Computation of enabled transition instances for colored Petri nets

Fei Liu and Monika Heiner

Department of Computer Science, Brandenburg University of Technology
Postbox 10 13 44, 03013 Cottbus, Germany
{liu, monika.heiner}@informatik.tu-cottbus.de

Abstract. Computation of enabled transition instances is a key but difficult problem of animation of colored Petri nets. To address it in our colored Petri net tool, we give an algorithm for computing enabled transition instances. This algorithm is based on pattern matching. So it first tries to bind tokens to variables covered by patterns. If some variables are not covered by any pattern, the algorithm will bind all the colors in the corresponding color sets to the variables. This algorithm uses the new principle of partial binding - partial test and adopts some optimization techniques for preprocessing to improve efficiency. The principle of partial binding - partial test allows us to test the expressions during the partial binding process so as to prone invalid bindings as early as possible. The preprocessing with optimization techniques not only prunes a lot of invalid potential bindings before the binding begins, but also finds disabled transitions at an early phase.

1 Introduction

Animation is an important technique for getting an intuitive understanding of a Petri net model as it demonstrates the dynamic behavior of the model in a visual way. Nearly all the visual tools for modeling Petri nets provide the animation functionality [Pet10]. For low-level Petri nets, the core of the animation is the scheduling algorithm of the transitions. However, for colored Petri nets, we have to consider another key problem, the computation of enabled transition instances. When checking whether a transition is firable or not at a given marking, we have to assign values to the variables (which are called bindings. A binding of a transition corresponds to a transition instance.) that occur in the arc expressions and the guard of the transition, and then evaluate if it respects the firing rule.

The introduction of colors to Petri nets makes it difficult to compute their firing rules [JKW07]. The efficiency of the animation for large-scale colored Petri nets is mainly determined by the efficiency of the computation of the enabled transition instances, which, however, is a NP-hard search problem because of the expressiveness of colored Petri nets. One possible way is to make an exhaustive search to check all the bindings and then prune the invalid ones, which is

inefficient at all especially if the transitions have many variables, but only a few bindings can fire the transitions.

In this paper, we focus on the problem of the compuatation of enabled transition instances for colored Petri nets. We adopt the same idea given in [KC04], that is, extracting patterns from input arc expressions and guards, then binding the tokens residing on the input places to these patterns, and thus obtaining an enabled binding set. The main contribution of this paper is that we give a more efficient algorithm for our colored Petri net tool [RMH10], [LH10], in which we use a new principle of partial binding - partial test and several heuristics techniques to compute enabled transition instances.

This paper is organized as follows. Section 1 describes the patterns that are used in the computation of enabled transition instances, and discusses how to classify and find patterns. Section 2 discusses the computation process and recalls some concepts. Section 3 gives the computation algorithm. Section 4 discusses some heuristics to optimize the computation process. Section 5 summarizes and compares the related work. Section 6 gives the conclusion.

2 Patterns

We use the same pattern match mechanism as CPN tools [KC04]. A pattern is defined as an expression with variables which can be matched with other expressions to assign the values of variables [KC04]. The difference is that CPN tools are based on the SML, but we do not. We do not employ all the patterns defined in SML [Ull98], but a subset, as we use less data types than CPN tools. The patterns that we use have the following syntactical structure:

$$
\begin{aligned}
Pattern \quad & ::= "Variable" \\
& \quad |"Constant" \\
& \quad |TuplePattern \\
TuplePattern & ::= (Pattern(,Pattern)*)
\end{aligned}
$$

Consider the example illustrated in Figure 1. According to the syntax of the patterns, we can see that (x, y) is a tuple pattern. If we bind the token $(1, a)$ residing on the place $P2$ to the pattern (x, y), after matching, we get an assignment $x = 1$ and $y = a$. This process is called the pattern match.

Pattern matching provides an easy and efficient way to compute enabled transition instances; therefore, to improve the efficiency of the computation, we have to find the patterns and use them to bind the tokens on the places as much as possible. To do so, we have to search all the 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 also can search the guard of the transition t to find the patterns in the guard, denoted by $GS(t)$. These two sets constitute the overall pattern set, $PS(t) = AS(t) \cup GS(t)$, which are used to bind tokens to variables. In the following, we in detail discuss how to find the patterns.

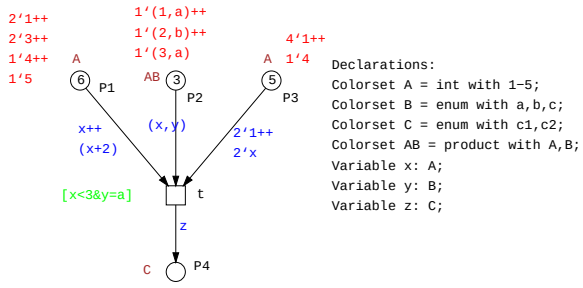

Figure 1. An example to demonstrate the patterns

2.1 Patterns in arcs

The patterns in arcs are the basic patterns that are used for the computation of enabled transition instances. To get them, we have to search though all the input arc expressions in the following two ways.

(1) If an input arc expression is exactly a pattern illustrated above, then we put it in the $PS(t)$. For example, in Figure 1, we can get such kind of pattern: (x, y).

(2) If an input arc expression is a multiset expression, having the form c_1‘$expr_1 + +... + +c_n$‘$expr_n$, where c_i is the multiplicity, and $expr_i$ has the type of its corresponding input place. If $expr_i$ is a pattern, then we add $expr_i$ to the $PS(t)$. For example, for the expression 2‘$1 + +2$‘x in Figure 1, we get two such kinds of patterns: the constant pattern, ’1’ and the variable pattern, x, but for the expression $x + +(x + 2)$, we only get one variable pattern, x, as $x + 2$ is not a pattern. At the same time, we record the multiplicity of the corresponding pattern so as to use them to test bindings once the pattern is used. For example, for the variable pattern x, once we bind a token to it, for instance ’4’ on the place $P3$, we immediately test if there are enough tokens with color ’4’ on the place $P3$. To do so, the invalid bindings can be discarded earlier.

2.2 Patterns in guards

As the guard of a transition imposes often a rather strong constraint on efficient bindings, it is better to consider it early when computing the bindings. For this, we adopt the similar appoach to that in [KC04], but we extend the forms of the guard that are used for binding, moreover we use some forms of guards for test during the binding process.

Like [KC04], we consider the guard in the conjunctive form, $G(t) \equiv \wedge_{i=1}^{n} g_i(t)$. For one of these conjuncts $g_i(t)$, we consider the following forms: $g_{il}(t) = g_{ir}(t)$, where $g_{il}(t)$ and $g_{ir}(t)$ are expressions with constant or variable patterns, but one of them must be a constant. We put these special expressions into the pattern set $GS(t)$. The advantage of using the patterns in the guards for binding is obvious. For example, consider the pattern, $y = a$, it directly makes the bindings relating to tokens without the color, ”a”, invalid.

2.3 Binding color sets to variables

If there are variables that are not covered by $PS(t)$, we have to let them bind to their corresponding color sets, otherwise they can not be bound. For example, for the variables that only appear in the output arcs, we have to bind them to their color sets. In Figure 1, we can see that the variable z is of this case, which has to be bound to the corresponding color set C.

2.4 Optimized pattern set

We herein give a formal representation of each pattern in $PS(t)$, which is a five-tuple $S = \langle P, E, X, M, m\rangle \in PS(t)$, where

- P, the type of the pattern: variable, constant, tuple, or guard,
- E, the expression that the pattern belongs to,
- X, the set of the variables in the pattern,
- M, the initial/current marking of the place that connects the arc whose expression the pattern belongs to, and
- m, the multiplicity of the pattern.

For the pattern of $AS(t) \subseteq PS(t)$, all the components above would be used, but for the pattern of $GS(t) \subseteq PS(t)$, only the first three components P, E, and X would be used. For a constant pattern, X will be always $\{\phi\}$.

For example, the patterns $PS(t)$ in Figure 1 can be formally written as follows:

$S_1 = \langle Variable, x, \{x\}, \{2`1, 2`3, 1`4, 1`5\}, 1\rangle$
$S_2 = \langle Tuple, (x, y), \{x, y\}, \{1`(1, a), 1`(2, b), 1`(3, a)\}, 1\rangle$
$S_3 = \langle Constant, 1, \{\phi\}, \{4`1, 1`4\}, 2\rangle$
$S_4 = \langle Variable, x, \{x\}, \{4`1, 1`4\}, 2\rangle$
$S_5 = \langle Guard, y = a, \{y\}\rangle$

In order to improve the efficiency of computation, we define an optimized pattern set. Let t be a transition, $PS(t) = AS(t) \cup GS(t)$ is the pattern set. An optimized pattern set $OPS = \{S_i | 1 \leq i \leq N\}$ for transition t is a set satisfying the following conditions:

1. $V(OPS(t)) = V(PS(t))$, where $V(PS(t))$ represents the set of all the variables in $PS(t)$,
2. $OPS(t) \subseteq PS(t)$,
3. $\forall S_i \in GS(t)$, $S_i \in OPS(t)$.

The first item ensures that the optimized pattern should cover all the variables that are covered by $PS(t)$. Please note that there may be some variables of transition t that are not covered by $PS(t)$, and these variables will be bound by their color sets. The second item ensures that all the elements in the $OPS(t)$

come from $PS(t)$. The third item states that all the guard patterns must be included in the $OPS(t)$. In the preprocessing section below, we will give the 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 set $NS(t)$, whose elements are non-multiset expressions. If an expression is a multiset expression, then we divide it first into a set of non-multiset expressions. Each expression in $NS(t)$ is denoted by a tuple $S = \langle E, X, M \rangle$, where

- E, the expression,
- X, the set of the variables in the expression,
- M, the initial marking of the place that connects the arc the expression belongs to.

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

For example, in Figure 1, if the variable x in $P1$ is bound by the value $1, 3, 4, 5$, we can immediately evaluate and test the expression $x + 2$. As a result, at this moment we can exclude the partial bindings $x = 3$,$x = 4$, and $x = 5$, as the place $P1$ has no enough tokens for these bindings.

3 Binding process

In this section, we recall the binding process and some definitions, which are adapted from [KC04].

In order to evaluate the arc expressions and the guard of a transition t, the variables relating to the transition (denoted by $V(t)$) must be bound by values (tokens). A binding of a transition is written in the form: $\langle v_1 = c_1, ...v_n = c_n \rangle$, where $v_i \in V(t)$, c_i is the color value belonging to a corresponding color set, $i = 1, 2, ..., n$.

Matching a token of an input place to a pattern would usually only bind to a subset of the variables of the transition t. For example, consider the transition t in Fig. 2, matching the token (1,a) to the pattern (x, y) will bind the variable x to 1, and y to a, but it will not bind any value to the variable z. So the concept of the partial binding is present, which means that a partial binding of a transition is a binding in which not all variables of the transition are bound by values. In the following, we use the $PartialBinding(p, c)$ to denote the partial binding by matching a pattern p with a token value c. If they are not matched, $PartialBinding(p, c) = \perp$.

In order to get a complete binding, we have to gradually merge the partial bindings. For example, matching the pattern x and the tokens of $P1$ yields the

following four partial bindings:

$\langle x = 1, y = \perp, z = \perp \rangle$
$\langle x = 3, y = \perp, z = \perp \rangle$
$\langle x = 4, y = \perp, z = \perp \rangle$
$\langle x = 5, y = \perp, z = \perp \rangle$

Matching the pattern (x, y) and the tokens of P2 yields the following three partial bindings:

$\langle x = 1, y = a, z = \perp \rangle$
$\langle x = 2, y = b, z = \perp \rangle$
$\langle x = 3, y = a, z = \perp \rangle$

If we merge them, we obtain the following two partial bindings:

$\langle x = 1, y = a, z = \perp \rangle$
$\langle x = 3, y = a, z = \perp \rangle$

The merging of two partial bindings relates to the concept of the compatible bindings. Two binding b_1 and b_2 are compatible (written as $Compatible(b_1, b_2)$), if and only if

$$\forall v \in V(t) : b_1(v) \neq \perp \wedge b_2(v) \neq \perp \Rightarrow b_1(v) = b_2(v)$$

For two compatible partial bindings b_1 and b_2, the combined partial binding (written as $Combine(b_1, b_2)$) satisfies:

$$b(v) = \begin{cases} b_1(v) \text{ if } b_1(v) \neq \perp \\ b_2(v) \text{ if } b_2(v) \neq \perp \\ \perp \quad \text{otherwise} \end{cases}$$

Based on those defintions above, the merging of two partial binding sets B_1 and B_2 (written as $Merge(B_1, B_2)$) is defined as:

$$Merge(B_1, B_2) = \{Combine(b_1, b_2) | \exists (b_1, b_2) \in B_1 \times B_2 : Compatible(b_1, b_2)\}$$

4 An algorithm for computing enabled transition instances

In this secition,we first give a top-level algorithm for computing enabled transition instances, which is illustrated in Algorithm 1. The algorithm inputs the pattern set $PS(t)$, and the non-pattern expression set of the transition, $NS(t)$, and outputs a complete binding set C.

The algorithm works as follows. First the algorithm performs a preprocessing (line 1) on $PS(t)$, and obtains an optimized pattern set, $OPS(t)$ by considering some optimization techniques. Afterwards, the algorithm executes the

BindbyPatterns process (line 2) to bind tokens residing on the places to the patterns. After that, the algorithm executes the BindbyColorSets process (line 3) to bind color sets to the variables that are not contained in the pattern set, $V(NS(t))\backslash V(OPS(t))$. During this process, the algorithm checks whether the guard is satisfied and whether the input places have sufficient tokens. So, finally we get all valid complete bindings. In the next sections, we will continue to discuss the three processes of the algorithm in more details.

Algorithm 1: An algorithm for computing enabled transition instances.

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

4.1 Preprocessing of the pattern set

The preprocessing of the pattern set is very important as it may prune a lot of invalid partial bindings in advance and find if the transition can be enabled as early as possible, thus improving the efficiency of computation of enabled transition instances. In this section, we give the steps to preprocess the pattern set and as a result obtain an optimized ordered pattern set.

(1)Testing multiplicity.

We begin the preprocessing of the pattern set with multiplicity testing. During this step, we can discard the tokens in the current marking that do not contribute to the valid bindings. This is performed by checking whether the number of tokens with the same color is greater than or equal to the muliplicity of the pattern. For a constant pattern, if this is evaluated to false, we immediately stop the preprocessing process, and directly disable this transition. If true, we can now remove the constant patterns from the binding pattern set, as we will not use it any longer for succedent processes. For a variable or tuple pattern, if this is evaluated to false, we will remove these tokens from the current marking set. If the current marking set becomes empty, we stop the preprocessing process, and directly disable this transition.

The algorithm is illustrated in Algorithm 2, which works as follows. The algorithm executes a loop for each pattern in the pattern set $PS(t)$. If a pattern is a constant pattern, the multiplicity of the constant pattern is checked with the tokens of the constant color. Here $S_i.M_i\langle c\rangle$ represents the number of the tokens with the color "c". If this is evaluated to false, the transition is determined not to be enabled (lines 2-6). If a pattern is a variable or tuple pattern, for each color in the initial marking, the multiplicity is tested. The tokens will be removed if the testing is false. If the current marking set of the pattern becomes empty, the transition is determined not to be enabled (lines 8-17).

After the multiplicity testing for the example in Figure 1, we get the following pattern set, where pattern S_3 is removed.

$S_1 = \langle Variable, x, \{x\}, \{2`1, 2`3, 1`4, 1`5\}, 1\rangle$
$S_2 = \langle Tuple, (x, y), \{x, y\}, \{1`(1, a), 1`(2, b), 1`(3, a)\}, 1\rangle$
$S_4 = \langle Variable, x, \{x\}, \{4`1\}, 2\rangle$
$S_5 = \langle Guard, y = a, \{y\}\rangle$

Algorithm 2: Multiplicity testing.

```
   Input: PS(t)
   Output: OPS(t)
 1 for each pattern S_i ∈ PS(t) do
 2     if S_i is a constant pattern then
 3         c ← S_i.E_i;
 4         if S_i.M_i⟨c⟩ < S_i.m_i then
 5             transition t is not enabled;
 6         endif
 7     endif
 8     if S_i is an variable or tuple pattern then
 9         for each color c ∈ S_i.M_i do
10             if S_i.M_i⟨c⟩ < S_i.m_i then
11                 S_i.M_i ← S_i.M_i\{S_i.M_i⟨c⟩};
12             endif
13         endfor
14         if S_i.M_i is empty then
15             transition t is not enabled;
16         endif
17     endif
18 endfor
```

(2)Merging identical patterns.

Usually, there exist several identical patterns (identical expressions) for a transition. Merging them can remove invalid partial bindings as much as possible before the binding begins. The algorithm is illustrated in Algorithm 3. To merge two identical patterns, for example, S_i and S_j, $i \neq j$, we need to get their colors that have tokens, denoted by C_i and C_j (lines 1-2), respectively. We calculate the merged colors by $C_k = C_i \cap C_j$ (line 3). If C_k is not empty, we build a new pattern S_k, where M_k stores the merged color with the multiplicity being 1 and m_k is set to 1 (lines 4-10). At the same time, we remove the old patterns S_i and S_j and add a new pattern S_k to the pattern set. If the set C_k is empty, we can directly set the transition disabled.

For example, Figure 1 has two identical patterns: S_1 and S_4. The colors with current tokens are $\{1, 3, 4, 5\}$ and $\{1\}$, respectively, and the merged color is $\{1\}$.

Algorithm 3: Merging identical patterns.

Input: S_i, S_j
Output: S_k

```
1  C_i ← S_i.M_i;
2  C_j ← S_j.M_j;
3  C_k ← C_i ∩ C_j;
4  if C_k is not empty then
5      S_k.P_k ← S_i.P_i;
6      S_k.E_k ← S_i.E_i;
7      S_k.X_k ← S_i.X_i;
8      S_k.M_k ← C_k;
9      S_k.m_k ← 1;
10 endif
11 if C_k is empty then
12     transition t is not enabled ;
13 endif
```

So we remove the patterns, S_1 and S_4 and add a new pattern, S_{14}. Now the patterns for Figure 1 become:

$S_2 = \langle Tuple, (x, y), \{x, y\}, \{1`(1, a), 1`(2, b), 1`(3, a)\}, 1\rangle$
$S_{14} = \langle Variable, x, \{x\}, \{1`1\}, 1\rangle$
$S_5 = \langle Guard, y = a, \{y\}\rangle$

(3)Sorting patterns in terms of the less different tokens first policy [Cae96].

After that, we can sort the patterns in terms of the less different tokens first policy that will be discussed in detail later. For example, after sorting, the binding patterns in Figure 1 become:

$S_5 = \langle Guard, y = a, \{y\}\rangle$
$S_{14} = \langle Variable, x, \{x\}, \{1`1\}, 1\rangle$
$S_2 = \langle Tuple, (x, y), \{x, y\}, \{1`(1, a), 1`(2, b), 1`(3, a)\}, 1\rangle$

After finishing the preprocessing, we finally get an optimized pattern set $OPS(t)$, which will be used as the input of the following algorithm.

4.2 Binding by matching tokens and patterns

In this section, we describe a key component of the algorithm for the computation of enabled transition instances (illustrated in Algorithm 4), binding by matching tokens residing on the places and patterns in the set, $OPS(t)$, which is based on the algorithm in [KC04].

The algorithm executes a loop to handle each member of the pattern set $OPS(t)$. Lines 4-9 consider the case of the guard patterns, in which the right

hand side of the guard is matched against the left hand side of it. Lines 10-18 consider the case of matching the tokens in the current marking and the arc expression patterns. Lines 19-31 test if each partial binding is valid using the non-pattern set $NS(t)$. For an expression of $NS(t)$ whose variables are fully bound, if it is a guard and is evaluated to be false for a partial binding, then the partial binding is invalid. If the expression is an arc expression and can not get enough tokens by evaluating it with a partial binding, then the partial binding is also invalid.

Compared to the algorithm in [KC04], our algorithm has the following distinguished features:

- The biggest difference is that our algorithm employs the partial binding - partial test principle, that is, during a partial binding process, if the variables in a non-pattern 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.
- The overall algorithm considers the case of the variable binding to the color set, as this case may be encountered in our colored Petri nets, which will be discussed in the next section.

We still use the example in Figure 1 to demonstrate how the algorithm above works. For the first loop, the guard pattern $y = a$ is processed, and let 'a' bind to y. Then the pattern S_{14} is processed, and let x be bound by '1'. After that, the non-pattern expression $x < 3$ begins to work as it finds that the variable x has been bound and keeps the binding '1' to x. We continue to bind $(1, a)$ to the pattern (x, y) and merge them with existing bindings. After these steps, we get the following partial bindings.

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

4.3 Binding colors of color sets to variables

When the variables are not contained by all the patterns, they have to be bound to colors of their color sets. The algorithm is illustrated in Algorithm 5. The algorithm works as follows. The algorithm executes a loop for each variable $v \in V(NS(t)) \backslash V(OPS(t))$, which stores all the variables that have to be bound by the color sets. Lines 3-9 bind the colors to the variables. Here $c(v)$ represents the color set of variable v. Lines 10-22 test if the expressions in $NS(t)$ satisfy the guard or have sufficient tokens in the corresponding places.

We continue to apply this algorithm to the example in Figure 1. Here, we bind the color set C with colors, $c1$ and $c2$, to the variable z. Then we get the following complete bindings.

$\langle x = 1, y = a, z = c1 \rangle$
$\langle x = 1, y = a, z = c2 \rangle$

Algorithm 4: An algorithm for matching tokens and patterns.

Input: $OPS(t) = \{S_i | 1 \leq i \leq N\}$, $V(t)$
Output: C

$C \leftarrow \phi$;
for *each pattern* $S_i \in OPS(t)$ **do**
  $C^{'} \leftarrow \phi$;
  `// binding`
  if $S_i \equiv g_{il} = g_{ir}$**then**
    $b^{'} \leftarrow PartialBinding(g_{il}, g_{ir})$;
    if $b^{'} \neq \perp$ **then**
      $C \leftarrow C \cup \{b^{'}\}$;
    endif
  endif
  if $S_i \in AS(t)$ **then**
    for *each colored token* $c \in S_i.M_i$ **do**
      $b^{'} \leftarrow PartialBinding(S_i.E_i, c)$;
      if $b^{'} \neq \perp$ **then**
        $C^{'} \leftarrow C^{'} \cup \{b^{'}\}$;
      endif
    endfor
    $C \leftarrow Merge(C, C^{'})$;
  endif
  `// testing`
  for *each expression* $S_k \in NS(t)$ **do**
    if $V(S_k) \subseteq V(C)$ **then**
      for *each binding* $b \in C$ **do**
        if $S_k.E_k$ *is a guard expression and* $S_k.E_k\langle b\rangle$ *is false* **then**
          $C = C \backslash \{b\}$;
        endif
        if $S_k.E_k$ *is an arc expression and* $S_k.E_k\langle b\rangle > S_k.M_k\langle c\rangle$ **then**
          $C = C \backslash \{b\}$;
        endif
      endfor
      $NS(t) \leftarrow NS(t) \backslash \{S_k\}$;
    endif
  endfor
endfor

Algorithm 5: An algorithm for binding colors of color sets to variables.

Input: C, $TS(t)$, $NS(t)$, $V(NS(t))\backslash V(OPS(t))$
Output: C
`// binding`
for *each variable* $v \in V(NS(t))\backslash V(OPS(t))$ **do**
  $C^{'} \leftarrow \phi$;
  for *each color* $c \in c(v)$ **do**
    $b^{'} \leftarrow PartialBinding(v, c)$;
    if $b^{'} \neq \perp$ **then**
      $C^{'} \leftarrow C^{'} \cup \{b^{'}\}$;
    endif
  endfor
  $C \leftarrow Merge(C, C^{'})$;
  `// testing`
  for *each expression* $S_k \in NS(t)$ **do**
    if $V(S_k) \subseteq V(C)$ **then**
      for *each binding* $b \in C$ **do**
        if $S_k.E_k$ *is a guard and* $S_k.E_k\langle b\rangle$ *is false* **then**
          $C = C\backslash\{b\}$;
        endif
        if $S_k.E_k$ *is an arc expression and* $S_k.E_k\langle b\rangle > S_k.M_k\langle c\rangle$ **then**
          $C = C\backslash\{b\}$;
        endif
      endfor
      $NS(t) \leftarrow NS(t)\backslash\{S_k\}$;
    endif
  endfor
endfor

5 Optimization Techniques

In this section, we briefly summarize some of the optimization techniques that we use to improve the efficiency of the computation of enabled transition instances.

(1)Partial binding - partial test.

As described above, we collect all the arc or guard expressions that are not covered by the pattern set. We do not leave them until finishing all complete bindings and then test them. Rather, we will test them once we find that all the variables of them have been bound during the partial binding process. For example, in Figure 2, the optimized pattern set is x, y and z. If we do not use this policy, we would first get $20 \times 30 \times 40$ complete bindings, then test these bindings by evaluating $x+1$ and $y+1$ and then get 480 valid bindings. However, if we use this policy, x is first bound and 20 partial bindings are gotten. After that the expression $x+1$ is tested, and the valid bindings for x are now 3. Then y is bound, and the partial bindings for x and y become 90. When the expression $y+1$ is tested, the partial bindings become 12. Finally, the variable z is bound, and the final complete bindings are gotten, whose number is 480. Obviously, using this policy usually reduces the amount of computation greatly.

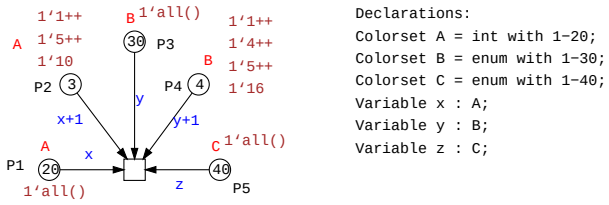

Figure 2. An example to demonstrate the policy of partial binding - partial test.

(2) Less different tokens first policy [Cae96].

As can be easily noticed and analyzed, the information of the tokens residing on different places can affect the efficiency of computation of enabled transition instances. For example, in Figure 1, for transition t, if we bind first x to the tokens of $P1$, we have 4 bindings, but if we bind the tokens in $P2$ to x first, we get 2 bindings. That is to say, the binding order of variables is quite different in efficiency; therfore, we can optimize the binding order of the patterns. We use the less different tokens first policy, which has been given by [Cae96].

(3) Multiplicity test.

When finding patterns, we also record the multiplicities of the patterns. We use them to test if the places contain enough tokens for enabling before the binding begins, which already is reflected in the Algorithm 2.

(4) Merging identical patterns.

Merging the same patterns before binding is more efficient than binding a pattern and then testing another pattern during binding. In the algorithm presented

above, we consider the merging of identical patterns during the preprocessing phase. This heuristics is very useful when there are many identical patterns for a transition and the tokens for each pattern are notably different.

All the heuristics have been used in our algorithm, which can be seen in different parts in Algorithm 1-5.

6 Related work

In this section, we describe and compare some related work concerning the computation of enabled transition instances.

Mäkelä [Mak01] used a unification technique to calculate enabled transition instances for the algebraic system nets that are in fact another kind of high-level Petri nets, which gave a different idea on finding enabled bindings.

Sanders [San00] considered the calculation of enabled binding as a constraint satisfaction problem. He imposed strong constraints on the form of arc expressions, only considering the form $n`exp$, which is impossible for nearly all the colored Petri nets.

Gaeta [Cae96] studied the enabled test problem of Well-Formed Nets, and gave some heuristics for determining the binding elements, i.e., less different tokens first policy, which are very useful for the efficiency of calculation of enabled transition instances.

Mortensen [Mor01] described data structure and algorithms used in the CPN tools. He used the locality principle to dicover enabled transitions rather than calculating all the transition each time. He also discussed how to optimize the binding sequences.

Kristensen et al. [KC04] gave a pattern reference algorithm for enabled binding calculation of CPN tools. We also adopt that idea to design our binding algorithm, but compared their algorithm, our algorithm considers more optimization techniques.

In our work, we take into account the main idea of [KC04] and also some ideas of [Mak01] and [Cae96], but comapared with all the previous work, we adopt a new principle, partial binding - partial test, and consider more optimization techniques to improve the efficiency of computing enabled transition instances for colored Petri nets.

7 Conclusions

In this paper, we present an algorithm for computation of enabled transition instances for colored Petri nets. This algorithm uses the principle of partial binding - partial test and adopts some optimization techniques for preprocessing. The principle of partial binding - partial test allows us to test the expressions during the partial binding process so as to prone invalid bindings as early as possible. The use of optimization techniques prunes invalid potential bindings before the binding begins, and also finds the disable transitions at an early

phase. Among them, the less different tokens first policy allows variables to have less bindings, the multiplicity test excludes insufficient tokens before the binding begins and the merging of identical patterns avoids repeated bindings for identical patterns. All these techniques contribute to the improvements of efficiency.

This algorithm can realize an efficient computation of enabled transition instances for large-scale colored Petri nets. At the same time, we are adapting this algorithm to unfold colored Petri nets, which will improve the efficiency of unfolding of colored Petri nets and thus simulation of colored stochastic Petri nets. In the future, we will investigate more optimization techniques to further improve the efficiency.

Acknowledgments

This work has been supported by the Modeling of Pain Switch (MOPS) program of Federal Ministry of Education and Research (Funding Number: 0315449H).

References

[Cae96] R. Gaeta: Efficient Discrete-Event Simulation of Colored Petri Nets. IEEE Transactions on Software Engineering. 22(9), 629-639 (1996)

[JKW07] K. Jensen, L. M. Kristensen, L. Wells: Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. International Journal on Software Tools for Technology Transfer. 9, 213-254 (2007)

[KC04] L. M. Kristensen, S. Christensen: Implementing Coloured Petri Nets Using a Functional Programming Language. Higher-Order and Symbolic Computation. 17(3), 207-243 (2004)

[LH10] F. Liu, M. Heiner: Colored Petri nets to model and simulate biological systems. International Workshop on Biological Processes and Petri Nets (BioPPN). (2010)

[Mak01] M. Mäkelä: Optimising Enabling Tests and Unfoldings of Algebraic System Nets. International Conference on Application and Theory of Petri Nets, LNCS, 2075, 283-302 (2001)

[Mor01] K. H. Mortensen: Efficient Data-Structures and Algorithms for a Coloured Petri Nets Simulator. 3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'01), 57-74 (2001)

[Pet10] Petri Nets World. http://www.informatik.uni-hamburg.de/TGI/PetriNets. (2010)

[San00] M. J. Sanders: Effecient Computation of Enabled Transition Bindings in High-Level Petri Nets. IEEE International Conference on Systems, Man and Cybernetics. 3153-3158 (2000)

[RMH10] C. Rohr, W. Marwan, M. Heiner: Snoopy - a unifying Petri net framework to investigate biomolecular networks. Bioinformatics. 26(7): 974-975 (2010)

[Ull98] J. D. Ullman: Elements of ML Programming. Prentice-Hall. (1998)