Efficient Implementation of Prioritized Transitions for High-level Petri Nets

Michael Westergaard[*] and H.M.W. (Eric) Verbeek

Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands
{m.westergaard,h.m.w.verbeek}@tue.nl

Abstract. Transition priorities can be a useful mechanism when modeling using Petri nets. For example, high-priority transitions can be used to model exception handling and low-priority transitions can be used to model background tasks that should only be executed when no other transition is enabled. Transition priorities can be simulated in Petri nets using, e. g., inhibitor arcs, but such constructs tend to unnecessarily clutter models, making it useful to support priorities directly.
Computing the enabling of transitions in high-level Petri nets is an expensive operation and should be avoided. As transition priorities introduce a nonlocal enabling condition, at first sight this forces us to compute enabling for all transitions in a highest-priority-first order, but it is possible to do better. Here we describe our implementation of transition priorities in CPN Tools 3.0, where we minimize the number of enabling computations. We describe algorithms for executing transitions at random, useful for automatic simulation without user interactions, and for maintaining a set of known enabled transitions, useful for interactive user-guided simulation. Experiments show that using our algorithms we can execute $4-7$ million transitions a minute for real-life models and more than 20 million transitions a minute for other models, a significant improvement over the $1-5$ million transitions a minute possible for simpler algorithms.

1 Introduction

Prioritized transitions can be of use when modeling using Petri nets. For example, one can give a transition high priority to force it occur before other transitions if it is enabled, which is useful for handling exceptions, by letting the exception handler have higher priority than transitions handling usual cases. One can assign a transition a lower priority to prevent it from occurring unless no other transitions are enabled, which is useful for implementing a scheduler that should only be executed when all interesting tasks are unable to proceed.

In this paper we are concerned with efficient implementation of simulation of high-level Petri net models with transitions with priorities as well as efficient enabling updates. The described algorithms are implemented in CPN Tools 3.0 [4].

[*] This research is supported by the Technology Foundation STW, applied science division of NWO and the technology program of the Dutch Ministry of Economic Affairs.

Priorities can be implemented using inhibitor arcs or any construction which serves the same purpose (by adding inhibitor arcs from places which have arcs to transitions with higher priority), but it is beneficial to support them directly in an implementation to reduce clutter in models. Furthermore, a direct implementation makes it possible to make enabling computation more efficient than implementations relying on general constructs.

Enabling computation of high-level Petri nets, such as coloured Petri nets (CPNs) supported by CPN Tools, is computationally expensive. To alleviate this, tools can implement algorithms to avoid having to compute the enabling of transitions too often. For example, if the goal is just to randomly execute transitions, there is no need to compute the enabling for all transitions – as soon as an enabled transition is found, it can be executed. By using caching of enabling status and structural properties of the model, the number of enabling computations can be reduced even further. We extend such an algorithm to handle prioritized transitions by modifying the step where transitions are picked at random to instead pick transitions at random in a highest-priority-first order, so enabled transitions with higher priority are executed before transitions with lower priority. We present an algorithm and data structures supporting this.

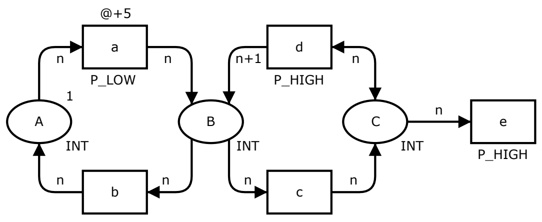
When a tool shows a model during simulation in a graphical user interface, the enabling status of transitions is typically shown to allow users to pick between enabled transitions for guided simulation. To do this, the enabling state of all transitions must be computed. It is not necessary to recompute the enabling status of all transitions after each execution of a transition, though. We only need to recompute the enabling of transitions for which it has potentially changed, and we can give a static over-approximation of this which roughly says that if a transition is connected to a place also connected to the executed transition, its enabling may have changed. We present an even better approximation in Sect. 2. This approximation is not good enough if allowing priorities, as the execution of a transition may enable or disable a transition with the highest priority, thereby causing unconnected transitions to be disabled or enabled. We present an algorithm for over-approximating the set of transitions influenced by this.

The remainder of this paper is structured as follows: in the next section, we present background material and in Sect. 3 we present algorithms for efficiently finding a random enabled transition taking priorities into account, and for efficiently updating the enabling status of all transitions. In Sect. 4, we conclude and provide directions for future work.

2 Background

In this section we briefly introduce coloured Petri nets using an example and describe an efficient algorithm for enabling computations. The algorithm is described in further detail in [6, 11].

A Petri net is a bipartite graph, where the nodes are partitioned into *places* and *transitions*. Places are usually drawn as circles or ellipses and transitions



	Dependency Set	Disable Set
a	{b, c}	{a}
b	{a}	{b, c}
c	{d, e}	{b, c}
d	{b, c}	∅
e	∅	{e}

Fig. 1: A simple coloured Petri net.

are typically drawn as rectangles or lines. In Fig. 1, we see a Petri net with 3 places (A–C) and 5 transitions (a–e). Places can contain *tokens* and represent the state of the system. In coloured Petri nets tokens are distinguishable and can have a value from the type of the place they reside on. In Fig. 1, all places have type INT (integer) and the only token is a single one with the value 1 residing on place A. Places and transitions are connected using directed arcs. Arcs describe preconditions and postconditions for transitions and are inscribed with *expressions* which may contain typed *variables*. For example, the arc from the place A to the transition a has inscription n, which is a variable of type INT. We allow double arcs as an abbreviation of an arc in both directions with the same expression. In the example, we have a double arc between C and d.

A transition of a CPN model is *enabled* if there exists a *binding* of values to all variables on arcs surrounding it so all *input places* (places with arcs to the transition) contain all tokens dictated by evaluation of the corresponding arc expressions. A transition with a binding is called a *binding element*. In Fig. 1, the transition a is enabled in the binding n = 1 as A contains a single token with value 1. An enabled binding element can be *executed*, consuming tokens on input places, and producing new tokens on *output places* (places with an arc from the transition). When a is executed in the binding n = 1, it consumes the single token 1 from A and produces a new token on the place B.

Tokens can have an attached time stamp, and are only available when a global clock reaches a value larger than or equal to their associated time stamp. Transitions can have execution times, shown as @+ annotations. In Fig. 1 only transition a has an execution time, namely 5. If a transition with an execution time is executed, all produced tokens shall have a time stamp that is the current global time plus the execution time of the transition. For example, if a is executed at time 2 in the binding n = 1, the token on A is consumed and a new token with value 1 and a time stamp of 7 $(2 + 5)$ is produced on B. Transitions b and c are not enabled before the global time reaches 7.

We can at any time partition transitions into enabled and *disabled* (i. e., not enabled) transitions. Computing enabling is a complex task, so CPN Tools implements an algorithm which uses heuristics to find bindings in a way that is fast in practice (see [6, 11] for details), but even using this technique, computation takes considerable time. If we just want to execute a random transition, there is no need to compute the enabled state of all transitions; we randomly pick a transition, check whether it is enabled, and if it is we execute it in a random binding. If the transition is not enabled, we cannot execute it and just continue with the next transition. This strategy, although better than computing the enabled state of all transitions, throws away information, namely that a transition is known to be disabled. As we execute transitions, transitions may move from disabled to enabled and vice versa, but only some transitions can move when certain other transitions are executed. For example, executing transition a in Fig. 1 can never alter the enabled state of e as the places they are connected do not intersect. We can exploit this to do a more efficient enabling computation. For example, during an execution of the model in Fig. 1, we first try executing transition e, and find it is disabled. We then try executing a and succeed. Now, there is no need to recompute the enabling of transition e as the enabling of this transition cannot be altered by the execution of a. The *dependency set* of a transition t captures this and is the set of all transitions that may be enabled by executing t. This can be computed as all transitions for which an output place of t is an input place (not counting places connected with double arcs to t). Similarly, the *disable set* is the set of transitions that can become disabled by executing a transition. We have summarized the dependency sets and disable sets of transitions of Fig. 1 in the table in the right side of the figure.

When we deal with timed models, we can have an additional state for each transition: it is not enabled right now, but may become enabled at a later stage when time has increased. This leads us to partitioning transitions into three sets: the Disabled, the Unknown, and the MaybeReady. The first are transitions known to be disabled, the second are transitions for which the enabled state is not yet known, and the last is for transitions that are not enabled but may become so at a later point in time. We note, we do not have a set for enabled transitions, as we immediately execute a transition if it is found to be enabled.

An algorithm for random execution of transitions is shown as Algorithm 1. The algorithm works in time epochs, where Unknown contains all transitions that are possibly enabled in the current epoch and MaybeReady transitions that may become enabled in a later epoch. We start with all transitions in the MaybeReady set. We start an epoch by increasing the time of the epoch to the least time stamp any transition of MaybeReady can be enabled (l. 5) and move all transitions that can be enabled at that time to Unknown (l. 6). As long as transitions can be enabled at the current epoch (l. 7), we pick one randomly (l. 8). We assume the existence of a function Enabled which returns one of three values: enabled, disabled, and maybe_ready_at(n), where the last value not only indicates that the transition is not enabled now, but also provides an estimate (n) of when the transition may be enabled. If a transition is not enabled it is moved to either Disabled or MaybeReady. If the picked transition is enabled (l. 9), we execute it, add its dependency set to Unknown and remove its dependency set from Disabled and MaybeReady (ll. 10–13). If a transition is disabled or maybe_ready_at(n), we move it from Unknown to either Disabled or MaybeReady. The inner while loop (ll. 7–19) executes all transitions enabled in a single epoch, and the outer loop

Algorithm 1 Algorithm for enabling computation for timed models.

```
 1: Unknown $\leftarrow \emptyset$
 2: Disabled $\leftarrow \emptyset$
 3: MaybeReady $\leftarrow \{0\} \times Transitions.all$
 4: while MaybeReady $\neq \emptyset$ do
 5:   $IncreaseTime$(MaybeReady)
 6:   Unknown $\leftarrow RemoveLeast$(MaybeReady)
 7:   while Unknown $\neq \emptyset$ do
 8:     Pick any $t \in$ Unknown
 9:     if $Enabled(t) =$ enabled then
10:       $Execute(t)$
11:       Unknown $\leftarrow$ Unknown $\cup\ DependencySet(t)$
12:       Disabled $\leftarrow$ Disabled $\setminus DependencySet(t)$
13:       MaybeReady $\leftarrow$ MaybeReady $\setminus DependencySet(t)$
14:     else if $Enabled(t) =$ disabled then
15:       Unknown $\leftarrow$ Unknown $\setminus \{t\}$
16:       Disabled $\leftarrow$ Disabled $\cup \{t\}$
17:     else if $Enabled(t) =$ maybe_ready_at($n$) then
18:       Unknown $\leftarrow$ Unknown $\setminus \{t\}$
19:       MaybeReady $\leftarrow$ MaybeReady $\cup \{(n, t)\}$
```

(ll. 4–19) executes all epochs. The algorithm terminates when (if) there are no more transitions in MaybeReady and Unknown, so all transitions are in Disabled.

The operations needed for Unknown are to add all transitions, pick a random element, add a set of elements not already contained, and remove a particular element. This can be efficiently implemented by enumerating all transitions from $0, 1, \ldots, |\mathit{Transitions}| - 1$, storing them in an array A of size $|\mathit{Transitions}|$ and adding a pointer *last* pointing to the position after the last element of Unknown. Add all transitions can be performed by setting all entries of the array to their index ($A[i] := i$) and setting the last pointer to $|\mathit{Transitions}|$, picking a random element corresponds to drawing a random number $r \in \{0, 1, \ldots, \mathit{last} - 1\}$ and returning the value $A[r]$. Adding a set of not already contained elements consists of adding the elements to positions $\mathit{last}, \mathit{last} + 1, \ldots$ and incrementing *last* accordingly. Removal of an element consists of swapping the element with the last one and decrementing the *last* counter. By combining the get random element and remove operations (this is possible by moving lines 15 and 18 up after line 8 in algorithm 1 and adding any transition to its own dependency set) we can perform picking in constant time and insertion in time linear in the number of elements we insert. We call this data-structure a *RandomSet* and use it to implement Unknown. For MaybeReady we insert each transition with a weight, namely the time at which it is earliest enabled, and only remove elements with the least weight, which naturally makes us implement MaybeReady as a priority queue, allowing us to add and remove elements in time $\log |\mathit{Transitions}|$ for each element. Storing the position of elements in the priority queue also allows us to remove internal elements (needed to remove the dependency set of a transition) in the same time. We never read from the Disabled set, and hence do not need to explicitly represent it. It is only shown to make the algorithm clearer (and can be computed as the complement of Unknown and MaybeReady anyway).

3 Algorithm

In this section we develop an algorithm for fast random execution of transitions for timed coloured Petri net models using priorities. We also develop algorithms for operations useful for graphical tool support for simulation and modification of such models. We also present experimental performance data of the algorithms on both toy examples and several real-life models [5, 10, 12] developed in other contexts.

When we talk about coloured Petri nets with priorities, we assign to each transition an expression evaluating to a nonnegative integer indicating the priority of the transition. Priorities considered here are global and cannot depend on the binding of the transition; we later discuss other priority concepts. We can think of the priority as a function assigning to each transition t a numeric priority, $Priority(t)$[1]. At any point in time, a transition is *preenabled* if all tokens required for executing the transition are available (also taking time into account). Only the transitions with the highest priority among the preenabled transitions are actually enabled. In the model in Fig. 1, we have assigned priorities to a, d, and e, namely P_LOW, P_HIGH, and P_HIGH respectively. We assume we have defined constants such that P_LOW < P_NORMAL < P_HIGH and that transitions without a priority inscription have priority P_NORMAL. Here we just use three levels of priorities, but our algorithm handles an arbitrary number, p.

3.1 Random Execution

Our goal is to randomly execute transitions quickly, adhering to the priorities. We use algorithm 1 as a basis. Extending this algorithm to handle priorities is simple: instead of picking transitions completely randomly in line 8, we pick them randomly among the transitions with the highest priority.

A way to implement this efficiently is to use a priority queue of *RandomSet*s for Unknown. That is, for each priority, we have a *RandomSet* like earlier. We can get nearly the same time guarantees for this implementation as for the simple *RandomSet*. We can get and remove an element with the lowest priority in time $\log p$ where p is the number of different priorities used (3 in the example). This extra cost (compared with constant time previously) is incurred as we may have to rebalance the priority queue. The time required to add elements to Unknown depends on the implementation. If we use no auxiliary data structure, we may need to search the priority queue for the correct *RandomSet* to insert into, i. e., insertion takes time p for each element. We can keep a search tree mapping priorities to *RandomSet*s, lowering the insertion time to $\log b$ for each element. We

[1] In our implementation we actually use a low number as high priority, but our explanation shall not reflect that for improved readability.

can also maintain an array mapping priorities to *RandomSet*s, bringing down insertion time for each element to constant time. This, however, comes at the cost of using memory linear in the highest numeric value of a priority. Finally, we could store the *RandomSet*s in a hash-map mapping priorities to the corresponding *RandomSet*, which allows constant time look-up and using space linear in p but using a larger constant than using the array. Unless p is large, which one we use in practice has little influence on the speed of the algorithm. We do not expect p to be larger than 10 in practice. We call any such implementation a *PriorityRandomSet* and obtain an algorithm for random execution of transitions adhering to priorities by using algorithm 1 with a *PriorityRandomSet* implementation for Unknown. In CPN Tools we use the implementation using an array as index into the priority queue to impose as little overhead in execution time as possible (as we do not have to traverse a pointer-based data-structure, but just look up a value in an array). We notice that if $p = 1$ all representations collapse to the same as the implementation not taking priorities into account, as we never have to rebalance the priority queue and search in the auxiliary data-structure pointing into the priority queue.

3.2 Random Enabling Computation

If we want to compute enabling for all transitions, this is easily done: sort the transitions according to priority and compute enabling highest-priority first. When an enabled transition is found, we stop computing enabling for transitions with lower priority.

Sometimes this may not be desired, however. For example, if a user is only looking at part of a model, the tool may only need to compute enabling for parts of the transitions (the visible ones) to show enough information to the user. Furthermore, we wish our algorithm to also efficiently handle maintenance of a set of enabled transitions, which can be done without recomputing enabling for all transitions. Hence, we seek an algorithm for computing the enabling of a random transition as efficiently as possible but still adhering to priorities. Furthermore, we want the algorithm to efficiently compute enabling of subsequent transitions, i.e., the main focus is on amortized running time.

When we want to compute enabling for a transition, we need to know whether any transition with higher priority is enabled. If we are computing enabling for more than one transition, part of this work may be reusable. For example, in Fig. 1, if we want to compute the enabling for a, b and c, we first need to establish the enabling of d and e as their priorities are higher. Naturally, this computation only needs to be done once, even if we first compute enabling for a and b and in a subsequent call (without executing any transition) for c.

The idea of our enabling computation algorithm is to use the data-structures Unknown, Disabled, and MaybeReady from algorithm 1. Ignoring priorities for the time being, we update the data-structures as in the inner loop in lines 8–19 of algorithm 1, except we do not execute transitions, and hence do not do operations based on dependency sets (ll. 11–13). The adapted algorithm is algorithm 2. The algorithm only checks if transitions are enabled at the current time, and needs

Algorithm 2 Algorithm for checking enabling without priority.

```
1: proc $CheckEnabling(t)$ is
2:   if $t \notin$ Unknown then
3:     return false
4:   else
5:     if $Enabled(t) =$ enabled then
6:       return true
7:     else if $Enabled(t) =$ disabled then
8:       Unknown $\leftarrow$ Unknown $\setminus \{t\}$
9:       Disabled $\leftarrow$ Disabled $\cup \{t\}$
10:      return false
11:    else if $Enabled(t) =$ maybe_ready_at$(n)$ then
12:      Unknown $\leftarrow$ Unknown $\setminus \{t\}$
13:      MaybeReady $\leftarrow$ MaybeReady $\cup \{(n, t)\}$
14:      return false
```

Algorithm 3 Simple algorithm for checking enabling with priority.

```
1: SortedTransitions $\leftarrow$ $PrioritySort(Transitions.all)$
2: proc $CheckEnablingPriority(t)$ is
3:   for all $t' \in$ SortedTransitions do
4:     if $Priority(t') > Priority(t)$ then
5:       if $CheckEnabling(t')$ then
6:         return false
7:     else
8:       return $CheckEnabling(t)$
9:   return $CheckEnabling(t)$
```

somebody external to increase time and move elements from MaybeReady to Unknown when Unknown becomes empty. We note that we could use a bit-array of entries in Unknown to retain constant time look-up in line 2 and maintain the performance of all other operations.

To also handle priorities, we can use algorithm 2 as a subprocedure to compute preenabledness, i. e., whether a transition is enabled when ignoring priorities. A simple way to do this is shown as algorithm 3; we sort all transitions according to priority and process them highest-priority-first until we reach t. If we find a preenabled transitions with higher priority than t, we return false. If we do not find a preenabled transition with higher priority than t we return the preenabledness of t. We assume that we traverse the transitions in a highest-priority-first order in line 3, and have introduced early termination as soon as the condition in the if statement in line 4 no longer holds. This is acceptable, as enabling of a transition with the same or lower priority cannot affect the enabling of t. If a transition is in Disabled it does not only mean it is disabled, but the stronger condition that it is not even preenabled.

We choose to compute SortedTransitions based on all transitions instead of based on Unknown (which would also work), as we then can precompute this for a given model, making the execution *CheckEnablingPriority* independent of this computation.

When this algorithm is called repeatedly, it only calls *Enabled* for each transition with higher priority than the first preenabled transition or the transition with the lowest priority (whichever is higher) plus once for each call (as soon as a transition is marked as disabled, it is no longer in Unknown). The number of calls to *CheckEnabling* is the sum of the numbers of transitions with higher priority than each of the transitions, which can be quadratic in the number of transitions (if each transition has a unique priority and only the one with the highest priority is enabled). A call to *CheckEnabling* is cheap as long as it does not result in a call to *Enabled*, but if we want to limit the number of calls here, we could introduce an approximation of the priority of the first enabled transition in SortedTransitions. As long as we have not found an enabled transition, this estimate is $-\infty$, and it is set to the priority of the first enabled transition as soon as one is found. We also maintain an index of the last transition checked for enabling, so we do not check transitions already verified to be disabled again, thus skipping calls to *CheckEnabling*. The resulting algorithm allows the same bound on the number of calls to *CheckEnabling*, namely one for each call plus one for each transition with priority higher than or equal to the first preenabled transition or the transition with the lowest priority (whichever is higher).

In CPN Tools we have implemented the version of the algorithm shown in algorithm 3, i. e., without estimation of the priority of the first enabled transition. This is done because CPN models rarely have more than a few transitions (50-100), so traversing SortedTransitions imposes a very small overhead.

3.3 Enabling Set Maintenance

Often we wish to run a random simulation and show intermediate results to users. We therefore wish to merge algorithm 1 (augmented to handle priority as described earlier) and 3 into a single algorithm sharing Unknown, Disabled, and MaybeReady in a way that makes it possible to do random simulation as well as to check enabling of selected transitions with as few calls to *Enabled* as possible.

We can get by with few changes, as we do not have to change algorithm 3 as long as we faithfully maintain Unknown, Disabled, and MaybeReady. The best place to call *CheckEnablingPriority* is between lines 9 and 10 in algorithm 1, as this is the only place we know we have increased the time sufficiently that a transition is enabled.

We can call *CheckEnablingPriority* for all the transitions we are interested in, but that is not necessary. The reason we wish to avoid that in CPN Tools is that this incurs a communication overhead, as the GUI and the simulator are separate processes. This can be relevant for any tool if the number of transitions is high, as enabling checks no longer depend directly on the total number of transitions in the model.

If we disregard priority, the enabling status can only have changed for transitions in the dependency set of the last transition executed, but when taking priority into account, things are not as simple, as the enabling of a transition

Algorithm 4 Algorithm for random simulation using priority while maintaining the set of all enabled transitions.

```
1: SortedTransitions $\leftarrow$ $PrioritySort(Transitions.all)$
2: Unknown $\leftarrow \emptyset$
3: Disabled $\leftarrow \emptyset$
4: MaybeReady $\leftarrow \{0\} \times Transitions.all$
5: Enabled $\leftarrow \emptyset$
6: while MaybeReady $\neq \emptyset$ do
7:    $IncreaseTime$(MaybeReady)
8:    Unknown $\leftarrow$ $RemoveLeast$(MaybeReady)
9:    while Unknown $\neq \emptyset$ do
10:      Enabled $\leftarrow \{t' \in$ Unknown $\mid CheckEnablingPriority(t')\}$
11:      while Enabled $\neq \emptyset$ do
12:         Pick any $t \in$ Enabled
13:         $Execute(t)$
14:         Unknown $\leftarrow$ Unknown $\cup\ DependencySet(t)$
15:         Disabled $\leftarrow$ Disabled $\setminus DependencySet(t)$
16:         MaybeReady $\leftarrow$ MaybeReady $\setminus DependencySet(t)$
17:         Enabled $\leftarrow$ Enabled $\setminus DisableSet(t)$
18:         New $\leftarrow$
               $\{t' \in DependencySet(t) \cup DisableSet(t) \mid CheckEnablingPriority(t')\}$
19:         if New $\neq \emptyset$ then
20:            if $\exists t1 \in$ New$, t2 \in$ Enabled$.Priority(t1) > Priority(t2)$ then
21:               Enabled $\leftarrow$ New
22:            else
23:               Enabled $\leftarrow$ Enabled $\cup$ New
```

with higher priority than all currently enabled transitions will disable them. We know that all transitions that have remained in the Disabled set since last time are still there (i. e., if a transition was not preenabled before and not in the dependency set of the transition executed last, it is still not preenabled). We also know that only if new transitions become enabled do we have to disable other transitions. If we disable all enabled transitions and do not enable any with the same or higher priority, we need to consider the preenabled transitions or increase the model time. We can thus compute the enabled transitions using algorithm 4. Here, we maintain a set Enabled in addition to the ones we already maintain. This set contains all enabled transitions and aside from initialization (l. 10), which takes place initially and whenever we need to increment time because no more transitions are enabled, we only ever update it according to the dependency set and the disable set of executed transitions (ll. 17, 20, 23, 25, and 27). This algorithm can be made interactive by pausing and asking the user for a transition to execute in line 12.

3.4 Extension to Other Priority Concepts

While the priority concept detailed until now, assigning to each transition a fixed numeric priority, is in line with standard statically prioritized Petri nets [3], it is not very high-level. For example, we cannot assign higher priority to a specific task in a folded net (such as assigning d priority depending on n in Fig. 1). In [2] a dynamic priority concept is adopted. This allows priorities to depend on the entire marking of the model. In our opinion, this is way too centralized to easily comprehend and specify.

With CPNs the natural way to assign dynamic priorities to transitions is using general expressions just like guards or arc expressions. Although this is a natural priority concept for coloured Petri nets, we have chosen not to adopt it. The problem is that when the priority depends on the binding of transitions, we have to compute every preenabled binding of every transition, subsequently compute the priorities for each preenabled binding element, and finally pick one with highest priority. Although this is conceptually nice and consistent with the other inscriptions, it leads to dramatically decreased performance. The remainder of this section is dedicated to extending the static notion of priority presented hitherto while compromising performance as little as possible.

Using a Subset of Variables in Priorities CPN Tools, in addition to restricting the number of times enabling of a transition is called, also partitions all variables surrounding a transition into *binding groups*. A binding group is a subset of the variables surrounding a transition that can be assigned values independently of all other variables (i. e., if two bindings of a transition are enabled, the binding obtained by replacing the value of all variables in a binding group in the first binding by the binding of the same variables from the second binding is also enabled). Variables that occur in the same arc expression or the guard must be within the same group. By requiring that all variables occurring in the priority expression come from the same binding group, we can just compute all possible bindings of variables in that binding group instead of for all variables of the transition.

It is always correct to combine two binding groups into one, so in the worst case transitions only have one binding group, forcing us to compute all enabled bindings of all transitions anyway. We believe, though, that only a small subset of variables will be used in the priority, typically just a process ID or an independent priority on a place. In those cases, we can compute the priority for all possibilities of the binding group, schedule the transition with all resulting priorities, and execute it like before. This approach requires that we dynamically add/remove transitions to SortedTransitions and compute all partial bindings for the binding group comprising variables of the priority inscription for all transitions in $DependencySet(t) \cup DisableSet(t)$ whenever we execute t. We have not implemented this, as we believe that users may inadvertently build nets that take prohibitively long to simulate, and many interesting cases can be solved by splitting a transition into several, one for each desired priority. For example, if we want d in Fig. 1 to execute with low priority if $n > 5$, we can just make two copies of d, one with high and one with low priority, and give the highly prioritized one a guard n<=5 and the one with low priority a guard n>5.

Scoped Priorities It is often useful to be able to use scoped priorities. For coloured Petri nets with hierarchy [7], this means that we would like to say that a given transition has higher or lower priority than all other transitions on the same page (module), but it should not necessarily be considered less important than enabled transitions on other pages. This is useful for implementing multiple schedulers (e. g., for two separate but connected systems) and for handling errors in multiple places without preempting unconnected operations (e. g., handle stale messages on different communication channels). Furthermore, making priorities local makes it much easier to use modular analysis techniques.

We can implement scoped priorities by running any of the algorithms for each page in isolation (using algorithm 1 with a *PriorityRandomSet* for random simulation, algorithm 3 if we want to compute enabling, and using algorithm 4 to maintain a set of enabled transitions). We introduce a new top loop which randomly selects a page to execute a step on. We have not implemented this in CPN Tools as we have not found an elegant way of having both scoped and global priorities coexist in an easy-to-understand manner. An added advantage is that flattening of a hierarchical CPN model remains a purely syntactical operation, where we would otherwise have to consider interplay of local priorities.

3.5 Experimental Validation

We have compared the algorithm for random non-interactive simulation (algorithm 3) developed in this section with a naive algorithm just evaluating enabling in a highest-priority-first order and an algorithm computing all enabled bindings for all transitions before selecting a transition to execute. Our findings are summarized in Table 1. We have executed the algorithms with three toy examples shipping with CPN Tools: the dining philosophers, a distributed database, and a simple stop-and-wait protocol. We have also tested with three industrial examples: a protocol for routing in mobile ad-hoc networks (ERDP) [10], the DYMO protocol for route discovery in mobile ad-hoc networks [5], and a protocol for operational support for workflow execution (OS) [12]. All models have been developed independently of the implementation of priorities and hence represent natural examples and not pathological examples designed to put our algorithms in a good light. We also show extended versions of the operational support protocol, modeling more details of the system, and a version with all extensions disabled in the model which is behaviorally equivalent to the original model, but has more transitions to consider. We have made large and small versions of the OS model; the small model (OS) only has a few participants, making the size of the model suitable for state-space analysis, and the large version (OS') has many more participants and can only be analyzed using simulation. The three versions of OS use priorities while the other models do not (as the others were developed before CPN Tools supported priorities). For each model, we show the complexity as reflected by the number of modules, the number of transition instances, and the number of place instances. We also show the number of place instances after merging all places in a port/socket assignment relationship as

Table 1: Experimental results.

Model	Instances				Transitions/minute		
	Pages	Transitions	Places		All Bindings	Priority Sorted	Algorithm 3
Philosophers	1	3	3	(3)	$3.21 \cdot 10^6$	$12.39 \cdot 10^6$	$22.19 \cdot 10^6$
Database	1	5	9	(9)	$5.01 \cdot 10^6$	$12.20 \cdot 10^6$	$17.26 \cdot 10^6$
Protocol	1	5	10	(10)	$2.81 \cdot 10^6$	$7.42 \cdot 10^6$	$21.18 \cdot 10^6$
ERDP	14	16	65	(11)	$0.50 \cdot 10^6$	$1.20 \cdot 10^6$	$3.97 \cdot 10^6$
DYMO	15	25	55	(18)	$0.75 \cdot 10^6$	$2.53 \cdot 10^6$	$4.14 \cdot 10^6$
OS	25	42	134	(25)	$2.44 \cdot 10^6$	$4.01 \cdot 10^6$	$6.64 \cdot 10^6$
Extended OS 1	31	50	164	(36)	$1.68 \cdot 10^6$	$2.70 \cdot 10^6$	$6.26 \cdot 10^6$
Extended OS 2	31	50	164	(36)	$2.06 \cdot 10^6$	$3.29 \cdot 10^6$	$6.05 \cdot 10^6$
OS'	25	42	134	(25)	$0.43 \cdot 10^6$	$1.24 \cdot 10^6$	$4.21 \cdot 10^6$
Extended OS 1'	31	50	164	(36)	$0.37 \cdot 10^6$	$0.94 \cdot 10^6$	$4.11 \cdot 10^6$
Extended OS 2'	31	50	164	(36)	$0.44 \cdot 10^6$	$1.21 \cdot 10^6$	$4.32 \cdot 10^6$

well as places in a fusion group in parentheses. We show the number of transitions we can execute for each model and algorithm. These tests are performed by running CPN Tools 3.0.3 on a computer with a 2.7 GHz Core i7 Sandy Bridge dual core CPU (using one core only). All tests were run for 5 minutes and the average has been reported. The tests repeatedly execute a model and resets the scheduler structures Unknown and MaybeReady as well as the state of the model when no more transitions are enabled. We have not evaluated the performance of algorithm 4 as it incurs a large communication overhead due to the architecture of CPN Tools. We have not compared with a baseline simulator without priority for two reasons: First, we only have an implementation with an old version of the simulator, which for independent reasons is much slower, and second, the performance when a model does not use priorities is exactly the same whereas the performance of a model using priorities is incomparable, as the lack of support for priorities may cause the model to be able to reach states not reachable when using priorities, thus comparing different behavior.

We see that using our optimized algorithm, the toy examples can execute around 20 million transitions a minute. The largest gain is from not computing all bindings (though we may compute enabling for all transitions). The reason is that toy examples often have few transitions but a lot of enabled bindings for each. Thus, computing enabling of all transitions is not very expensive (as this terminates early in our implementation) but computing all bindings is. For real-life models, we see that performance of the simple algorithms significantly decreases as the number of transitions grow. The performance of our improved algorithm is roughly constant at $4 - 7$ million transitions a minute. When looking at the results for the large and small versions of OS, we see the improved algorithm handles the large model almost without any penalty whereas the simple algorithms are orders of magnitude slower. This is again because we have more enabled bindings of each transition. The penalty of the optimized algorithm for real-life models stems from the fact that each transition is much more complex (calls functions and does more advanced matching on the input tokens consumed) and therefore takes longer to execute.

4 Conclusion and Future Work

We have presented algorithms for performing fast simulation of coloured Petri nets with priorities. We have given details on performing fast random simulation of CPN models with statically prioritized transitions. We have given an algorithm for performing fast amortized enabling check of statically prioritized transitions without assuming that enabling is tested in a specific order. Additionally, we have given an algorithm which can be used for maintaining a set of enabled transitions during simulation, providing fast user-guided simulation with interactive feedback. We have implemented all these features in CPN Tools 3.0 [4], and our experiments show we are able to execute $4 - 7$ million transitions a minute for real-life models and more than 20 million transitions for other models. This is an improvement over the $1 - 5$ million transitions a minute for simple algorithms.

We have considered and sketched algorithms for extending our algorithms to handle dynamically prioritized transitions and for scoped priorities. We have chosen not to implement these, first, because dynamic priorities are prone to introducing performance bottlenecks, and, second, because we have not been able to introduce scoped priorities in a way that nicely coexists with global priorities. As scoped priorities can be useful, it would be nice to consider this in more detail.

We have not been able to find any published work concerning efficient simulation of models with priorities. We think this is because the problem only really becomes important with high-level Petri nets, where enabling computation is several orders of magnitude more computationally expensive than for low-level net classes. We have experimented with using the scheduling algorithm 1 for Place-Transition Petri nets, but have not been able to make it outperform a simple algorithm trying transitions at random without the extra book-keeping. The complexity of enabling computation for high-level nets stems from the fact that the high-level nature makes modelers more prone to generating many tokens, and that these tokens are not equal, so in the worst case we have to try all combinations of tokens. Papers treating simulation of low-level nets with priorities often translate nets with priorities to nets without, e. g., [3]. Work exists on translating high-level nets with priorities to nets without [8] or for doing distributed simulation with priorities present [9]. Here, we instead focus on efficient algorithms for direct simulation of high-level nets, which allows us to do optimizations not possible in a parallel or distributed setting.

Our algorithms can also be used for analysis by means of state-space exploration. As for simulation, analysis can be done by translating to equivalent models without priority [3, 8] and for low-level nets additionally by means of static analysis or restriction [1, 2]. This is probably because a strong feature of low-level nets is that this kind of analysis is possible. For high-level nets, analysis is usually only possible by means of state-space exploration or simulation, making fast simulation algorithms more important. Our current state-space tool implementation in CPN Tools is tuned toward breadth-first traversal, so unless we store the $\mathsf{Disabled}$ and $\mathsf{MaybeReady}$ sets for each state, we cannot make use of this information without recomputing it from scratch each time. As the state-space tool of CPN Tools comprises a lot of legacy code, we decided that instead of doing this directly, we would use the algorithm for computing enabling (algorithm 3) instead. It would be interesting to look into algorithms for improving state space analysis by computing transitions according to their priority and also to investigate ways of using the Disabled and MaybeReady sets during state-space generation.

References

1. F. Bause. On the analysis of Petri nets with static priorities. *Acta Informatica*, 33:669–685, 1996. 10.1007/BF03036470.
2. F. Bause. Analysis of Petri Nets with a Dynamic Priority Method. In *Proc. of ATPN'97*, volume 1248 of *LNCS*, pages 215–234. Springer, 1997.
3. E. Best and M. Koutny. Petri net semantics of priority systems. *TCS*, 96(1):175–215, 1992.
4. CPN Tools webpage. Online: `cpntools.org`.
5. K.L. Espensen, M.K. Kjeldsen, and L.M. Kristensen. Modelling and Initial Validation of the DYMO Routing Protocol for Mobile Ad-Hoc Networks. In *ATPN'08*, volume 5062 of *LNCS*, pages 152–170. Springer, 2008.
6. T.B. Haagh and T.R. Hansen. Optimising a Coloured Petri Net Simulator. Master's thesis, Dept. of Computer Science, Aarhus University, 1994.
7. K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems.* Springer, 2009.
8. H. Klaudel and F. Pommereau. A Concurrent and Compositional Petri Net Semantics of Preemption. In *Proc. of IFM'00*, volume 1945 of *LNCS*, pages 318–337. Springer, 2000.
9. M. Knoke, F. Kühling, A. Zimmermann, and G. Hommel. Towards Correct Distributed Simulation of High-Level Petri Nets with Fine-Grained Partitioning. In *Proc. of ISPA'04*, volume 3358 of *LNCS*, pages 64–74. Springer, 2004.
10. L.M. Kristensen and K. Jensen. Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad-hoc Networks. In *Integration of Software Specification Techniques for Application in Engineering*, volume 3147 of *LNCS*, pages 248–269. Springer, 2004.
11. K.H. Mortensen. Efficient Data-Structures and Algorithms for a Coloured Petri Nets Simulator. In *Proc. of 3rd CPN Workshop*, volume 554, pages 57–74. DAIMI PB, 2001.
12. M. Westergaard and F.M. Maggi. Modelling and Verification of a Protocol for Operational Support using Coloured Petri Nets. In *Proc. of ATPN'11*, LNCS. Springer, 2011.