Extended elementary expression). *Let $C \in \mathcal{C}$, $\delta \in \Delta$, and $e \in \mathcal{E}_{C,\delta}$. The extended expression $e^\star$ from $C^\star$ to $\delta^\star$ is defined by:*
$\forall c \in C^\star$ such that $c = \langle c_1, \ldots, c_n \rangle$:

$$e^\star(c) = \begin{cases} \textbf{if } e = V_i \textbf{ then } c_i \\ \textbf{if } e = (f, \langle e_1, \ldots, e_m \rangle) \textbf{ then } \begin{cases} \textbf{if } \forall i \in [1..m], e_i{}^\star(c) \neq \star \textbf{ then } e(c) \\ \textbf{else } \star \end{cases} \end{cases}$$

We will often use in this section the unfolding mapping defined below which is used to enumerate all the items within a class, e.g., $Unf_{\mathbb{B}\times\mathbb{B}}(\langle \star, true \rangle) = \{\langle false, true \rangle, \langle true, true \rangle\}$.

Definition 13 (Class unfolding). *Let $\delta \in \Delta$ and $C \in \mathcal{C}$. The mappings Unf_δ from $\delta^\star$ to $\mathcal{P}(\delta)$ and Unf_C from $C^\star$ to $\mathcal{P}(C)$ are defined by:*

- $Unf_\delta(e) = \textbf{if } e = \star \textbf{ then } \delta \textbf{ else } \{e\}$
- $Unf_C(\langle c_1, \ldots, c_n \rangle) = Unf_{C_1}(c_1) \times \cdots \times Unf_{C_n}(c_n)$

Lastly we introduce the inclusion relation $\succeq_C$ defined for every basic type or color domain C. We have $c \succeq_C c'$ if each item c_i of c is either the $\star$ symbol either c'_i, the item at the same position in c'. For instance, it holds that $\langle true, \star \rangle \succeq_{\mathbb{B}\times\mathbb{B}} \langle true, false \rangle$, but $\langle true, \star \rangle \succeq_{\mathbb{B}\times\mathbb{B}} \langle false, \star \rangle$ does not. Trivially, if $c \succeq c'$ then $Unf_C(c') \subseteq Unf_C(c)$. We will then say that c' is a subclass of c.

Definition 14 (Inclusion relation). *Let $C \in \mathcal{C}$. The relation $\succeq_C$ over $C^\star \times C^\star$ is defined by: $\langle c_1, \ldots, c_n \rangle \succeq_C \langle c'_1, \ldots, c'_n \rangle \Leftrightarrow \forall i \in [1..n], c_i = \star \vee c_i = c'_i$.*

We will omit in the sequel the subscript and superscript of Unf_C, $\succeq_C$ or $e^\star$ when there will be no ambiguity.

3.4 The Algorithm

We now introduce the general algorithm (figure 2) to compute a stubborn set of transitions of a CPN. Its input is a marking m of the CPN and it returns a stubborn set at m. Three main data structures are used. $\mathcal{S}$ is the stubborn set computed. $\mathcal{U}$ is the set of bindings classes which have not been treated yet. $\mathcal{N}$ is the set of binding classes which must be included in the stubborn set.

An enabled binding is randomly chosen to initialize the stubborn set and the set of unprocessed classes (line 1). If there is no enabled binding at m, the empty set is directly returned. At each iteration, a binding class (t, c_t) is removed from $\mathcal{U}$ and treated by the algorithm (lines 3-4). If this class is composed of a single binding, i.e. the $\star$ symbol does not appear in it, enabled at m we apply item 2 of the stubbornness definition and we compute the set of bindings classes in conflict with (t, c_t) (line 6). Otherwise we consider it as a class of disabled bindings. We first check (line 8) if c_t is not a subclass of a previously treated class c'_t, i.e., all the bindings of c_t are in c'_t. In this case the stubborn set is unchanged. Else the algorithm applies item 3 and computes the classes of bindings which produce tokens needed by the disabled bindings of (t, c_t). Additionally, we must include in $\mathcal{N}$ all the enabled bindings which belong to the bindings class (t, c_t) (line 11).

```
STUBBORN (m)
1  S ← if ∃(t, c_t) such that m[(t, c_t)⟩ then {(t, c_t)} else ∅;  U ← S
2  while U ≠ ∅ do
3     let (t, c_t) ∈ U with c_t = ⟨c_{t,1}, ..., c_{t,n}⟩
4     U ← U \ {(t, c_t)}
5     if ∀i ∈ [1..n], c_{t,i} ≠ ⋆ and m[(t, c_t)⟩ then
6        N ← disablingClasses(t, c_t)          (* apply item 2 of definition 5 *)
7     else
8        if ∃(t, c'_t) ∈ S such that c'_t ⪰ c_t then N ← ∅
9        else
10          N ← enablingClasses(t, c_t, m)      (* apply item 3 of definition 5 *)
11          N ← N ∪ {(t, c'_t) such that m[(t, c'_t)⟩ and c_t ⪰ c'_t}
12       end if
13    end if
14    U ← U ∪ (N \ S);  S ← S ∪ N
15 end while
16 return S
```

Fig. 2. A stubborn sets computation algorithm for colored Petri nets

3.5 Implementing Algorithm's Operations

The algorithm of figure 2 is a generic one in the sense that it is not related to our CPN class and could theoretically be implemented for any colored net. We have seen in our introductory example that an efficient implementation of the mappings *enablingClasses* and *disablingClasses* seems to require colored nets well structured enough to enable symbolic computations or alternatively some user supplied informations as it is done in [13].

We detail now an implementation of these operations for our CPN class.

Reversing color mappings. A frequently used operation in our algorithm is the reverse operation. This one consists of finding for a given color c and a color mapping f the set of colors c' such that $f(c')(c) > 0$. Different methods have been proposed in the literature to address this problem in an efficient way (e.g., [6], [1], [2], [8]) for Well Formed nets or similar classes. Even for this formalism, the problem is quite hard, and the solutions proposed either extend the formalism [2] or add some extra restrictions on arc functions [8].

For our colored nets, the reverse operation is, in general, impossible to apply without an explicit enumeration of colors. This is due to the possibility to insert in tuples some functional expressions which can obviously not be reversed. However, we can still approximate the result of this operation by exploiting the elementary expressions that are "well formed", i.e., the projections, and which are easily reversible.

We now introduce the mapping $reverseMap_{t,p}$ defined for every couple (p, t) of $P \times T$. Given a color mapping map from $C(t)$ to $Bag(C(p))$ and a class $c_p \in C(p)^\star$, $reverseMap_{t,p}$ computes a set of classes of transition t which is such that any binding of t which image by map contains a token of class c_p belongs to one of the classes computed. More formally, the following proposition must hold for all $c \in Unf(c_p)$ and $c' \in C(t)$:

$$map(c')(c) > 0 \Rightarrow \exists c_t \in reverseMap_{t,p}(c_p, map) \mid c' \in Unf(c_t)$$

Using the fact that a color mapping is a sum of tuples we reduce the problem to the definition of the tuple reversal mapping $reverseTup_{t,p}$ (definition 15).

We can clearly identify two steps in this procedure. In the first one we identify the class of bindings (t, c_t) such that $tup(c_t)(c) > 0$ for some c of class c_p. This is done by initializing the resulting class to $\langle\star, \ldots, \star\rangle$ which covers all the bindings of t and by looking for projections in tup. When a projection V_j is found at position i in the tuple we replace in r the $\star$ at position j by the item at position i in c_p. For instance $reverseTup_{t,p}(\langle 0, 1, 2\rangle, \langle X, Z, f(Y)\rangle) = \{\langle X = 0, Y = \star, Z = 1\rangle\}$. The class computed is clearly an over-approximation of the bindings which are really needed. This is mainly due to the fact that functional expressions may appear in the tuple and that such expressions can not be handled without being "unfolded". Since this is typically what we want to avoid, we have no other choice than ignoring these expressions in this first step. A possible optimization would be to identify expressions which can be reversed, and to exploit this reversibility, e.g., $reverseTup_{t,p}(\langle 1\rangle, \langle X + 1\rangle) = \{\langle X = 0\rangle\}$.

The second step of the reverse operation consists of checking if the computed class is inconsistent with respect to the tuple or the transition. A first inconsistency may appear if there is an expression e_i in the tuple which can be evaluated with binding r, i.e., $e_i(r) \neq \star$, and which value is different from c_i, the item at the same position in class c_p (if $c_i \neq \star$). We can then directly return the empty set since there is obviously no binding of t which can produce by tup a token of class c. For instance, $reverseTup_{t,p}(\langle 2, 2\rangle, \langle X, X + 1\rangle) = \emptyset$. A second inconsistency is detected if either the guard of the tuple or the guard of the transition can be evaluated and does not hold. For instance, $reverseTup(\langle 0, \star\rangle, [X > 0] \langle X, Y\rangle) = \emptyset$. Let us recall that these guards or the expressions in the tuple may not be evaluable since the variables which appear in these can have an undefined value.

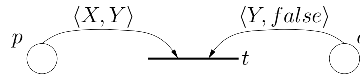
Definition 15. *Let $t \in T$ and $p \in P$. The mappings $reverseMap_{t,p}$ from $C(p)^\star \times Map_{C(t),C(p)}$ to $\mathcal{P}(C(t)^\star)$ and $reverseTup_{t,p}$ from $C(p)^\star \times Tup_{C(t),C(p)}$ to $\mathcal{P}(C(t)^\star)$ are defined by:*

$$reverseMap_{t,p}(c_p, map) = \bigcup_{tup \in map} reverseTup_{t,p}(c_p, tup)$$

$reverseTup_{t,p}\,(c_p, tup)$

1 **let** $c_p = \langle c_1, \ldots, c_n\rangle$
2 **let** $tup = [\gamma]\ \alpha \cdot \langle e_1, \ldots, e_n\rangle$
3 $r \leftarrow \langle\star, \ldots, \star\rangle$
4 **for** $i \in [1..n]$ **do if** $e_i = V_j$ **and** $r_j = \star$ **then** $r_j \leftarrow c_i$
5 *(* check inconsistencies *)*
6 **if** $\exists i \in [1..n]$ **such that** $e_i(r) \neq \star$ **and** $c_i \neq \star$ **and** $e_i(r) \neq c_i$ **then return** $\emptyset$
7 **if** $\gamma(r) = false$ **or** $\phi(t)(r) = false$ **then return** $\emptyset$
8 **return** $\{r\}$

Computing scapegoats. The treatment of a disabled bindings class c involves to identify the bindings firing of which can enable the elements of c. This detection is based, in our static stubbornness definition, on the ability to compute a *scapegoat*. A scapegoat of a low-level transition t at a marking m is a place which disables t, i.e. $W^-(p,t) > m(p)$. For a high-level transition binding (t, c_t) it is simply a couple (p, c_p) such that $p \in P, c_p \in C(p)$ and $W^-(p,t)(c_t)(c_p) > m(p)(c_p)$. When directly handling bindings classes instead of explicit bindings a difficulty appears that was not highlighted by our introductory example. Indeed, we must find a scapegoat for all the bindings within the class. Thus, in some cases, we will have to choose several scapegoats. We illustrate this problem with the help of the following net.



Let us consider the class $c_t = \langle X = 2, Y = \star\rangle$ of transition t for which we have two possible scapegoats: $(q, \langle\star, false\rangle)$ and $(p, \langle 2, \star\rangle)$. For instance, at the two markings m and m' our algorithm will proceed as follows:

- for m defined by $m(p) = \langle 2, 3\rangle, m(q) = \langle 4, true\rangle$
  Class $(q, \langle\star, false\rangle)$ is a valid scapegoat for all the bindings of c_t since no token in q has $false$ as its second component.
- for m' defined by $m'(p) = \langle 2, 3\rangle, m'(q) = \langle 4, false\rangle$
  Class $(q, \langle\star, false\rangle)$ can not be chosen since $m'(q) \geq W^-(t,q)(c)$ for $c = \langle X = 2, Y = 4\rangle$ which belongs to the class c_t. For the same reason, $(p, \langle 2, \star\rangle)$ can not be chosen since $m'(p) \geq W^-(p,t)(\langle X = 2, Y = 3\rangle)$. Consequently, both classes, i.e., $(q, \langle\star, false\rangle)$ and $(p, \langle 2, \star\rangle)$, must be chosen to ensure that each binding of c_t has a scapegoat in one of the two classes.

Once again, this example shows that working at the high-level has a cost insofar as we compute a set of scapegoats which, from a PT-net point of view, is clearly unnecessarily large.

The purpose of function *scapegoat* is to find, given a bindings class (t, c_t), and a marking m, a set of scapegoat classes for (t, c_t) at m: each disabled binding of (t, c_t) has a scapegoat in a class of $scapegoat(t, c_t, m)$. More formally, the following must hold for all $c \in Unf(c_t)$ such that $\neg m[(t,c)\rangle$:

$$\exists (p, c_p) \in scapegoat(t, c_t, m), c' \in Unf(c_p) \mid W^-(p,t)(c)(c') > m(p)(c')$$

The function proceeds in two steps. It first tries to find a unique class which is an acceptable scapegoat for all the bindings of c_t. A sufficient condition is that there is a tuple $[\gamma]\alpha \cdot \langle e_1, \ldots, e_n\rangle$ appearing in the input arc from p to t such that all the tokens present in p at m fulfill one of these two conditions:

- The multiplicity of the token is strictly less than the factor of the tuple α.
- There is an expression at position i in the tuple which produces with c_t a value different from $\star$ and different from the item at the same position in the token.

In addition, the guard of the tuple must evaluate to *true* with c_t. Otherwise, i.e., $\gamma(c_t) = false$ or $\gamma(c_t) = \star$, there could be instances in c_t for which the tuple does not produce any item. For these instances, the token consumed by the tuple is obviously not a valid scapegoat. If there exists such a tuple *tup*, then it is straightforward to see that $\forall c \in Unf(c_t), tup(c) > m(p)$. The token produced by the tuple can therefore be chosen as a scapegoat.

If we fail to find such a tuple then we pick all the tokens consumed by the bindings of c_t. This set is obviously a correct scapegoat for c_t. However, all the couples (p, c') for which it necessarily holds that $m(p)(c') \geq W^-(p,t)(c)(c')$ for any c of c_t can be safely withdrawn from this set. A sufficient condition for this to hold is that there exists a tuple *tup* in $W^-(p,t)$ such that (1) c_p, the image of c_t by *tup* does not contain the $\star$ symbol, (2) the multiplicity of c_p at m is greater than the maximal multiplicity of any item produced by $W^-(p,t)$, i.e., $\Gamma(W^-(p,t))$.

Definition 16. *The mapping scapegoat from $T \times \mathcal{C}^\star \times \mathbb{M}$ to $\mathcal{P}(P \times \mathcal{C}^\star)$ is defined by: scapegoat(t, c_t, m) =*

$$\begin{cases} \textbf{if} \quad \exists p \in {}^\bullet t, (\gamma, \alpha, \langle e_1, \ldots, e_n \rangle) \in W^-(p,t) \textbf{ such that} \\ \qquad\qquad \gamma(c_t) = true \\ \qquad \textbf{and} \ \forall c_p = \langle c_{p,1}, \ldots, c_{p,n} \rangle \in m(p), \ m(p)(c_p) < \alpha \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{or} \ \exists i \in [1..n] \mid e_i(c_t) \notin \{\star, c_{p,i}\} \\ \textbf{then} \ \{(p, \langle e_1(c_t), \ldots, e_n(c_t) \rangle)\} \\ \\ \textbf{else} \ \{(p, c_p) \mid p \in {}^\bullet t \textbf{ and } \exists (\gamma, \alpha, \langle e_1, \ldots, e_n \rangle) \in W^-(p,t) \textbf{ such that} \\ \qquad\qquad c_p = \langle e_1(c_t), \ldots, e_n(c_t) \rangle \ \textbf{and} \ \gamma(c_t) \neq false \\ \qquad \textbf{and} \ \neg(\forall i \in [1..n], e_i(c_t) \neq \star \ \textbf{and} \ m(p)(c_p) \geq \Gamma(W^-(p,t)))\} \end{cases}$$

where $\Gamma(map) = \sum_{i=1}^{k} \alpha_i$ *with* $map = \sum_{i=1}^{k} (\gamma_i, \alpha_i, E_i)$.

The size of the reduced reachability graph depends to a large extent on the stubborn sets computed. Though always choosing the stubborn set with the lowest number of enabled bindings does not necessarily yields the best reduction it seems however to be the best and simplest heuristic. The choice of the scapegoats is a nondeterministic factor which can affect the number of enabled bindings in the resulting stubborn set. For PT-nets different strategies have been proposed in [21]. Our implementation of mapping *scapegoat* sorts “scapegoat candidates” according to three criteria and chooses the first candidate according to this sorting. These three criteria are (we note C the set of bindings classes which are directly inserted into the stubborn set if the scapegoat if chosen): (1) the number of enabled bindings in C, (2) the number of $\star$ which appear in the classes of C, and (3) the number of classes in C. We thus try to limit the number of enabled bindings and the number of transitions of the unfolded net inserted to the stubborn set right after the choice of the scapegoat. Indeed, the number of low-level transitions covered by a class directly depends on the number of stars which appear in the class.

After intensive experiments we observed that this strategy competes favorably against a pseudo-random strategy.

Mappings *disablingClasses* and *enablingClasses*. Concluding, we define the mappings *disablingClasses* and *enablingClasses*.

The definition of mapping *disablingClasses* is based on this simple observation: a binding (t, c_t) remains firable as long as any binding which could withdraw tokens needed by (t, c_t) are not fired. It is thus sufficient to inspect the tokens consumed by (t, c_t) and to identify the bindings which consume these tokens by an application of the reverse operation. Therefore we closely follow the stubbornness definition for PT-nets (definition 5).

Definition 17. *The mapping disablingClasses from* $T \times \mathcal{C}^\star$ *to* $\mathcal{P}(T \times \mathcal{C}^\star)$ *is defined by:* $disablingClasses(t, c_t) =$

$$\bigcup_{p \in {}^\bullet t, t' \in p^\bullet, c_p \in W^-(p,t)(c_t)} reverseMap_{t',p}(c_p, W^-(p, t'))$$

To identify the bindings which can enable the instances of a class c_t of transition t we enumerate the scapegoats of (t, c_t) and, for each scapegoat (p, c_p), we look at each input transition t' of p. An application of the reverse operation gives us the bindings of t' which put tokens of class c_p in p. Once again this definition respects the static stubbornness definition for PT-nets.

Definition 18. *The mapping enablingClasses from* $T \times \mathcal{C}^\star \times \mathbb{M}$ *to* $\mathcal{P}(T \times \mathcal{C}^\star)$ *is defined by:* $enablingClasses(t, c_t, m) =$

$$\bigcup_{(p,c_p) \in scapegoat(t,c_t,m), t' \in {}^\bullet p} reverseMap_{t',p}(c_p, W^+(p, t')) \cdot$$

4 Experiments

The algorithm described in this work has been implemented in our model checker Helena [7]. This section reports the results of two series of experiments that have been carried out. In the first series considered we analyzed some models obtained from concurrent programs by an automatic translation using the Quasar [9] tool. In the second one we considered several academic models included in the Helena distribution (available at `http://helena.cnam.fr`) of which some are recurrent examples of the CPN literature.

All measures were obtained on a Pentium IV with a 2.4 Ghz processor.

Models extracted from programs (table 1). Four real concurrent programs were first translated using the tool Quasar: an implementation of Chang and Roberts election protocol, two different implementations of the dining philosophers and a client-server protocol with dynamic creation of servers to handle client requests. Helena could not unfold these CPNs due to the huge color domains of the nodes. Indeed, some places of the net model variables having high-level data types, e.g., records or arrays. Each program is scalable by a parameter and we considered several values of this parameter (the value is given in the first column) and ran two tests: one without the stubborn method enabled (column

Table 1. Data collected for some models extracted from concurrent software

	Complete Graph			Reduced Graph		
	$\|\mathcal{N}\|$	$\|\mathcal{A}\|$	$\mathcal{T}$	$\|\mathcal{N}\|$	$\|\mathcal{A}\|$	$\mathcal{T}$
	The leader election protocol					
7	198 039	1 041 750	61	45 780	93 361	30
8	1 037 209	6 175 069	644	201 943	430 120	184
9	5 961 241	40 179 197	10 035	824 362	2 061 193	1 038
	The dining philosophers (first implementation)					
7	1 398 615	5 050 508	138	29 412	44 166	4
8	5 416 243	21 585 453	1 038	83 670	127 191	14
9	24 842 432	112 433 417	13 581	221 865	319 833	40
	The dining philosophers (second implementation)					
4	26 539	55 245	1	6 322	7 667	1
5	219 304	505 765	11	37 139	46 911	7
6	1 789 459	4 582 322	145	214 853	359 901	43
	The client / server program					
2	4 141	14 461	1	108	131	0
3	130 221	593 583	14	1 131	1 434	0
4	5 445 681	30 593 745	2 508	13 921	19 232	1

Complete Graph), the second with it (column *Reduced Graph*). The columns $|\mathcal{N}|$, $|\mathcal{A}|$, and $\mathcal{T}$ indicate for each run the number of nodes and arcs of the graph, and the exploration time of the graph in seconds.

We observe that, despite the complexity of the CPNs obtained from an automatic translation of programs, our algorithm gives a significant reduction for the four programs considered. The reductions factor goes from 7 in the worst case (the leader election program) to almost 400 in the best case (the client / server program) and makes realistic the automatic verification of concurrent software.

To further enhance the reduction we plan to combine our method with [13]. Concurrent programs can indeed easily be mapped to process-partitioned CP nets of [13] with a simple static analysis of the program. For instance, places corresponding to variables local to a process can be typed as local and places corresponding to variables accessed by several processes can be typed as shared.

Academic models (table 2). We then considered several academic models which size allows (except for one) an unfolding. For each model we therefore ran an additional test with Prod and its stubborn set algorithm activated. Let us recall that Prod unfolds the net in order to apply the stubborn set method. We used the deletion algorithm of Prod (option `-d`) which is, to our best knowledge, the most advanced algorithm for PT-nets. The reduction observed with Prod must be seen as a lower bound which is hard to reach without unfolding the net. Comparing the size of the graph reduced by Helena to the size of the complete graph tells us how good the reduction is while comparing it to the graph reduced by Prod tells us how good the reduction could be.

The examples studied can be classified in four categories.

Table 2. Data collected for some academic models

Helena						Prod		
Complete Graph			Reduced Graph			Reduced Graph		
$\|\mathcal{N}\|$	$\|\mathcal{A}\|$	$\mathcal{T}$	$\|\mathcal{N}\|$	$\|\mathcal{A}\|$	$\mathcal{T}$	$\|\mathcal{N}\|$	$\|\mathcal{A}\|$	$\mathcal{T}$
The resource allocation system								
2 550 759	11 435 684	49	72 637	100 925	2	72 637	100 925	7
The distributed database system (from [11])								
649 540	4 330 282	19	67 585	112 662	1	232	242	0
The dining philosophers								
1 153 351	10 416 483	34	602 493	2 131 338	18	265 143	616 555	41
Eisenberg and McGuire's mutual exclusion algorithm								
624 790	2 490 418	9	414 555	1 345 417	129	223 482	428 297	491
The distributed Sieve of Eratosthenes								
2 028 969	9 947 808	59	273	272	0	Net not unfoldable		
Lamport's fast mutual exclusion algorithm (from [12]								
1 672 728	7 944 684	41	959 494	3 176 750	188	197 554	338 504	58
Chang and Roberts leader election protocol								
218 931	1 836 299	7	156 254	799 871	26	123 979	212 531	51
The load balancing system								
9 324 768	54 723 965	295	275 090	499 615	11	252 458	477 487	867
The multiprocessor system (from [4])								
7 322 076	85 522 635	356	138 239	283 646	76	138 239	283 646	572
Peterson's mutual exclusion algorithm								
1 242 528	4 970 112	19	186 037	386 272	46	80 193	152 565	38
The slotted ring protocol (from [15])								
439 296	2 897 664	11	287 508	97 8514	26	20 613	37 806	2

In a first category we can put the models for which the graph is weakly reduced by Helena but efficiently reduced by Prod. The two models which belong to this category are the slotted ring protocol and to a lesser extent Lamport's algorithm. We are currently not able to explain the bad results obtained for these two models but plan to investigate this.

The dining philosophers, Eisenberg and McGuire's algorithm, and the election protocol constitute a second category. Their characteristic is that their graph is weakly reduced both by Helena and Prod. A surface analysis may let us think that our algorithm performs a bad reduction for these models. However the poor results obtained with Prod show that we can not expect much better. Indeed, there are some problems for which partial order methods are inefficient. Models such as the dining philosophers or the Eisenberg and McGuire's algorithm which make heavy use of shared resources such as global variables usually fall into this category. These shared resources are a major source of conflicts which lead to compute large stubborn sets yielding a small reduction.

A third category is made up of the distributed database system and Peterson's algorithm. For these two models Helena builds a very reduced graph but Prod can do even better by unfolding the net. This particularly holds for the database system.

Its initial graph is in $\mathcal{O}(N \cdot 3^N)$. Helena builds a reduced graph in $\mathcal{O}(N^2 \cdot 2^N)$. Prod produces a reduced graph in $\mathcal{O}(N^2)$. An analysis of these nets reveals a certain complexity in arc mappings which makes it hard to efficiently detect dependencies without unfolding the net. Let us note that our reduced graph has exactly the same size as in [13] where they use a semi-automatic algorithm.

At last, for the four other models Helena performs a reduction close or equal to the reduction performed by Prod but outperforms significantly Prod with respect to time. The best results are observed for Eratosthene's algorithm: Helena builds a reduced reachability graph which size is linear with respect to the parameter of the system despite the high complexity of the arc functions. In addition, we notice that the net could not be unfolded because of the size of the color domains. For the multiprocessor and the resource allocation systems we obtain a reduced graph which is exactly the same as the one obtained by Prod. Lastly, for the load balancing system, the gain obtained by unfolding the net is completely marginal with respect to the brutal increase of the exploration time.

5 Conclusion

The contribution of this paper is a stubborn sets computation algorithm for colored Petri nets which avoids the unfolding by mixing two approaches. First we do not directly handle explicit bindings but rather bindings classes. We therefore stay at the high-level and never explicitly enumerate the transitions of the unfolded net. Second we define a syntactically restricted class of colored Petri nets for which it is possible to detect dependencies in a symbolic manner while preserving high expressiveness. As a counterpart, the detection of the dependencies between transitions can not be as fine as on the unfolded net and some approximations are done which lead to larger stubborn sets.

A set of experimental results have shown the benefits of our approach. For many academic models we achieved a reduction close or equal to the one obtained after an unfolding of the net. We were also able to significantly reduce the state spaces of several concurrent programs automatically translated to colored nets by the Quasar tool. The unfolding approach fails for these colored nets having huge unfolded nets.

In future works we will combine our algorithm with the method of [13]. Our algorithm exploits the structuring of our CPN class whereas their method is based on user supplied informations. Both should therefore be fully compatible and lead to better reductions.

References

1. R. Brgan and D. Poitrenaud. An efficient algorithm for the computation of stubborn sets of well formed petri nets. In *Application and Theory of Petri Nets*, volume 935 of *LNCS*, pages 121–140. Springer, 1995.
2. L. Capra, G. Franceschinis, and M. De Pierro. A high level language for structural relations in well-formed nets. In *Applications and Theory of Petri Nets*, volume 3536 of *LNCS*, pages 168–187. Springer, 2005.

3. G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. On well-formed coloured nets and their symbolic reachability graph. In *Application and Theory of Petri Nets*, pages 373–396. Springer, 1990.
4. G. Chiola, G. Franceschinis, and R. Gaeta. A symbolic simulation mechanism for well-formed coloured petri nets. In *Simulation Symposium*, 1992.
5. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.
6. C. Dutheillet and S. Haddad. Structural analysis of coloured nets. application to the detection of confusion. Technical Report 16, IBP/MASI, 1992.
7. S. Evangelista. High level petri nets analysis with Helena. In *Applications and Theory of Petri Nets*, volume 3536 of *LNCS*. Springer, 2005.
8. S. Evangelista, S. Haddad, and J.-F. Pradat-Peyre. Syntactical colored petri nets reductions. In *Automated Technology for Verification and Analysis*, volume 3707 of *LNCS*. Springer, 2005.
9. S. Evangelista, C. Kaiser, C. Pajault, J.-F. Pradat-Peyre, and P. Rousseau. Dynamic tasks verification with Quasar. In *Reliable Software Technologies*, volume 3555 of *LNCS*. Springer, 2005.
10. G.J. Holzmann and D. Peled. An improvement in formal verification. In *Formal Description Techniques*, pages 197–211, 1994.
11. K. Jensen. Coloured petri nets: A high level language for system design and analysis. In *Advances in Petri Nets*, volume 483 of *LNCS*. Springer, 1991.
12. J.B. Jørgensen and L.M. Kristensen. Computer aided verification of lamport's fast mutual exclusion algorithm using colored petri nets and occurrence graphs with symmetries. *IEEE Transactions on Parallel and Distributed Systems*, 10(7), 1999.
13. L.M. Kristensen and A. Valmari. Finding stubborn sets of coloured petri nets without unfolding. In *Application and Theory of Petri Nets*, volume 1420 of *LNCS*, pages 104–123. Springer, 1998.
14. D. Peled. All from one, one for all: on model checking using representatives. In *Computer Aided Verification*, volume 697 of *LNCS*, pages 409–423. Springer, 1993.
15. D. Poitrenaud and J.-F. Pradat-Peyre. Pre- and post-agglomerations for LTL model checking. In *Application and Theory of Petri Nets*, volume 1825 of *LNCS*, pages 387–408. Springer, 2000.
16. A. Valmari. Error detection by reduced reachability graph generation. In *Application and Theory of Petri Nets*, volume 424 of *LNCS*. Springer, 1988.
17. A. Valmari. *State Space Generation : Efficiency and Practicality*. PhD thesis, Tampere University of Technology, 1988.
18. A. Valmari. Eliminating redundant interleavings during concurrent program verification. In *Parallel Architectures and Languages*, volume 366 of *LNCS*, pages 89–103. Springer, 1989.
19. A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets*, volume 483 of *LNCS*, pages 491–515. Springer, 1991.
20. A. Valmari. Stubborn sets of coloured petri nets. In *Application and Theory of Petri Nets*, pages 102–121, 1991.
21. K. Varpaaniemi. On choosing a scapegoat in the stubborn set method. In *Workshop on Concurrency, Specification & Programming*, pages 163–171, 1993.
22. K. Varpaaniemi. *On the Stubborn Set Method in Reduced State Space Generation*. PhD thesis, Helsinki University of Technology, 1998.