Efficient Implementation of Simulation of Prioritized Transitions for High-level Petri Nets

M. Westergaard* and H.M.W. 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, exception handling can be modeled using high-priority transitions and background tasks can be modeled using low-priority transitions. Although transition priorities can be simulated in Petri nets using, e. g., inhibitor arcs, such constructs tend to unnecessarily clutter models. Hence, it is useful to support priorities directly. The main problem with transition priorities is that they introduce a nonlocal enabling condition. At first sight, this forces us to compute enabling for all transitions in a highest-priority-first order. However, this should be avoided whenever possible as computing whether transitions in high-level Petri nets are enabled is an expensive operation. This paper shows that we can minimize the number of enabling computations, and hence can do better. Experiments show that using the algorithms presented in this paper we can execute approximately 10 times as many transitions a second as is possible for simpler algorithms. This holds for both toy examples and real-life models, though the gain is often larger for real-life models.

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. This 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. Prioritized transitions are also useful for analysis, as we can assign internal transitions a higher priority, thereby preempting other transitions and reducing concurrency, leading to smaller state spaces.

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. While simulation is closely related to verification, e.g., by means

* 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.

of implicit or explicit state-space generation, we here focus on random simulation, which is different, as state-space generation assumes that we compute all enabled bindings of all enabled transitions, where we are only interested in one. Furthermore, we deal with high-level Petri nets, where enabling computation is expensive, so we cannot justify computing all enabled bindings of one or all transitions as the computation time of doing so is too large. The described algorithms are implemented in CPN Tools 3.0 [6] and newer. 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. This means that 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 the marking of any input place may have have changed, the enabling of the transition may have changed. We present a better approximation in Sect. 2. This approximation is not enough if we allow 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 investigate other notions of transition priority, and in Sect. 5, we conclude and provide directions for future work. An earlier version of this work has been pub-

lished as [16]. This paper improves presentation and provides formal definitions of introduced concepts, provides more details about some of the theoretical improvements previously only described in text, shows how the algorithms can be used to efficiently implement interactive switch between model modification and simulation, updates experiments with developments in CPN Tools 3.2 (higher speed for improved algorithms and a user-accessible implementation of the naive algorithm), and adds further discussion of extension of the algorithms to handle more advanced priority concepts.

2 Background

In this section we introduce coloured Petri nets and describe an efficient algorithm for enabling computations. The algorithm is described in further detail in [8,14]. We also adapt two notions of priority for low-level Petri nets from [3] and [2] to coloured Petri nets.

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 are typically drawn as rectangles or bars. In Fig. 1, we see a Petri net with 3 places (A–C) and 5 transitions (a–e). Places contain multi-sets of *tokens* which represent the state of the system. In coloured Petri nets tokens have values 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 place A to 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 requested 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

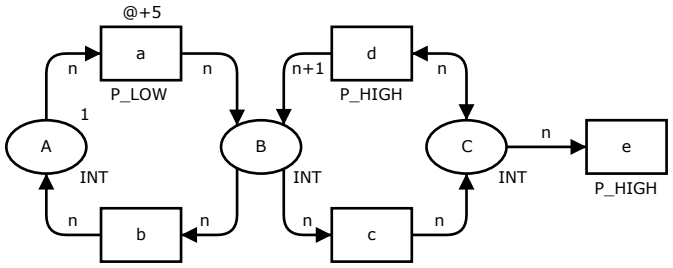

Fig. 1: A simple coloured Petri net.

value 1. The same transition is not enabled in the binding n = 2 (A contains no token with value 2), and the transition b is not enabled in any binding (as B contains no tokens). An enabled binding element can be *executed*, consuming the specified 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, enabling b and c, both in the binding n = 1, and disabling a.

Coloured Petri nets have a notion of time. Tokens can have an attached time stamp, and are only available for consumption by transitions 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 get 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.

The following definition summarizes a coloured Petri net as defined in ISO/IEC 15909 part 1 combined with the time extensions of [10]. In the definition we use the notion of a type for transitions instead of assignments of values to variables; the graphical representation of Fig. 1 corresponds to a high-level Petri net graph of [9], which can be shown to be equivalent to Def. 1. In the following we use the intuition of the figures but the formal definition in Def. 1. A timed CPN model assumes we a given a time domain, TD, which can be any totally ordered set, but typically is the set of non-negative reals or natural numbers. We provide the definition in general terms, but in the following we assume that time stamps are non-negative real numbers. All markings additionally associate a time stamp to each value and the forward incidence function produces tokens with a time stamp. We omit the word timed from the name and just refer to the models as coloured Petri nets or CPN models.

Definition 1 (Coloured Petri net (Def. 5.1 and 5.2 in [9][1] and Def. 11.4 in [10])). *A **coloured Petri net** is a tuple, $\mathcal{CPN} = (P, T, D, Type, TD, Pre, Post, M_0)$, where*

- *P is a finite set of **places**,*
- *T is a finite set of **transitions** such that $P \cap T = \emptyset$,*
- *$D \neq \emptyset$ is a finite set of non-empty **types**,*
- *$Type : P \cup T \rightarrow D$ is a **type function** assigning a type to each place and transition,*
- *TD is the **time domain**, a totally ordered set,*
- *$TRANS = \{(t, m) \mid t \in T, m \in Type(t)\}$ is the set of all **binding elements**,*
- *$\mathbf{N}^{PLACE} = \mathbf{N}^{\{(p,g) \mid p \in P, g \in Type(p)\}}$ is the set of all **untimed markings**,*
- *$\mathbf{N}^{PLACE \times TD} = \mathbf{N}^{\{(p,g,ts) \mid p \in P, g \in Type(p), ts \in TD\}}$ i is the set of all **markings**,*

[1] In this paper we will use the term coloured Petri net rather than high-level Petri net as used in [9].

- $Pre : TRANS \to \mathbf{N}^{PLACE}$ *is the backward incidence function assigning to each incoming arc an* ***arc annotation****,*
- $Post : TRANS \times TD \to \mathbf{N}^{PLACE \times TD}$ *is the forward incidence function assigning to each outgoing arc an arc annotation, such that for all binding elements* (t, m)*, places* p*, values* $g \in Type(p)$*, and time stamps* $ts, ts' \in TS$*,* $Post(t, m, ts)(p, g, ts') > 0 \implies ts' \geq ts$*, and*
- $M_0 \in \mathbf{N}^{PLACE \times TD}$ *is the* ***initial marking****.*

We note that only the forward incidence function depends on time and it ensures that time stamps of produced tokens are never in the past.

In the following, we use the notion of pre-sets and post-sets of a transition t, defined as $\bullet t = \{p \in P \mid \exists m \in Type(t), g \in Type(p).Pre(t, m)(p, g) > 0\}$ and $t\bullet = \{p \in P \mid \exists m \in Type(t), g \in Type(p).Post(t, m)(p, g) > 0\}$, i.e., the set of places with an arc to and from the given transition.

The state of a coloured Petri net is given by a marking of the places, which is a multi-set, $M \in \mathbf{N}^{PLACE \times TD}$. A transition is enabled at a time stamp ts if all tokens required by Pre are present and have a smaller time stamp than ts. If this is the case, we can execute t by removing tokens described by Pre and producing new tokens described by $Post$:

Definition 2 (Enabling and occurence of transitions (Def. 5.3.1 and 5.4 in [9] and Def. 11.6 in [10])). *A binding element,* $(t, m) \in TRANS$*, is* ***enabled*** *at time* $ts \in TD$ *in marking* $M \in \mathbf{N}^{PLACE \times TD}$ *if for all* $p \in P$ *and* $g \in Type(p)$*, we have that* $Pre(t, m)(p, g) \leq \sum_{ts' \in TS, ts' \leq ts} M(p, g, ts')$*. If* (t, m) *is enabled in* M *at time* ts*, it may* ***occur*** *and lead to a marking* M'*. This is written* $M \xrightarrow{(t,m,ts)} M'$*, where* M' *is defined by* $M' = M - M'' + Post(t, m)$ *where* $M'' \subseteq M$ *such that for all* $p \in P$ *and* $g \in Type(p)$ *we have that* $Pre(t, m)(p, g) = \sum_{ts' \in TS, ts' \leq ts} M''(p, g, ts')$*.*

We note that we do not distinguish between tokens with the same value and different time stamps prior to the time of checking enabling and during execution just remove random tokens with time stamp less than or equal to the time a binding element is executed.

2.1 Automatic Simulation of Coloured Petri Nets

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* (i. e., not enabled).

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 [8, 14] for details), but even using this technique, computation takes considerable time. In the following, we assume that we can compute enabling of a transition and execute it in a random enabled binding.

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, we saw that executing transition a from Fig. 1 enabled b and c, disabled a, and had no effect on the other transitions. In fact, executing a can never alter the enabled state of e as the places they are connected to do not intersect. [5, 8, 14] introduce a static way to recognize when a transition may be enabled. The idea is to introduce the *dependency set* of a transition. This can be computed as all transitions for which an output place of a given transition t is an input place, i.e., $DependencySet(t) = \{t' \in T \mid t\bullet \cap \bullet t' \neq \emptyset\}$. In our example, the dependency set of a comprises exactly b and c. We additionally introduce a *disable set* for each transition, consisting of all transitions that can become disabled by executing a transition, i.e., any transition sharing an input place with a transition t, i.e., $DisableSet(t) = \{t' \in T \mid \bullet t \cap \bullet t' \neq \emptyset\}$. In our example, the disable set of a only contains a itself. We can make the definitions more efficient by not counting double arcs of t in the definitions of $DependencySet$ and $DisableSet$ (as executing t never changes any place connected by a double arc). We cannot ignore double arcs of t', though, as they may still prevent enabling.

Based on dependency sets, it is possible to obtain an improved algorithm for random simulation of high-level nets. The basic idea is to partition all transitions into the transitions that are known to be disablede and those for which the enabling status is currently unknown. We note, we do not have a set for enabled transitions, as we immediately execute a transition if it is found to be enabled. When we deal with timed models, we can further split up the transitions that are not known to be disabled, as such transitions may not be enabled right now, but may become enabled at a later stage when time has increased. Thus, we partition transitions into three sets: the Disabled, the Unknown, and the MaybeReady.

An algorithm for random execution of transitions using the $DependencySet$ of transitions is shown as algorithm 1. The algorithm works in time epochs, each executing all transitions enabled at a certain time stamp. Unknown contains all transitions that are possibly enabled in the current epoch and MaybeReady transitions that may become enabled in a later epoch. We assume that Enabled returns one of three values: enabled, disabled, or maybe_ready_at(n), where the last value not only indicates that the transition is netiher enabled now nor definitely disabled, but also provides an estimate (n) of when the transition may be enabled. We initially insert all transitions into MaybeReady with weight 0 (we use weight instead of priority when referring to priority queues to not confuse this with the priority of transitions). The weight is an estimate of which time the transition is first enabled. We have an outer loop (ll. 4–19) checking if MaybeReady is empty (l. 4). Before executing the inner loop we increase the global clock to the value of the least weight of MaybeReady (when the first transition may be enabled, l. 5), and remove transitions with the least weight from MaybeReady and add them to Unknown (l. 6). The inner loop (ll. 7–19) picks a

Algorithm 1 Algorithm for random simulation of timed models.

```
 1: Unknown ← ∅
 2: Disabled ← ∅
 3: MaybeReady ← {0} × T
 4: while MaybeReady ≠ ∅ do
 5:    IncreaseTime(MaybeReady)
 6:    Unknown ← RemoveLeast(MaybeReady)
 7:    while Unknown ≠ ∅ do
 8:       Pick any t ∈ Unknown
 9:       if Enabled(t) = enabled then
10:          Execute(t)
11:          Unknown ← Unknown ∪ DependencySet(t)
12:          Disabled ← Disabled \ DependencySet(t)
13:          MaybeReady ← MaybeReady \ DependencySet(t)
14:       else if Enabled(t) = disabled then
15:          Unknown ← Unknown \ {t}
16:          Disabled ← Disabled ∪ {t}
17:       else if Enabled(t) = maybe_ready_at(n) then
18:          Unknown ← Unknown \ {t}
19:          MaybeReady ← MaybeReady ∪ {(n, t)}
```

transition at random from Unknown. If is enabled, it is executed (ll. 9–13) and the three data structures updated accordingly. If it is disabled it is moved from Unknown to Disabled and if it may become enabled in the future, it is moved to MaybeReady.

2.2 Data-structures for Random Simulation

To implement the algorithm 1, we need to implement data-structures for Unknown and MaybeReady. 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).

The operations needed for Unknown are to add all transitions, pick a random element, add a set of elements, and remove a particular element. This can be efficiently implemented by enumerating all transitions from $0, 1, \ldots, |T| - 1$, storing them in an array A of size $|T|$ 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 $|T|$. Picking a random element corresponds to drawing a random number $r \in \{0, 1, \ldots, last - 1\}$ and returning the value $A[r]$. Adding a set of contained elements consists of adding the elements to positions $last, last + 1, \ldots$ and incrementing $last$ accordingly, making sure to ignore duplicates (which can be recognized using a bit-array with $|T|$ bits). 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

Algorithm 2 Algorithm for checking enabling with caching.

```
1: proc CheckEnabling(t) is
2:   if t ∉ 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 ← Unknown \ {t}
9:       Disabled ← Disabled ∪ {t}
10:      return false
11:    else if Enabled(t) = maybe_ready_at(n) then
12:      Unknown ← Unknown \ {t}
13:      MaybeReady ← MaybeReady ∪ {(n, t)}
14:      return false
```

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 $\log |T|$ 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.

Using these data structures, we can also efficiently cache enabling computations, which is useful for graphically showing which transitions are enabled after execution of one or more steps. This is shown as algorithm 2. We could also cache the status of enabled transitions, but have chosen not to as we do not need this information for algorithm 1. The algorithm only checks for enabling at the current time and needs someone external to move elements from MaybeReady to Unknown when Unknown becomes empty. We note the bit-array indicating membership of elements in Unknown allows to retain constant time look-up in line 2.

2.3 Priority Concepts

For low-level nets, static priorities are defined in [3] and dynamic priorities in [2]. Adapting the definition of [3] to coloured Petri nets, we get:

Definition 3 ((Relational) Static Priority (Def. 3.1 in [3])). *A **static (relational) priority system** is a pair $(\mathcal{CPN}, p)$ such that $\mathcal{CPN}$ is a coloured Petri net ($\mathcal{CPN} = (P, T, D, Type, TD, Pre, Post, M_0)$) and $p \subseteq T \times T$ is a relation, called the **priority** relation.*

Intuitively, if $(t, t') \in p$ then t' has priority over t. In [3] no restrictions are put on p, but it is mentioned that making it transitive is often a good idea. We have

added relational to the names here as we later wish to introduce a slightly more restricted notion of static priorities. When dealing with priority systems, we say that if a transition is enabled according to Def. 2, it is *preenabled*. We can now define enabling for priority systems as:

Definition 4 (Enabling with priority (Def. 3.3 in [3])). *A transition $t \in T$ is **enabled** in a marking $M \in \mathbf{N}^{PLACE \times TD}$ at time $ts \in TD$ if t is preenabled and no transition t' with $(t, t') \in p$ is preenabled.*

The definition of priorities is extended in [2] to dynamic priorities, by making p a function of the current marking; adapting this to CPNs yields:

Definition 5 ((Relational) Dynamic Priority (Def. 3 in [2])). *A **dynamic (relational) priority system** is a pair $(\mathcal{CPN}, p)$ such that $\mathcal{CPN}$ is a coloured Petri net ($\mathcal{CPN} = (P, T, D, Type, TD, Pre, Post, M_0)$) and $p : \mathbf{N}^{PLACE \times TD} \rightarrow \subseteq T \times T$ assigns to each marking a priority relation.*

We can extend the notion of enabling to take the marking into account for the priority function.

3 Static Priorities

In this section we develop an algorithm for fast random execution of transitions for timed coloured Petri net models using static priorities. We also develop algorithms for operations useful for graphical tool support for simulation and modification of such models. We present experimental performance data of the algorithms on both toy examples and several real-life models [7,13,15] developed in other contexts.

When we talk about coloured Petri nets with priorities, we prefer not having to deal with relations as they are difficult to grasp and specify. Instead we assign to each transition an expression resulting in a non-negative integer, which indicates the priority of the transition:

Definition 6 (Static priority). *A **static priority system** is a pair $(\mathcal{CPN}, p)$ such that $\mathcal{CPN}$ is a coloured Petri net ($\mathcal{CPN} = (P, T, D, Type, TD, Pre, Post, M_0)$) and $p : T \rightarrow \mathbf{N}$ assigns to each transition a natural number, the priority.*

Priorities considered here are global and do not depend on the binding of the transition; we later discuss other priority concepts. We can translate such a function to a static relational priority system in the sense of Def. 3 by considering two transitions, t and t', in relation if $p(t) < p(t')$. This induces an enabling condition saying that 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 Simulation with Priorities

Our goal is to quickly execute transitions randomly, 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 highest 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 to get the *RandomSet* with highest priority.

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 $\log |p|$ for each element. We can also maintain an array mapping priorities to *RandomSet*s, bringing down insertion time for each element to constant time, as we can get the correct *RandomSet* in constant time from the priority. This, however, comes at the cost of using memory linear in the highest numeric value of a priority. Finally, we can 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 has little influence on the practical speed of the algorithm.

We call a priority queue of *RandomSet*s with an auxiliary data-structure allowing fast insertions a *PriorityRandomSet* and obtain an algorithm for random execution of transitions adhering to priorities by using algorithm 1 with a *PriorityRandomSet* implementation for Unknown.

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 the auxiliary data-structure to obtain the correct *RandomSet* for insertion. 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).

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, compute enabling for all remaining transitions with the same priority, and do not compute enabling for transitions with lower priority. Sometimes we may want to know enabling of only a subset of all

transitions. For example, if a user is only looking at part of a model, the tool may only need to compute enabling for 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 good 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. We already have an algorithm using caching this information in the case without priority, namely algorithm 2. The idea is to use that algorithm 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 transition 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 $CheckEnablingPriority$ 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

Algorithm 3 Simple algorithm for checking enabling with priority.

```
1: SortedTransitions ← PrioritySort(T)
2: proc CheckEnablingPriority(t) is
3:   for all t' ∈ SortedTransitions do
4:     if p(t') > p(t) then
5:       if CheckEnabling(t') then
6:         return false
7:     else
8:       return CheckEnabling(t)
```

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 lowest 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 can 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*.

We have implemented algorithm 1 with priority support in CPN Tools. In Fig. 2, we see two snap-shots from CPN Tools for a simulation of the model in Fig. 1. In the top screen-shot, we have executed the trace a, c, d from the initial state. The number of tokens on a place is shown in a circle and the full specification of multi-sets is shown in the adjoining rectangle. We see that even though b and c are preenabled (have enough tokens to be enabled) they are shown without a halo, indicating they are not enabled. This is because d and e are enabled and have higher priority. At the bottom of Fig. 2 we see the state after executing the trace d, e, b from the top screen-shot. Now, b and c with normal priority are enabled, but a, which has low priority, is only preenabled. In both screen-shots we see that the priorities are just defined as symbolic constants in the Declarations (middle left side) and that we use numerically low values for high priorities (an implementation detail we shall ignore for readability). A user can add any other desired priority or any closed expression (an expression not dependent on the binding of the transition, such as a number or a call to a function with arguments that are closed expressions) in the priority inscriptions.

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.

In order to be able to implement a view such as the one in CPN Tools (Fig. 2), showing enabled transitions efficiently during simulation, we can get by with few changes. 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. This can be relevant for any tool. If the number

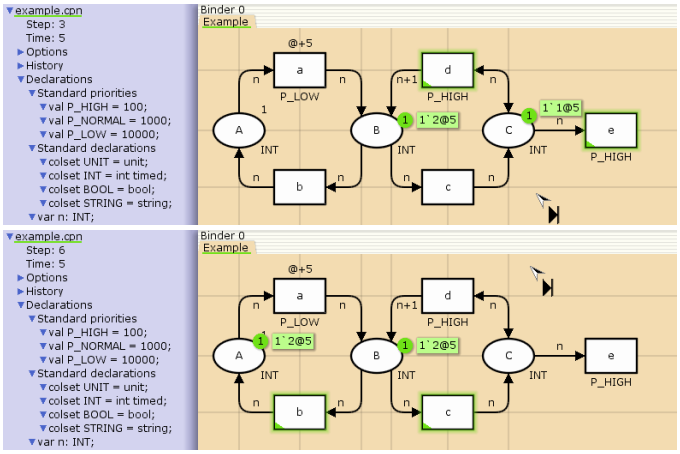

Fig. 2: Screen-shots of the example from Fig. 1 in CPN Tools. The top screen-shot shows the state after executing the trace a, c, d from the initial state, and the bottom after executing d, e, b from the top state.

of transitions is high, it is a good idea to make the number of enabling checks not directly dependent on the total number of transitions in the model. In CPN Tools this is particularly important because such calls incur a communication overhead, as the GUI and the simulator are separate processes.

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. The reason is that the enabling of a transition with higher priority than all currently enabled transitions will disable them. We note that we can just call *Enabled* for all transitions (we are interested in) with the same priority as t between lines 9 and 10, but we prefer an algorithm that does not require global knowledge, that *CheckEnablingPriority* can be called independently of algorithm 1, and that Unknown, Disabled, and MaybeReady are updated with information about the enabling status.

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). This algorithm can be made interactive by asking the user for input in line 12.

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

```
 1: SortedTransitions ← PrioritySort(T)
 2: Unknown ← ∅
 3: Disabled ← ∅
 4: MaybeReady ← {0} × T
 5: Enabled ← ∅
 6: while MaybeReady ≠ ∅ do
 7:    IncreaseTime(MaybeReady)
 8:    Unknown ← RemoveLeast(MaybeReady)
 9:    while Unknown ≠ ∅ do
10:       Enabled ← {t' ∈ Unknown | CheckEnablingPriority(t')}
11:       while Enabled ≠ ∅ do
12:          Pick any t ∈ Enabled
13:          Execute(t)
14:          Unknown ← Unknown ∪ DependencySet(t)
15:          Disabled ← Disabled \ DependencySet(t)
16:          MaybeReady ← MaybeReady \ DependencySet(t)
17:          Enabled ← Enabled \ DisableSet(t)
18:          New ←
                 {t' ∈ DependencySet(t) ∪ DisableSet(t) | CheckEnablingPriority(t')}
19:          if New ≠ ∅ then
20:             if ∃t1 ∈ New, t2 ∈ Enabled.p(t1) > p(t2) then
21:                Enabled ← New
22:             else
23:                Enabled ← Enabled ∪ New
```

3.4 Application to Incremental Syntax Check

CPN Tools supports incremental syntax check, i.e., incrementally building and simulating a model. We would like to extend this incremental syntax check to also support transitions with priority. If a tool is supposed to be interactive, interactive operations cannot be directly dependent on the size of the full model; the algorithms explained here only depend on the dependency and disable sets and hence on the local surroundings of transitions. We mostly need to support enabling set maintenance, as modification during automatic non-interactive simulation rarely makes sense. The method we use to support enabling set maintenance during model modification can easily be extended to also work with random simulation (assuming proper locking of data-structures). We assume that any modification of the model takes place between line 11 and 12 in the algorithm, i.e., while the user has the ability to manually pick a transition. This can be obtained if the tool waits for the user to manually pick a transition to execute or by introducing a mutex that is released after line 11, reclaimed before line 12, and necessary to make any changes to the model. Adding or removing unconnected places never changes the enabling of transitions, and can be ignored in relation to enabling computation.

For transitions, the trick is to observe that adding or removing a transition corresponds to executing a particular hypothetical transition. For example, adding a transition corresponds to executing a τ transition (which does not change the marking of any place) with the new transition in its dependency set. The user interface is updated according to the changes of Enabled. Modification of a transition can be thought of as first removing the old transition and then adding a new one, possibly with another priority. In CPN Tools, this is exactly how it is implemented, so we do not have to handle this case explicitly, but it is possible to handle it more efficiently by combining the two operations into one, having the old transition in the disable set and the new one in the dependency set. This use of the enabling set maintenance algorithm imposes extra requirements on the efficiency of that algorithm.

3.5 Experimental Validation

We have compared the algorithm for random non-interactive simulation (Algorithm 1 with priority support) with a naive algorithm just evaluating enabling in a highest-priority-first order (Priority), an algorithm computing all enabled bindings for all transitions before selecting a transition to execute (All), and a new fair simulation mode in CPN Tools 3.2 using algorithm 1 but computing all enabled bindings for each transition instead of just checking enabling (Fair). Fair thus sits between All and Algorithm 1 with similarities to Priority as it evaluates transitions highest-priority-first but uses the Unknown and MaybeReady structures. As Fair and Algorithm 1 are not prototype implementations like All and Priority, they are expected to perform slightly worse, as they perform extra tasks needed in CPN Tools, such as evaluating break-points, checking if logging should be performed, evaluating whether monitors should be updated, etc. 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) [13], the DYMO protocol for route discovery in mobile ad-hoc networks [7], and a protocol for operational support for workflow execution (OS) [15]. 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 have four versions of the OS model: a base model (OS 1) similar to the one presented in [15], an extended version (OS 2) allowing clients to share sessions, an extended version (OS 3) also modeling a compatibility layer, and a version of OS 3 (OS 2/3), where the compatibility layer has been disabled so the behavior is the same as for OS 2 but the model has more transitions to consider. The four versions of OS use priorities while the other models do not (they were developed before CPN Tools supported priorities).

For each model, we show the complexity in terms of the number of modules, the number of transitions, and the number of places. We also show the number of place instances after merging all places in port/socket assignment relationships. We show the number of transitions we can execute each minute for each model

Table 1: Experimental results.

Model	Instances				10^6 Transitions/minute			
	Pages	Transitions	Places		All	Priority	Fair	Algorithm 1
Philosophers	1	3	3	(3)	0.34	11.03	0.41	24.78
Database	1	5	9	(9)	4.92	15.01	2.62	22.38
Protocol	1	5	10	(10)	2.73	8.10	1.99	23.40
ERDP	14	16	65	(15)	0.51	1.31	0.26	4.23
DYMO	15	25	55	(18)	0.77	2.57	1.66	4.59
OS 1	27	39	146	(32)	0.67	1.93	0.61	5.66
OS 2	29	50	147	(34)	0.54	1.53	0.47	4.92
OS 3	35	59	178	(44)	0.29	0.80	0.23	3.49
OS 2/3	35	59	178	(44)	0.46	1.28	0.41	4.39

and algorithm. The tests are performed by running CPN Tools 3.2.1 on a computer with a 2.7 GHz Core i7 Sandy Bridge dual core CPU (using one core). All tests were run for 5 minutes and the average is 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 algorithm 4 as it incurs a communication overhead due to the architecture of CPN Tools. We have not compared with a baseline simulator without priority for two reasons: Firstly, while the performance of simulation of a model without priorities is the same for our present implementation and an implementation not supporting priorities, the performance of a model using priorities is incomparable, as lack of support for priorities may cause the model to be able to reach states not reachable when using priorities, hence comparing models with different behavior. Secondly, we only have an implementation with an old version of the simulator, which for independent reasons is much slower.

We see that using our improved algorithm, the toy examples can execute more than 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. The penalty of the improved algorithm for real-life models compared to toy examples stems from the fact that each transition is much more complex (they call functions and do more advanced matching on the input tokens consumed) and therefore take longer to execute. We note that the algorithm Fair performs worse then All in some cases, even though we would expect them to perform equally well for models without priority, and Fair to perform better for models with. This is due to the extra work performed to allow logging, breakpoints, etc., not performed by the prototype implementations. This overhead is also performed by Algorithm 1, so the actual advantage of this algorithm is even larger than the numbers suggest.

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 more dynamic priority concept is defined (as shown in Def. 5). 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 allowing general expressions for inscriptions, like guards or arc expressions. Formally, this is defined as:

Definition 7 (Coloured priority). *A **coloured priority system** is a pair* $(\mathcal{CPN}, p)$ *such that* $\mathcal{CPN}$ *is a coloured Petri net (*$\mathcal{CPN} = (P, T, D, Type, TD, Pre, Post, M_0)$*) and* $p : TRANS \rightarrow \mathbf{N}$ *assigns to each binding element a natural number, the priority.*

Although this concept is natural, we have chosen not to implement 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 All algorithm seen in Table 1 is the most efficient implementation of coloured priorities unless we make further assumptions; we see that this algorithm is outperformed by a factor of around 10 in all cases. In this section we discuss alternatives to the static, global notion of priority presented hitherto without compromising performance too much.

4.1 Using a Subset of Variables in Priorities

CPN Tools, in addition to restricting the number of times enabling of a transition is computed, also tries to make each computation as efficient as possible. One of the tricks it uses is to partition all variables surrounding a transition into *binding groups* [4,8,14]. 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).

In order to compute all possible priority values for a transition, we can just compute bindings for all binding groups having variables occurring in the priority expression. If the priority expression only has few variables but the transition many, this may yield a reduction as we no longer have to compute a full binding to know enabled priorities of the transition. 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.

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`.

4.2 Scoped Priorities

Coloured Petri nets have a module concept. Basically, it is possible to use a model as a module in another model, refining the behavior of a special kind of transition, called a substitution transition. Such modules are called pages. It is often useful to be able to use scoped priorities. For coloured Petri nets, 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. Scoped priorities can be defined using any priority concept as:

Definition 8 (Scoped priority). *A* scoped priority system *is a pair* $(\mathcal{PS}, S)$ *such that* $\mathcal{PS} = (\mathcal{CPN}, p)$ *is a priority system with* $\mathcal{CPN} = (P, T, D, Type, TD, Pre, Post, M_0)$ *and* $S : T \to \mathbf{N}$ *is a partitioning of the transitions.*

Enabling is defined relative to the partitioning so we only consider transitions with higher priority in the same scope. For examnple, if the underlying priority system is assumed to be a static priority system, enabling would be defined as

Definition 9. *In a scoped static priority system* $((\mathcal{CPN}, p), S)$ *a transition* t *is enabled if it is preenabled and no* t' *with* $S(t) = S(t')$ *and* $p(t) < t(t')$ *is preenabled.*

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 as executing an action in one module no longer is able to disable actions in others unless they are connected directly.

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 of not allowing scoped priorities is that flattening of a hierarchical CPN model remains a purely syntactical operation, where we would otherwise have to consider interplay of local priorities.

5 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. We have also given an algorithm 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 [6], 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 previously possible $1 - 5$ million transitions a minute regardless of the complexity.

We have considered algorithms for handling coloured priorities and for scoped priorities. We have chosen not to implement these, because coloured priorities are prone to introducing performance bottlenecks, and 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 interesting 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 expensive than for low-level net classes. 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 [11] or for doing distributed simulation with priorities present [12]. 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, though they are not tailored for this. In fact, many of the challenges encountered for efficiently simulating CPNs disappear if we turn to state-space exploration or model-checking in general. This is because in order to do this kind of analysis, we need to compute all enabled bindings of all transitions anyway, so doing this to evaluate, e.g., coloured priorities is no an overhead but necessary in any case. As for simulation, analysis can be done by translating to equivalent models without priority [3, 11] and for low-level nets additionally by means of static analysis or restriction [1, 2]. For high-level nets, analysis is usually only possible by means of state-space exploration or simulation, making fast simulation algorithms more important.

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. G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. On Well-Formed Coloured Nets and their Symbolic Reachability Graph. In *High-Level Petri Nets. Theory and Application*, pages 373–396. Springer, 1991.
5. J.M. Colom, M. Silva, and J.L. Villarroel. On software implementation of Petri Nets and colored Petri Nets using high-level concurrent languages. In *Proc. of ATPN'86*, 1986.
6. CPN Tools webpage. Online: `cpntools.org`.
7. 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.
8. T.B. Haagh and T.R. Hansen. Optimising a Coloured Petri Net Simulator. Master's thesis, Dept. of Computer Science, Aarhus University, 1994.
9. Software and system engineering – High-level Petri nets – Part 1: Concepts, definitions and graphical notation. ISO/IEC 15909-1:2004.
10. K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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, pages 169–188. Springer, 2011.
16. M. Westergaard and H.M.W. Verbeek. Efficient Implementation of Prioritized Transitions for High-level Petri Nets. In *Proc. of PNSE'11*, volume 723 of *CEUR Workshop Proceedings*, pages 27–41. CEUR-WS.org, 2011.