Proceedings of the 2012 Winter Simulation Conference
C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. M. Uhrmacher, eds.

AN EFFICIENT METHOD FOR UNFOLDING COLORED PETRI NETS

Fei Liu

Harbin Institute of Technology
West Dazhi Street 92
150001 Harbin, CHINA

Monika Heiner

Brandenburg University of Technology
Walther-Pauer-Strasse 2
D-03046 Cottbus, GERMANY

Ming Yang

Harbin Institute of Technology
West Dazhi Street 92
150001 Harbin, CHINA

ABSTRACT

Unfolding is an essential problem in reusing existing Petri net simulation and analysis techniques and related tools for colored Petri nets. We present an efficient unfolding method, in which we provide two approaches to efficiently compute transition instances. That is, for a transition, if the color set of each variable in its guard is a finite integer domain, a constraint satisfaction approach is used to obtain all valid transition instances; otherwise, a general algorithm is adopted, in which some optimization techniques like partial binding – partial test and pattern matching are used. This method has been applied to unfold large-scale colored Petri nets, which has proven its efficiency.

1 INTRODUCTION

Colored Petri nets (Genrich and Lautenbach 1979; Jensen 1981) provide a compact and convenient way for modeling complex systems, but many basic properties and analysis techniques of standard Petri nets are difficult to extend to colored Petri nets. Therefore it is a reasonable approach to unfold colored Petri nets to equivalent standard Petri nets in order to use existing analysis techniques and related tools for standard Petri nets. Fortunately, each of our colored Petri nets corresponds to a standard Petri net as we only support finite color sets. Besides, we can also simulate colored stochastic (continuous) Petri nets using stochastic (continuous) simulation algorithms by automatic unfolding. Thus, unfolding plays an important role in analyzing and simulating colored Petri nets.

In order to unfold a colored Petri net we usually have to make an exhaustive search for the colors of places and transitions (the bindings defining transition instances), and then unfold each place color to an unfolded place and each transition instance to an unfolded transition. However, we are able to improve the unfolding efficiency considerably when we deploy some features of colored Petri nets and exploit some optimization techniques.

In this paper, we present an efficient unfolding method, which includes two approaches to efficiently compute transition instances. If the color set of each variable in a guard is a finite integer domain, a constraint satisfaction approach is used to obtain all valid bindings; otherwise, a general algorithm is adopted, in which some optimization techniques, e.g., pattern matching and partial binding – partial test principle, are used (Liu 2012).

This paper is organized as follows. Section 2 describes the mapping from colored Petri nets to their unfolded Petri nets, and Section 3 presents an unfolding algorithm. Then, Section 4 discusses how to

978-1-4673-4781-5/12/$31.00 ©2012 IEEE

compute transition instances for unfolding, and Section 5 gives some experimental results. In Section 6 we analyze related work, and conclude our paper with Section 7.

2 EQUIVALENCE BETWEEN COLORED PETRI NETS AND STANDARD PETRI NETS

In this section, we will recall standard Petri nets, colored Petri nets and their equivalence.

Petri nets. Petri nets can be defined as follows (Heiner, Gilbert, and Donaldson 2008).

Definition 1 (Petri net) A *Petri net* is a tuple $N =< P,T,F,f,m_0 >$ where:

- P is a finite, non-empty set of places.
- T is a finite, non-empty set of transitions.
- F is a finite set of directed arcs.
- $f : F \to \mathbb{N}_+$ is a function that assigns a positive integer to each arc $a \in F$.
- $m_0 : P \to \mathbb{N}_0$ gives the initial marking.

Colored Petri nets. In colored Petri nets, there are different types of expressions, i.e. guards, arc expressions, and expressions for defining initial markings or rate functions. An expression is built up from variables, constants, and operation symbols. An expression is not only associated with a particular color set, but has also to be written in terms of a predefined syntax. Here we denote by EXP a set of expressions that comply with a predefined syntax. Besides, a multiset is a set in which there can be several occurrences of the same element. The collection of all multisets over S is denoted by S_{MS}. Then we give the definition of colored Petri nets based on Jensen and Kristensen (2009).

Definition 2 (Colored Petri net) A colored Petri net is an eight-tuple $< P,T,F,\sum,C,g,f,m_0 >$, where:

- P is a finite, non-empty set of places.
- T is a finite, non-empty set of transitions.
- F is a finite set of directed arcs.
- $\sum$ is a finite, non-empty set of types, also called color sets.
- $C : P \to \sum$ is a color function that assigns to each place $p \in P$ a color set $C(p) \in \sum$.
- $g : T \to EXP$ is a guard function that assigns to each transition $t \in T$ a guard expression of Boolean type.
- $f : F \to EXP$ is an arc function that assigns to each arc $a \in F$ an arc expression of a multiset type $C(p)_{MS}$, where p is the place connected to the arc a.
- $m_0 : P \to EXP$ is an initialization function that assigns to each place $p \in P$ an initialization expression of a multiset type $C(p)_{MS}$.

Equivalence. If the color sets of a colored Petri net are finite, the colored Petri net exactly corresponds to an equivalent standard Petri net (Jensen 1992), which is described as follows.

Definition 3 (Unfolded Petri net) Let $N =< P,T,F,\sum,C,g,f,m_0 >$ be a colored Petri net, its unfolded Petri net $N^* =< P^*,T^*,F^*,f^*,m_0^* >$ is defined by:

1. $P^* = I_P$.
2. $T^* = I_T$.
3. $F^* = \{(p(c),t(b)) \in P^* \times T^* | (f(p,t)\langle b \rangle)\langle c \rangle > 0\} \cup$
   $\{(t(b),p(c)) \in T^* \times P^* | (f(t,p)\langle b \rangle)\langle c \rangle > 0\}$.
4. $\forall (p(c),t(b)) \in F^* : f^*(p(c),t(b)) = (f(p,t)\langle b \rangle)\langle c \rangle$,
   $\forall (t(b),p(c)) \in F^* : f^*(t(b),p(c)) = (f(t,p)\langle b \rangle)\langle c \rangle$.
5. $\forall p(c) \in P^* : m_0^*(p(c)) = m_0(p)\langle c \rangle$.

The explanations about the definition are as follows.

1. Each place instance (each color) in the place instance set I_P of the colored Petri net N corresponds to a place of the Petri net N^*. That is, the colored tokens in the colored Petri net are now distinguished by different places in its corresponding Petri net.
2. Each transition instance (each binding) in the transition instance set I_T of the colored Petri net N corresponds to a transition of the Petri net N^*. This means that each binding of the colored Petri net is instantiated as a transition in its corresponding Petri net.
3. If the occurrence of t with binding b removes at least one token of color c from p, denoted by $((f(p,t)\langle b\rangle)\langle c\rangle > 0$, then an arc that connects $p(c)$ and $t(b)$ exists for the Petri net, whose weight is the number of tokens of color c, denoted by $((f(p,t)\langle b\rangle)\langle c\rangle$. Analogously, If the occurrence of t with the binding b adds at least one token of color c to p, denoted by $((f(t,p)\langle b\rangle)\langle c\rangle > 0$, then an arc that connects $t(b)$ and $p(c)$ exists for the Petri net, whose weight is the number of tokens with color c, denoted by $((f(t,p)\langle b\rangle)\langle c\rangle$.
4. If the initial marking of the colored Petri net N contains tokens of color c on a place p, then the place $p(c)$ of the Petri net N^* has initial tokens, whose coefficient is the number of tokens of color c, denoted by $m_0(p)\langle c\rangle$.

Algorithm 1: Unfolding a colored Petri net.

```
   Input: a colored Petri net N =< P,T,F,Σ,C,g,f,m_0 >
   Output: an unfolded Petri net N* =< P*,T*,F*,f*,m_0* >
 1 for each transition t ∈ T do
 2     B = ComputeBindings(t);
 3     if V(t) is not empty and B is empty then
 4         t is isolated;
 5     endif
 6     for each binding b ∈ B do
 7         t*(b) ← t(b);
 8         for each pre-arc (p,t) of t do
 9             for each color c in (f(p,t)⟨b⟩) do
10                 p*(c) ← p(c);
11                 m_0*(p*(c)) ← m_0(p)⟨c⟩;
12                 f*(p*(c),t*(b)) ← (f(p,t)⟨b⟩)⟨c⟩;
13                 (p*(c),t*(b)) ← (p(c),t(b));
14             endfor
15         endfor
16         for each post-arc (t,p) of t do
17             for each color c in (f(t,p)⟨b⟩) do
18                 p*(c) ← p(c);
19                 m_0*(p*(c)) ← m_0(p)⟨c⟩;
20                 f*(t*(b),p*(c)) ← (f(t,p)⟨b⟩)⟨c⟩;
21                 (t*(b),p*(c)) ← (t(b),p(c));
22             endfor
23         endfor
24     endfor
25 endfor
```

3 UNFOLDING ALGORITHM

The unfolding algorithm is illustrated in Algorithm 1, which works as follows.

The algorithm executes a loop for each transition t of the net N. In each loop, it first computes valid bindings for transition t, which is realized by a sub-algorithm, *ComputeBindings*(t). If the variable set $V(t)$ of transition t is not empty, but the binding set B is empty, then the transition is an isolated transition, which can be excluded from the unfolded net right now (Lines 3-5). Afterwards, each binding $b \in B$ (corresponding to a transition instance $t(b)$) of transition t, is assigned to a Petri net transition $t^*(b)$ (Line 7). For each pre-arc $F(p,t)$ of t, we first evaluate its expression in terms of binding b, denoted by $(f(p,t)\langle b\rangle)$. For each color c in the evaluated expression $(f(p,t)\langle b\rangle)$, we assign a place instance $p(c)$ to a Petri net place $p^*(c)$

(Line 10). At the same time, the tokens corresponding to color c on place p are assigned to $p^*(c)$ (Line 11). Besides the number of tokens of color c in $(f(p,t)\langle b\rangle)$, denoted $(f(p,t)\langle b\rangle)\langle c\rangle$, is assigned to a Petri net arc expression $(f^*(p(c),t(b))$ (Line 12) and an arc $(p(c),t(b))$ is added to the unfolded net (Line 13). We deal with the post-arcs in the same way as the pre-arcs.

This algorithm implicitly discards those transition instances which evaluate the guard to false, which is in fact an optimal way for unfolding colored Petri nets. Besides, we consider other ways to optimize unfolding, e.g., removal of isolated places or transitions during the unfolding process.

4 ALGORITHMS FOR COMPUTING TRANSITION INSTANCES

When we unfold a transition, we have to compute all its instances (or bindings). In fact, the gain in the efficient unfolding mostly depends on the computation of transition instances. For a transition, the number of its instances is decided by its guard, which is in fact a logical expression. We can consider the computation of transition instances as a combinatorial problem. If we test a guard for each combination of values, then we would have the combinatorial explosion problem which we would like to avoid whenever possible. In this situation, we can think of using the constraint satisfaction approach to solve the problem of the computation of transition instances as it has been used in many combinatorial problems, e.g., scheduling and timetabling in operational research.

In the following, we will first address this issue and then consider a more general algorithm for computing transition instances.

4.1 Computing Transition Instances Using the Constraint Satisfaction Approach

A constraint satisfaction problem (CSP) (Brailsford, Potts, and Smith 1999; Tsang 1993) requires to pick for each variable a value from its finite domain, so that all given constraints concerning the variables are satisfied.

Formally, a constraint satisfaction problem is a triple $CSP =< V,D,C >$, where:

- $V = \{v_1, v_2, .., v_n\}$ is a set of variables.
- $D = \{d_1, d_2, .., d_n\}$ defines a finite domain d_i of possible values for each variable v_i, $i = 1,2,...,n$.
- C is a set of constraints on variables in V, which restricts the values taken by the variables.

A state of a CSP is defined by an assignment of values to the variables. A complete assignment means that each variable gets assigned a value. A solution to a CSP is a complete assignment satisfying all the constraints. For our purpose, we will restrict the domains of the variables to finite integer domains, and we are interested in all solutions of a CSP. That is, if all variables of the guard of a transition are finite integer domains, we can formulate it as a CSP. We implemented the constraint satisfaction approach in Snoopy using Gecode, an open constraint solving library (Gecode 2011).

Take Figure 1 as an example. We will see how to formulate the computation of transition instances as a CSP, which can take the following steps.

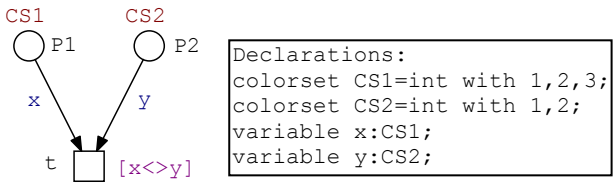

Figure 1: A colored Petri net for demonstrating the constraint satisfaction approach.

1. Obtain the guard of a transition, e.g., $x <> y$ (x is not equal to y) for transition t, which is a logic expression.
2. Obtain all the variables in the guard, e.g., x and y in $x <> y$, which will be the variables of the CSP.
3. Define the color set of each variable as its domain in the CSP. For example, the color sets $CS1$ and $CS2$ will become the domains of x and y in the CSP, respectively.
4. Define the guard as a constraint in the CSP. For example, $x <> y$ will become a constraint of the CSP. If the guard is a conjunctive form, we can define each conjunct as a constraint.

Using this procedure, we can formulate Figure 1 as a CSP: $V = \{x, y\}$, $D = \{CS1, CS2\}$ and $C = \{x <> y\}$. Then we can obtain all the solutions:

1. $\langle x = 1, y = 2 \rangle$
2. $\langle x = 2, y = 1 \rangle$
3. $\langle x = 3, y = 1 \rangle$
4. $\langle x = 3, y = 2 \rangle$

That is, transition t in Figure 1 has four instances.

4.2 A General Algorithm for Computing Transition Instances

However, if the guard of a transition is always evaluated to true or not all the variables in the guard are integer types, we have to take another way to compute transition instances. Here we will adopt a pattern matching approach.

4.2.1 Patterns

We basically use the same pattern matching mechanism as CPN tools (Kristensen and Christensen 2004). A pattern is defined as an expression with variables which can be matched with other expressions to assign values to variables (Kristensen and Christensen 2004). CPN tools are based on a standard meta language (SML) and thus employ the patterns defined in SML (Ullman 1998). In contrast, our tool is not based on SML, but we consider a subset of SML patterns, as we support less data types than CPN tools. The patterns that we use have the following syntactical structure:

$$
\begin{array}{lll}
\mathit{Pattern} & ::= & \text{“}\mathit{Variable}\text{”} \\
 & | & \text{“}\mathit{Constant}\text{”} \\
 & | & \mathit{TuplePattern} \\
\mathit{TuplePattern} & ::= & (\mathit{Pattern}(,\mathit{Pattern})*)
\end{array}
$$

Consider the example illustrated in Figure 2. According to the syntax of the patterns, we can see that (x, y) is a tuple pattern. If we assign the color $(1, a)$ on place $P4$ to (x, y), we obtain an assignment $x = 1$ and $y = a$. This process is called pattern matching (Kristensen and Christensen 2004).

Pattern matching provides an easy and efficient way to compute transition instances; therefore, in order to improve the efficiency of the computation, we have to find and use patterns for assigning subsets of color sets on places to variables as much as possible. To do this, we can search over all input arc expressions of a transition to find available patterns, which we call a pattern set concerning arc expressions, denoted by $AS(t)$ for a transition t. Besides, we can search the guard of a transition t to find patterns in the guard, denoted by $GS(t)$. These two sets constitute the overall pattern set for a transition t, $PS(t) = AS(t) \cup GS(t)$, which are used to assign subsets of color sets to variables. In the following, we discuss in detail how to find these two types of patterns.

(1) Patterns in arc expressions.

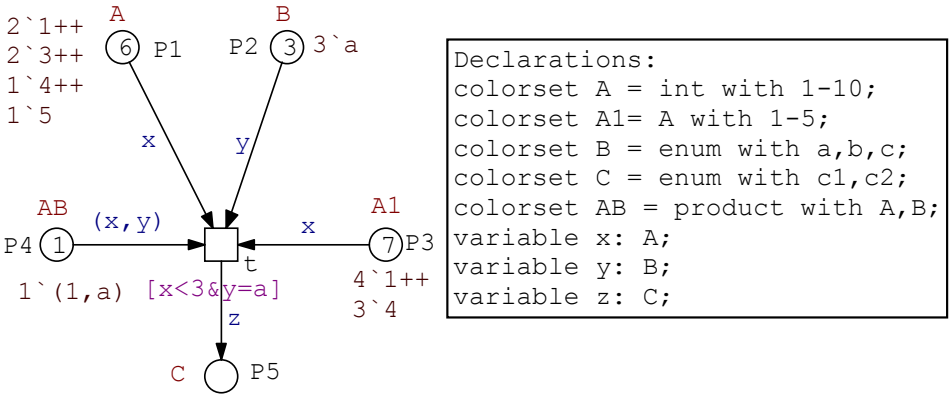

Figure 2: A colored Petri net for demonstrating the computation of transition instances.

The patterns in arc expressions are the basic ones that are used for the computation of enabled transition instances. To obtain them, we can search though all input arc expressions of a transition t.

We can write an input arc expression as c_1‘exp_1++...++c_n‘exp_n, where c_i is the multiplicity of exp_i $(i = 1,2,...,n)$ that has the type of its corresponding input place. If exp_i is a pattern, then we add exp_i to $AS(t)$. For example, for the expression 4‘1++3‘4 in Figure 2, we obtain two constant patterns: 1 and 4.

(2) Patterns in guards.

As the guard of a transition often imposes a rather strong constraint on efficient binding, it is better to consider it early when computing bindings. For this, we adopt the similar approach as that in Kristensen and Christensen (2004). Like Kristensen and Christensen (2004), we consider a guard in the conjunctive form, $g(t) \equiv \wedge_{i=1}^{n} g_i(t)$. For each conjunct, $g_i(t)$, we only consider the following form: $g_{il}(t) = g_{ir}(t)$, where $g_{il}(t)$ and $g_{ir}(t)$ are expressions of patterns, but one of them must be a constant. We add these special expressions to the pattern set $GS(t)$ for a transition t. The advantage of using patterns in guards for binding is obvious. For example, consider the pattern, $y = a$, it directly makes the bindings without the color a, invalid.

(3) Binding variables to color sets.

For a transition t, if there are variables that are not covered by $PS(t)$, we have to bind them to their corresponding color sets. For example, for the variables that only appear in output arcs, we have to assign their color sets to them. In Figure 2, we can see that the variable z is of this case, which has to be bound to its corresponding color set C.

(4) Formal representation of patterns.

We herein give a formal representation of each pattern $S \in PS(t)$ for a transition t as $S = \langle P,E,X,M \rangle$

1. P, the type of the pattern: constant, variable, tuple or guard (a special pattern),
2. E, the expression of the pattern,
3. X, the set of variables in the pattern, and
4. M, the color set relating to the pattern.

For example, Figure 2 has the following patterns before preprocessing.

1. $S_1 = \langle Variable, x, \{x\}, A \rangle$
2. $S_2 = \langle Variable, y, \{y\}, B \rangle$
3. $S_3 = \langle Variable, x, \{x\}, A1 \rangle$
4. $S_4 = \langle Tuple, (x,y), \{x,y\}, AB \rangle$
5. $S_5 = \langle Guard, y = a, \{y\} \rangle$

(5) Optimized pattern set.

In order to further improve the efficiency of computation, we define an optimized pattern set like Kristensen and Christensen (2004). Let $PS(t) = AS(t) \cup GS(t)$ be the pattern set of a transition t. An optimized pattern set $OPS(t)$ for transition t is a set satisfying at least the following conditions:

1. $OPS(t) \subseteq PS(t)$,
2. $GS(t) \subseteq OPS(t)$, and
3. $V(OPS(t)) = V(PS(t))$.

The first item ensures that all members of $OPS(t)$ come from $PS(t)$. The second item states that all guard patterns must be included in $OPS(t)$. The third item ensures that the optimized pattern set should cover all variables that appear in $PS(t)$. Please note that there may be some variables of transition t that are not covered by $PS(t)$, and these variables will be bound to their color sets.

In the preprocessing section below, we will give steps to obtain an optimized pattern set, $OPS(t)$ for a transition t from its pattern set, $PS(t)$, where we will see more conditions that an optimized pattern set should satisfy.

Besides, we collect other expressions that are not in the optimized pattern set to a test set $TS(t)$ for a transition t, which will be used to test if bindings are valid during the binding process. Each expression $S \in TS(t)$ is denoted by a tuple $S = \langle E, X, M \rangle$, where

1. E, the expression,
2. X, the set of variables in the expression, and
3. M, the color set relating to the pattern.

For the expressions in $TS(t)$, we do not leave them until finishing all bindings and then test them. Instead, we will use the partial binding - partial test principle to test an expression in $TS(t)$ once we find that all variables of it have been bound during the partial binding process. This could exclude invalid bindings as early as possible.

4.2.2 Algorithms

In this section, we give a top-level algorithm for computing transition instances, which is illustrated in Algorithm 2. The algorithm inputs the pattern set $PS(t)$ and the test set $TS(t)$ of a transition t, and outputs a complete binding set C.

Algorithm 2: Computing transition instances.

Input: $PS(t), TS(t)$
Output: C
1 $OPS(t)$ = *Preprocess*$(PS(t))$;
2 C = *BindbyPatterns*$(OPS(t), TS(t))$;
3 C = *BindbyColorSets*$(C, TS(t), V(TS(t)) \setminus V(OPS(t)))$;

The algorithm works as follows. First it conducts a preprocessing (Line 1) on $PS(t)$, and obtains an optimized pattern set, $OPS(t)$ by considering some optimization techniques. Afterwards, it executes the *BindbyPatterns* process (Line 2) to assign colors to patterns. Finally, it executes the *BindbyColorSets* process (Line 3) to assign the colors of color sets to the variables that are not contained in the pattern set, $V(TS(t)) \setminus V(OPS(t))$. During these two processes, the algorithm checks whether the guard is satisfied. So, finally we will obtain all valid complete bindings. In the following, we will discuss these three processes in the algorithm in detail. Please see also Liu (2012) for more details.

(1) Preprocessing of a pattern set.

The preprocessing of the pattern set of a transition is very important as it may prune a lot of invalid partial bindings at an early stage, thus improving the efficiency of the computation of transition instances. In the following, we give steps to preprocess a pattern set, which results in an optimized pattern set.

Merging identical patterns. We support the definition of subsets of a color set, which then may be used by different places relating to a transition and which may have identical arc expressions for this transition. This can be seen as identical patterns. Merging these identical patterns means to obtain the intersection of the subsets of a color set, which can reduce the number of the values to which variables are bound before the binding process begins. After merging identical patterns, we obtain:

1. $S_2 = \langle Variable, y, \{y\}, B\rangle$
2. $S_3 = \langle Variable, x, \{x\}, A1\rangle$
3. $S_4 = \langle Tuple, (x,y), \{x,y\}, AB\rangle$
4. $S_5 = \langle Guard, y = a, \{y\}\rangle$

Sorting patterns in terms of the less colors first policy. After that, we can sort the patterns in terms of the less colors first policy. That is, for two patterns S_i and S_j whose corresponding color sets are $S_i.M$ and $S_j.M$, respectively, if $S_i.M \subseteq S_j.M$, S_i will be put ahead of S_j. After sorting the patterns, we further obtain:

1. $S_5 = \langle Guard, y = a, \{y\}\rangle$
2. $S_2 = \langle Variable, y, \{y\}, B\rangle$
3. $S_3 = \langle Variable, x, \{x\}, A1\rangle$
4. $S_4 = \langle Tuple, (x,y), \{x,y\}, AB\rangle$

Removing redundant patterns. For a pattern set, after some patterns have been matched, all variables in the remaining patterns may have been bound, so these remaining patterns are not necessary to finish the binding process. Instead, they can be used to test if the current bindings are valid or not. Hence, we can remove these patterns from the pattern set and add them to the test set. After removing redundant patterns, we finally obtain an optimized pattern set $OPS(t)$ for transition t:

1. $S_5 = \langle Guard, y = a, \{y\}\rangle$
2. $S_3 = \langle Variable, x, \{x\}, A1\rangle$

(2) Binding by matching colors with patterns.

When we obtain an optimized pattern set, we then bind variables to values by matching colors with the patterns in the set. The algorithm is illustrated in Algorithm 3. For each pattern $S \in OPS(t)$, we bind each color in $S.M$ to this pattern and obtain a partial binding set C. For each member of C, we use the test set $TS(t)$ to test if it is valid. That is, for an arc expression in $TS(t)$, we evaluate if its value is in the range of the color set of its corresponding place; for a guard expression in $TS(t)$, we evaluate if it is true. If not, the corresponding partial binding is not valid and removed from C. We call this the partial binding - partial test principle. After this step, we obtain the following partial bindings:

1. $\langle x = 1, y = a, z = \perp\rangle$
2. $\langle x = 2, y = a, z = \perp\rangle$

(3) Binding variables to color sets.

For the variables left, we bind them to the colors of their color sets. The algorithm is similar to Algorithm 3. In both binding algorithms, we use a partial binding - partial test principle to prune invalid bindings as early as possible. For our running example, we finally obtain the complete bindings:

1. $\langle x = 1, y = a, z = c1\rangle$

Algorithm 3: Binding by matching colors with patterns.

```
Input: OPS(t), TS(t)
Output: C
C ← φ;
for each pattern S ∈ OPS(t) do
    Binding each color in S.M to S and obtain a partial set C;
    for each expression S_t ∈ TS(t) do
        Testing if each member of C is valid using S_t;
    endfor
endfor
```

2. $\langle x = 2, y = a, z = c1 \rangle$
3. $\langle x = 1, y = a, z = c2 \rangle$
4. $\langle x = 2, y = a, z = c2 \rangle$

Applying our unfolding tool to this example, we obtain the unfolded Petri net given in Figure 3.

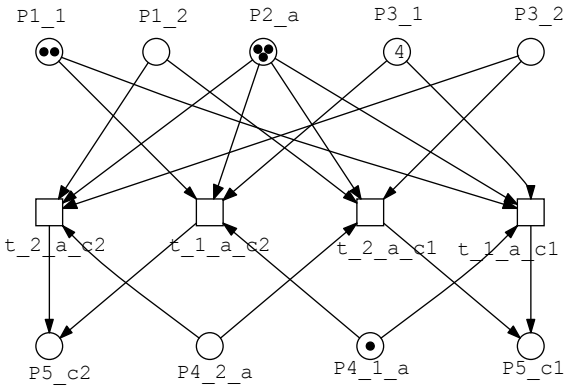

Figure 3: The unfolded Petri net for Figure 2. Due to the removal of some isolated places, there are less tokens shown in this figure than Figure 2.

In summary, the features of this algorithm are as follows:

- The algorithm employs a partial binding - partial test principle, that is, during a partial binding process, if the variables in a test expression have been detected to be fully bound, then we evaluate and test it immediately. As a result, this would not produce any invalid complete binding when the binding process ends.
- If we use subsets of color sets, the pattern matching approach can make us bind variables to subsets of color sets. This prevents a lot of useless bindings from the difference between color subsets and their father color sets.
- A "less colors first" policy is used to order patterns, which can prune invalid bindings during the preprocessing phase.
- The algorithm also makes full use of the guard patterns to disregard false transition instances as early as possible.

In Kristensen and Christensen (2004), the pattern matching approach is used to compute enabled transition instances for animation/simulation, while we use it to find all transition instances for unfolding. Therefore, the purposes are slightly different. Besides this, we additionally employ different optimization techniques to further improve our unfolding approach.

5 EXPERIMENTAL RESULTS

We have implemented this unfolding algorithm in our Petri net tool Snoopy (Rohr, Marwan, and Heiner 2010; Liu 2012), see Liu (2012) for more details about the algorithm.

We now compare the computational efficiency of unfolding before and after using the optimization techniques, especially the constraint satisfaction approach. The model we use is illustrated in Figure 4 (Table 1 gives its declarations.), which simulates the diffusion of chemical signals in a spatial grid to form a chemical gradient. The place P represents the species and the transitions t1 and t2 represent the replication and degradation of the species, respectively. The species can jump to one of its immediate eight neighbors in the $M \times N$ grid, which is modeled by transition t3. This model can be parameterized by the number of rows M and the number of columns N of the grid.

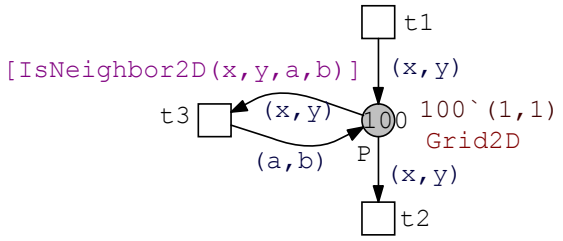

Figure 4: A diffusion model for testing the unfolding efficiency.

Table 1: Declarations for the diffusion model in Figure 4.

Type	**Declaration**
constant	$int : M = 100;$
constant	$int : N = 100;$
colorset	$CD1 = int\ with\ 1-M;$
colorset	$CD2 = int\ with\ 1-N;$
colorset	$Grid2D = product\ with\ CD1 \times CD2;$
variable	$x : CD1,\ y : CD2,\ a : CD1,\ b : CD2;$
function	$bool\ IsNeighbor2D\ (CD1\ x, CD2\ y, CD1\ a, CD2\ b)$
	$\{(a = x \vert a = x+1 \vert a = x-1)\&(b = y \vert b = y+1 \vert b = y-1)\&$
	$(!(a = x\&b = y))\&(a <= D1\&b <= D2)\&(a >= 1\&b >= 1);\}$

The experimental results are shown in Table 2. We can clearly see that the unfolding efficiency substantially improves by use of the optimization techniques, especially the constraint satisfaction approach. For example, for a 100×100 grid, the unfolding before optimization lasts about 933 times longer than after optimization. The unfolding algorithm in this section can tackle large-scale models within a reasonable time.

Table 2: Comparison of the size of the diffusion model in Figure 4 and unfolding runtime*.

Size			Unfolding time (seconds)	
$M \times N$	Places	Transitions	before optimizing	after optimizing
10×10	100	884	4.024	1.024
50×50	2500	24,404	2,057.318	8.532
100×100	10,000	98,804	40,301.145	43.199
200×200	40,000	397,604	$\diamond$	238.429

* done on PC, Intel(R) Xeon(R) CPU 2.83GHz, RAM 4.00GB.

$\diamond$ we did not obtain the result within 24 hours.

6 RELATED WORK

In Mäkelä (2001), unfolding is obtained by an enabling test algorithm, so the places of an unfolded net are all ever marked under the initial marking, that is, all other places that can not get marked under the initial marking are excluded. Maria uses an explicit data structure to represent unfolded Petri nets. Kordon et al. Kordon, Linard, and Paviot-Adet (2006) use data decision diagrams to symbolically represent unfolded nets, but the optimization, such as removal of false guarded transitions, is performed on unfolded nets, which is not efficient.

Compared to those work, our unfolding algorithm adopts an efficient algorithm to compute transition instances, in which some optimization techniques are used to improve the efficiency of unfolding. The disadvantage is that we still adopt an explicit representation of unfolded Petri nets. In the next step, we will concentrate on how to combine the efficient unfolding algorithm with a compact data structure.

7 CONCLUSIONS

In this paper, we have presented an efficient method for unfolding colored Petri nets, in which the constraint satisfaction approach and pattern matching have been used to improve the unfolding efficiency. This method has been implemented in Snoopy, which has been shown to efficiently unfold very large colored Petri nets. A non-trivial case study can be found in Gao, Gilbert, Heiner, Liu, Maccagnola, and Tree (2012).

As a future development of our tool, we will continue to improve the unfolding method to cope with much larger colored Petri nets. We will concentrate on the improvement of the constraint satisfaction approach, e.g., removing the restriction that all variables in a guard are integers. We will also equip this unfolding method with a compact data structure in order to unfold much larger nets and to easily manipulate the unfolded nets. Besides, we will improve the pattern matching approach, e.g., considering more patterns and more optimization techniques.

8 ACKNOWLEDGEMENTS

This work has been supported by Germany Federal Ministry of Education and Research [0315449H] and Natural Science Foundation of China [61273226]. We would like to thank David Gilbert and Wolfgang Marwan for many fruitful discussions, and Mary Ann Blätke, Mostafa Herajy, Christian Rohr, and Martin Schwarick for their assistance in model construction, software development and model checking. We also would like to thank the anonymous referees for their constructive comments.

REFERENCES

Brailsford, S. C., C. N. Potts, and B. M. Smith. 1999. “Constraint Satisfaction Problems: Algorithms and Applications”. *European Journal of Operational Research* 119 (3): 557–581.

Gao, Q., D. Gilbert, M. Heiner, F. Liu, D. Maccagnola, and D. Tree. 2012. “Multiscale Modelling and Analysis of Planar Cell Polarity in the Drosophila Wing”. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*. to appear.

Gecode 2011. “Gecode: An Open Constraint Solving Library”. http://www.gecode.org.

Genrich, H. J., and K. Lautenbach. 1979. “The Analysis of Distributed Systems by Means of Predicate/Transition-Nets”. In *Proc. of the International Sympoisum on Semantics of Concurrent Computation*, edited by G. Kahn, LNCS 70, 123–146: Springer.

Heiner, M., D. Gilbert, and R. Donaldson. 2008. “Petri Nets for Systems and Synthetic Biology”. In *Proc. of the 8th international conference on Formal methods for computational systems biology*, edited by M. Bernardo, P. Degano, and G. Zavattaro, LNCS 5016, 215–264: Springer.

Jensen, K. 1981. “Coloured Petri Nets and the Invariant-Method”. *Theoretical Computer Science* 14 (3): 317–336.

Jensen, K. 1992. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Vol 1, Basic Concepts*. Berlin Heidelberg: Springer.

Jensen, K., and L. Kristensen. 2009. *Coloured Petri Nets*. Springer.

Kordon, F., A. Linard, and E. Paviot-Adet. 2006. "Optimized Colored Nets Unfolding". In *Proc. of the 26th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems*, edited by V. D.-G. Elie Najm, Jean-François Pradat-Peyre, LNCS 4229, 339–355: Springer.

Kristensen, L. M., and S. Christensen. 2004. "Implementing Coloured Petri Nets Using a Functional Programming Language". *Higher-Order and Symbolic Computation* 17 (3): 207–243.

Liu, F. 2012. *Colored Petri Nets for Systems Biology*. Ph. D. thesis, Brandenburg University of Technology Cottbus.

Mäkelä, M. 2001. "Optimising Enabling Tests and Unfoldings of Algebraic System Nets". In *Proc. of the 22nd International Conference on Application and Theory of Petri Nets*, edited by M. K. José Manuel Colom, LNCS 2075, 283–302.

Rohr, C., W. Marwan, and M. Heiner. 2010. "Snoopy - a Unifying Petri Net Framework to Investigate Biomolecular Networks". *Bioinformatics* 26 (7): 974–975.

Tsang, E. P. K. 1993. *Foundations of Constraint Satisfaction*. London and San Diego: Academic Press.

Ullman, J. D. 1998. *Elements of ML Programming*. Prentice-Hall.

AUTHOR BIOGRAPHIES

FEI LIU is associate professor in the Control and Simulation Center at Harbin Institute of Technology. He received a doctorate in control science and engineering from Harbin Institute of Technology and a second doctorate in computer science from Brandenburg University of Technology. His research interests are modeling and simulation, colored Petri nets, systems biology. His email and web addresses, respectively, are liufei@hit.edu.cn and http://homepage.hit.edu.cn/pages/liufei.

MONIKA HEINER is professor and head of the Data Structures and Software Dependability group, Brandenburg University of Technology. Her research interests include modeling and analysis of technical as well as biochemical networks using qualitative and quantitative Petri nets, model checking, and simulation techniques. Over the last 15 years her group at Cottbus has developed a sophisticated toolkit to support the methodology, comprising Snoopy, Charlie, and Marcie. She has served on the editorial board and program committee of several journals and conferences including Journal Natural Computing, BioPPN, CMSB, and PETRI NETS. Her e-mail and web addresses, respectively, are monika.heiner@informatik.tu-cottbus.de and http://www-dssz.informatik.tu-cottbus.de/DSSZ/.

MING YANG is Professor at Harbin Institute of Technology. He received his education at Harbin Institute of Technology. His research interests include system simulation, system control and VV&A. His email address is myang@hit.edu.cn.