Universiteit Leiden
Opleiding Informatica

A Formal Representation and Calculation Method

for Generic Petri nets

Name: J. Viëtor

Date: 31/05/2016

1st supervisor: Jetty Kleijn
2nd supervisor: Fons Verbeek

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

A Formal Representation and Calculation Method for Generic Petri nets

J. Viëtor

May 31, 2016

Abstract

Petri nets can be used for a wide range of applications. However due to the increasing amount of extensions made to "classic" Petri nets there is no single unifying method to work with all of them. This paper introduces a generic notation model for Petri net classes, encompassing the most popular types. This model is then extended with algorithms that can be used for all classes that fit the model. To complete the solution, an open-source proof of concept Petri net simulator/animator (PetriCalc) is provided which works according to this theory.

Contents

1 Introduction

The usefulness of Petri nets is undisputed, and there is a myriad of tools available to work with them - anything from interfaces for building Petri nets to analysing their structure or running simulations using them.

Unfortunately, this increasingly widespread use and interest has led to a fragmentation of classes of Petri nets as well as confusion around the vocabulary and semantic meaning used for them. There are various efforts (such as [HMR05, BM01]) to classify and analyse transition systems, of which Petri nets are just one example.

While there are tools that allow running simulations on Petri nets, these are mostly geared towards animation of small nets, producing graphics. Tools for doing large-scale simulations are much harder to find, and often crash-prone or too slow to be of practical use. This problem became especially apparent while Laura Bertens was working on [BKH+15], where she needed to simulate a Petri net that contained millions of tokens and hundreds of places and transitions (see Figure 1). This seemingly simple task proved to be too difficult for Snoopy (the tool the net was modelled in) or any other existing simulation method and/or tool. Additionally, only a handful of places needed to be monitored for token counts and these counts were only needed exactly every 5 simulation steps.

Inspired by the inability to simulate particularly large and/or complex Petri nets, a calculation tool (`PetriCalc`) was developed especially designed to be able to handle this. In order to do such calculations, it was important to have an efficient internal data model for the net as well as a straightforward and correct way to convert existing nets to this internal data model.

After completion of `PetriCalc` around the end of 2011, Jetty Kleijn and the author of this paper agreed that the methods and data model as used in `PetriCalc` could be very interesting if properly formalized. This paper is the result of those efforts. The formal method was then turned into changes to `PetriCalc` itself so that it accurately follows this formal notation, of which the first version of `PetriCalc` could be considered a rough draft. Since `PetriCalc` and the formal methods described in this paper are so closely tied together, the constant goal to keep simulation possible for large and/or complex Petri nets will be visible in nearly all decisions made.

Because of the new notation used to work with Petri nets, this paper takes a different approach to Petri nets compared to most other literature. Usually Petri nets are introduced as simple place-transition systems, then weights and other extensions such as colours, extra arc types, time and/or stochasticity are added. In this paper however, we focus more on the notation, starting with an already extended Petri net model based loosely on and conforming to old and well-accepted definitions of extended Petri nets such as [Pet77] and [Mur89].

Specifically, the notation is shifted from a place/transition-centric model to an arc-centric model. The reason for this shift is three-fold:

First, we would like to provide a more direct translation from theory to computer programming code for running simulations. Using only a single concept

(arcs) as opposed to two (places and transitions) makes this translation both easier to follow and less complex.

Secondly, we would like a fast and efficient implementation of this code. Having only a single concept to keep track of makes optimizations easier and lowers memory usage because there are less objects which matter to the calculations to keep track of.

Finally, the focus on arcs allows us to come up with rules for combining arcs together, which not only benefits optimization (by further reducing the objects that we need to keep track of) but also assists net analysis efforts by providing means of classifying groups of arcs together as opposed to only singular arcs.

The focus on speed and translation to computer programming code will be visible as a recurring theme throughout this paper as the main motivation behind most decisions that were made.

While the focus may be shifted, the model is still fully compatible with and intuitively similar to traditional models, and straightforward translations both to and from those models to this model are possible. This enables the use of all techniques and tools as discussed in this paper on many existing Petri net classes and thus many existing Petri nets.

Throughout this paper both Demarcations and Definitions will be used. Demarcations give information on principles, ways of thinking; while definitions give the harder, mathematically sound boundaries.

Demarcation 1 *A* Demarcation *explains the reasoning or principles (soft boundaries) of a subject, while a* Definition *gives mathematically sound (hard) boundaries.*

Demarcations usually precede definitions, allowing terminology to be used without fully defining it first. Additionally, they allow for easily looking up the basic ideas behind concepts without needing to read through the full definition.

This paper will begin with some preliminaries that establish basic principles. It continues with a brief overview of Petri nets in the new notation, followed by an explanation of the theoretical focus of this paper. Then the concepts of Petri net structure and Petri net classes are introduced, coming together with a marking to form marked Petri nets. With all the basics then established, it continues with a list of various properties that hold for all nets that fit in the new notation as well as algorithms for stepping through simulations with various step types. At that point, the Petri net class of Natural Petri nets is introduced and some properties of this class are explored. Finally, the translation of all this theory to code is given and it ends with a short discussion and conclusion section.

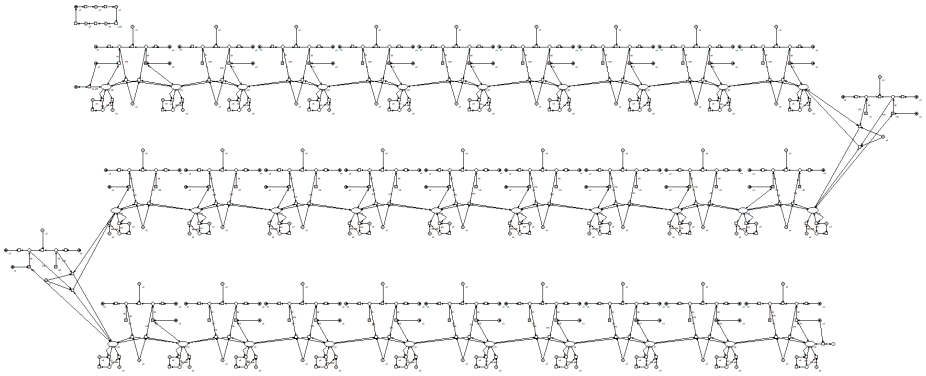

Figure 1: The Petri net from [BKH+15] modelling gradient formation. It has 125 places and 272 transitions.

2 Preliminaries

We use $\mathbb{Z}$ to denote the set of integers, meaning both positive and negative whole numbers as well as zero; $\mathbb{N}$ to denote the set of natural numbers, meaning positive integers including zero; and $\mathbb{N}_\infty$ to denote $\mathbb{N}$ with the addition of the special symbol *infinity* ($\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$). In this paper, the only property of ∞ that is used is that $\infty > i$ where $i \in \mathbb{N}$.

A multiset $\mu : A \to \mathbb{N}$ is a mapping from the set A to $\mathbb{N}$, with $\mu(a)$ indicating how many times a occurs in the multiset. Sets and subsets can be seen as multisets through their characteristic function:

$\mu_X(a) = \begin{Bmatrix} 0 & \texttt{if} & a \notin X \\ 1 & \texttt{if} & a \in X \end{Bmatrix}$ where $X \subseteq A$ and $\mu_X : A \to \{0, 1\}$.

`min` and `max` denote the smallest, respectively largest element of a finite totally ordered set.

A commutative monoid is a 3-tuple $(L, \otimes, \mathbb{I} \in L)$ where L is a set, $\otimes$ is a commutative and associative binary operator, and $\mathbb{I}$ is the identity element for the $\otimes$ operation on set L.

The concept of "connected" elements will be used quite often in this paper. In order to save space and make things easier to read, the symbol ‡ represents the phrase "connected to" and the symbol † represents the phrase "not connected to", where the nature of the connection is dependent on the context.

3 What is a (Petri) net?

This section introduces "Petri nets" as we will consider them in this paper. Since the world of Petri nets is filled with a wide range of specialized net classes, it is important to clarify exactly which classes we are dealing with.

Petri nets are usually introduced as (annotated) directed bipartite graphs, where the set of nodes is partitioned in two disjoint subsets. Places are in one subset and transitions in the other. Places are never directly connected to places, and transitions are never directly connected to transitions. Transitions represent *actions* while places represent *state variables*. The state of these variables is represented by "marking" the places. Depending on the exact class of net, the marking of a place may be binary, a number, a set or even a complex expression. The arcs between a particular transition and places define its sphere of influence. The neighbourhood of an element consists of the element itself, the elements it is connected to, and the arcs involved. A transition may not affect places outside of its neighbourhood.

For the sake of unambiguity, these are the elements of a Petri net:

Demarcation 2 *A* transition *is a representation of an action. The occurring of an action is called* firing.

If a transition is allowed to fire, this transition is said to be enabled. *Whether a transition is enabled and the effect of its firing are fully defined by the arcs in its environment (see Definition 6).*

Transitions will be visually displayed as rectangles:

Figure 2: A transition

Demarcation 3 *A* place *is a representation of a state variable.*

Places will be visually displayed as circles:

Figure 3: A place

Demarcation 4 *The* marking *of a net defines a value $d \in D$ (where D is a given domain) for each place in the net. A marking describes a state of a net, but is not part of the net itself.*

Markings will be visually displayed by writing the corresponding value inside the places:

Figure 4: A place with the value of the current marking

Demarcation 5 *An* arc *connects a single place to a single transition. It represents both under what conditions on its connected place its connected transition may fire (the arcs* range*) and how the value of its connected place is altered when its connected transition fires (the arcs* effect*).*

Arcs will be visually represented by a labelled line between a place and a transition, with the range and effect of the arc together forming the label:

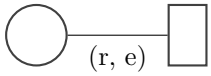

Figure 5: A place and a transition, connected by a labelled arc

Range and effect are not usually represented in this way. We have chosen arc-labelled undirected bipartite graphs, as opposed to non-labelled directed bipartite graphs. More on this decision is explained in the following subsection.

3.1 Theoretical focus

We have introduced arcs as specifying an undirected connection between place and transition, together with an arc label consisting of the arc range and arc effect. This is not the usual representation of arcs, however.

In the usual representation, Petri nets are seen with a focus on either the places or the transitions, with a flow relation between them. The things that give Petri nets their behavioural properties are hence properties of the places and transitions. The flow relation merely points out which places and transitions affect each other.

This paper uses a quite different way of looking at things. Instead of focusing on places and/or transitions, it focuses on the flow relation (the arcs) between them. We assign properties to the arcs (specifically, the arc range and arc effect). This leaves the places as nothing more than identifiers for specific parts of a marking, and transitions as nothing more than identifiers for a set of arcs. Since they are identifiers for parts of a net, they are still quite useful to humans but have lost their computational significance.

Because places and transitions have lost their computational significance, only the arcs are left as important for computations. Instead of needing to keep track of two different types of objects and their relation to each other, we only have to keep track of one type of object: the relations themselves. This also has the added benefit of making transformations on the net easier, such as combining transitions.

Since the properties that are computationally significant are all defined through the arcs, it becomes possible to write down a single set of arc-based rules that governs all behaviour. We shall refer to such rule sets as the class of a Petri net, while the places, transitions and arcs themselves shall be referred to as the structure of a Petri net.

Both the structure and class of a net are static for the lifetime of a net. But there is also a third, dynamic part: the marking. These three parts together fully describe a Petri net and its current state.

The structure defines what transitions, places and arcs (including their labels) exist in specific net. Changing the structure is thus effectively creating a new, different, net.

The class defines how and under what conditions arcs interact with the marking of a net. The class can be seen as a set of 'rules', which the net must follow. Changing the class doesn't just create a new net: it creates an entirely new type of net; hence the naming "class" for this part.

Finally, the marking describes the current state of a net. It is the dynamic part, changing when a net evolves.

3.2 Structure

The structure of a Petri net is determined by places, transitions and the arcs between them.

Definition 1 *$N = (P, T, A)$ is a `PetriStructure`, where:*

- *P is a set of* places.
- *T is a set of* transitions*, such that $P \cap T = \emptyset$*
- *A is a set of* arcs*, each connecting a place and a transition and labelled with a tuple (r, e) where r is the range of the arc, and e is the effect of the arc. If an arc a with label (r, e) thus connects (‡) place $p \in P$ and transition $t \in T$, then we write:*
  - $a_P = p$
  - $a_T = t$
  - $a_R = r$
  - $a_E = e$

Some remarks about the above definition:

- We require that $P \cap T = \emptyset$, because P and T represent different entities.
- Each of P, T, and A is allowed to be infinite or empty.
- The graph represented by this definition is *not* directed, so unlike the traditional definition of Petri nets, there is no distinction between input arcs / places / transitions and output arcs / places / transitions.

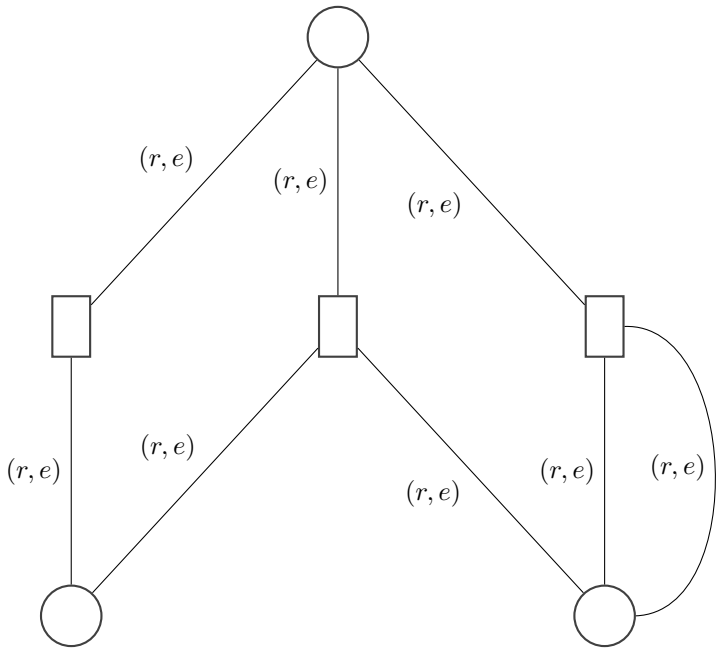

Figure 6: An example of a Petri net structure. The (r, e) arcs are placeholders for more concrete labels.

- There may be several arcs $a \neq b$ such that $a_P = b_P, a_T = b_T, a_R, = b_R$ and/or $a_E = b_E$. In other words, the identity of an arc is not fully described by its parameters (p, t, r, e).

Arc range and effect were partially chosen for practical reasons (they provide a sensible base for calculations), but also partially inspired by [KK10], where the ideas of range arcs and testing are heavily used. The ideas about localities as they are used in [KK10] also mesh nicely with the theoretical focus of this paper on arcs as opposed to places or transitions.

Taking into account the representation of places, transitions and arcs as introduced at the beginning of this section, an example net following this definition of structure could look as depicted in Figure 6.

3.3 Class

The class of a Petri net describes the behavioural rules defining its dynamics.

First, let us introduce some new concepts: domain, the set of arc labels, the range function, the effect function, the combination operator and the identity arc label. Together these are sufficient to define behaviour.

The domain together with the set of arc labels puts restrictions on the markings and arc labels of net.

Demarcation 6 *The* domain *of a class defines the possible values that a marking may assign to places, as well as the possible values for arc range and arc effect.*

The set of arc labels of a class specifies which combinations of arc range and arc effect may occur on arc labels.

The range and effect functions together describe the equivalent of what is usually referred to as the firing rule.

The range function is used to define under what conditions an arc enables its connected transition.

Demarcation 7 *The* range function *of a class is used to determine for any given arc and marking whether a transition is enabled with regard to the place the marking corresponds to. A transition is enabled if and only if all the arcs connected to it enable it. In other words, a transition is enabled when the range function of all its connected arcs evaluate to true for the marking of their connected place.*

The effect function defines what happens to the place of an arc, when its transition fires.

Demarcation 8 *The* effect function *of a class determines for any given arc and the current marking of its place what the resulting marking of its place is after firing its transition.*

In addition there is a combination operator that defines concurrent behaviour: how two arcs connected to the same place can be combined to create a single new arc that behaves as a simultaneous occurrence of these arcs, from the perspective of the range function and effect function.

Since simultaneous behaviour is inherently unordered, this operator is commutative and associative. Thus it may be used to combine any number of arcs connected to the same place into one. See Figure 7 for an example.

Demarcation 9 *The* combination operator *($\otimes$) of a class defines how any two arcs connected to a shared place may be combined into a single arc with the same effect as if the two arcs would work simultaneously. It is commutative and associative.*

All of the arc labels involved in the use of this operator must be in the set of arc labels of the class.

Finally, there is an identity arc label. This is effectively an "unconditionally, do nothing" arc label. This sounds like a very useless arc label to have at first, but it serves two key purposes.

Firstly, it makes it possible to make a fully connected net, where every transition is connected to every place. Unconnected places and transitions can be connected using an arc labelled with the identity arc label, without altering the semantics of the net itself.

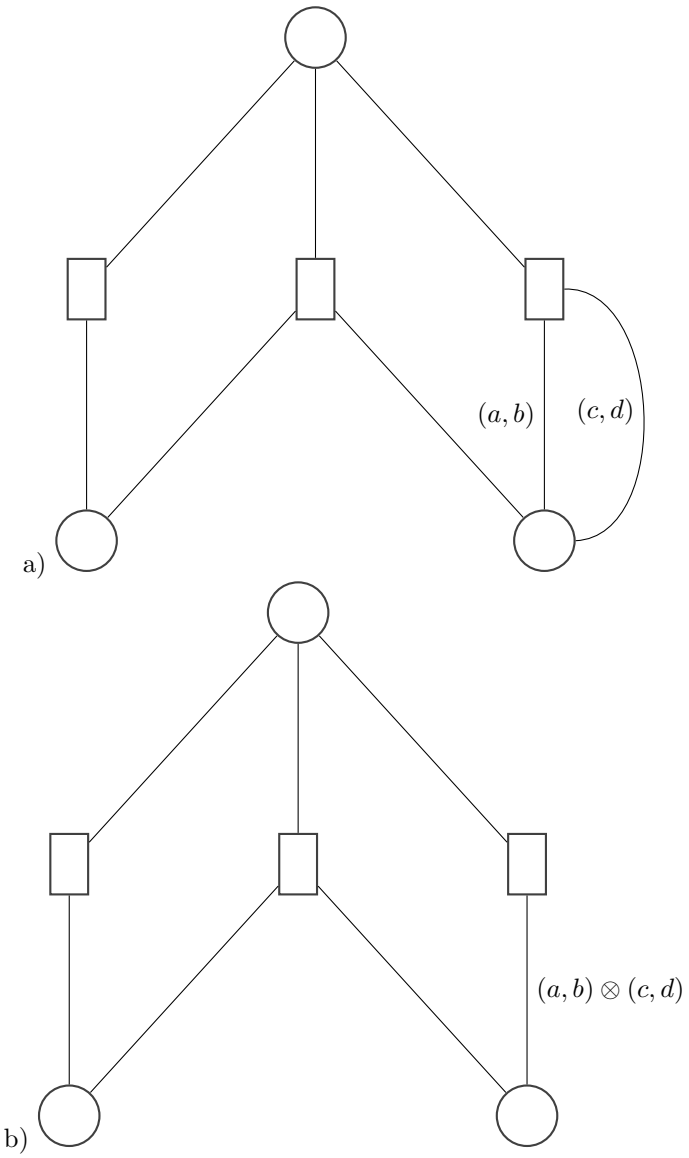

Figure 7: In b, two arcs from a combined into one with a new label obtained from the original arcs using the combination operator $\otimes$.

Secondly, the existence of this arc label is the final piece of the puzzle to be able to claim that when put together with the set of arc labels and the combination operator, a commutative monoid is created. This is desirable because commutative monoids have very convenient properties (specifically, being associative and commutative).

Demarcation 10 *The* "identity arc label" *($\mathbb{I} = (\mathbb{I}_r, \mathbb{I}_e)$), is an element of the set of arc labels such that $a \otimes \mathbb{I} = a$ for any given arc a. It must have an unconditional range, and may not alter any marking through its effect.*

Putting all of the above into more formal language results in the following definition of Petri net class:

Definition 2 *A* `PetriClass` *is a tuple $(D, f_r, f_e, L, \otimes, \mathbb{I})$ where:*

- *$D = (D_r, D_e, D_m)$, where:*
  - *D_r is the domain for arc range.*
  - *D_e is the domain for arc effect.*
  - *D_m is the domain for markings.*
- *$f_r : D_r \times D_m \to \{true, false\}$ is the range function.*
- *$f_e : D_e \times D_m \to D_m$ is the effect function.*
- *L is a subset of $(D_r \times D_e)$, the set of arc labels of the* `PetriClass`*.*
- *$\otimes : L \times L \to L$ as the combination operator.*
- *$\mathbb{I} = (\mathbb{I}_r, \mathbb{I}_e) \in L$, as the identity label, such that:*
  - *$f_r(\mathbb{I}_r, i) = true$ for all $i \in D_m$*
  - *$f_e(\mathbb{I}_e, i) = i$ for all $i \in D_m$*
- *$(L, \otimes, \mathbb{I})$ is a commutative monoid.*

Note that because $(L, \otimes, \mathbb{I})$ is a commutative monoid, there is only one $\mathbb{I}$, but there may be more than one valid $\mathbb{I}_r$ or $\mathbb{I}_e$.

The property that $(L, \otimes, \mathbb{I})$ is a commutative monoid is a particularly interesting one. Petri nets have commutative monoid tendencies when modelled and/or viewed in various ways. For example, in [MM90] it is already shown that Petri nets can be seen as monoids of places with a composition operator on transitions (of which a `PetriClass` can be seen as a more generalized form that only operates on arcs instead of places and transitions). In [DMM96] a similar modelling method is used to show that both sequential and parallel step semantics (see subsection 4.3 for more on steps) can be modelled as monoids as well. More recently, [KKPKR12] has taken the approach of modelling the connections between places and transitions as monoids, again from the vantage

point of step semantics. The model described by `PetriClass`es takes inspiration from all of these and ends up with something complementary to the connection monoids in [KKPKR12] (keeping the focus on arcs as opposed to steps), while retaining the expressive ability that is seen in [MM90] and [DMM96]. This will become more clear in section 6, where we introduce `NatNet`s.

One might wonder why multiple arcs between one place and one transition are allowed, especially since the combination operator allows combining them together, preserving semantic meaning. The reason for this is to make it easier to convert nets from other models to this model. And indeed, in a later section we will use the combination operator to reduce the amount of arcs between combinations of places and transitions to a maximum of one each. This greatly simplifies the complexity of both the model in general as well as computations done on it, by keeping the amount of arcs to a minimum.

3.4 Marked Petri nets

Combining the above definitions of `PetriStructure` and `PetriClass` together, a marked Petri net is defined as a combination of structure (`PetriStructure`), class (`PetriClass`) and marking (M). The `PetriStructure` defines the structure of the net itself, the `PetriClass` defines the class of the net and the marking defines the current state of the net. Only the last of those properties (the marking) can change within the operational semantics of a net, because changing either of the other two would change the underlying net itself.

The combination of a `PetriStructure`, `PetriClass` and marking is called *permissible* when they "fit" together:

Demarcation 11 *A combination of `PetriStructure`, `PetriClass` and marking is* permissible *when the marking assigns to the places of the `PetriStructure` elements from the domain for markings of the `PetriClass` and all arc labels are allowed by the `PetriClass`.*

This gives us the following formal definition of a marked Petri net:

Definition 3 *A* marked Petri net N *is a tuple* $N = (S, C, M)$ *where:*

- $S = (P, T, A)$ *is a `PetriStructure`, as in Definition 1.*
- $C = (D, f_r, f_e, L, \otimes, \mathbb{I})$ *is a `PetriClass`, as in Definition 2.*
- S, C, M *is permissible by* L*, meaning that* $M : P \to D_m$ *is the marking and for all* $a \in A$ *we have* $(a_R, a_E) \in L$

4 Common properties of marked Petri nets

We are now able to consider a marked Petri net (as defined in Definition 3) as comprising of a `PetriStructure`, `PetriClass` and marking, but have not actually provided any of the three yet in a tangible form. Even without doing so, it is already possible to reason with marked Petri nets and further define general properties that will hold true for all possible values S, C and M.

4.1 Enabledness and firing

The concepts of enabledness and firing were already briefly mentioned in Demarcations 2 and 7.

Before we can give the full formal definitions of enabledness and firing, we first need to define the "place-transition-combined arc label". This is a method relying on the combination operator to reduce the number of arcs between any given place and transition pair to a single arc.

Definition 4 *Let (P, T, A) be a* `PetriStructure`*, $p \in P$ and $t \in T$. Let $a_1, \ldots, a_n$ be all distinct arcs connecting p and t. Let l_i be the label of a_i for $1 \leq i \leq n$. The* place-transition-combined arc label *of p and t is the label resulting from $\mathbb{I} \otimes l_1 \otimes \ldots \otimes l_n$.*

The ordering of the arc labels used above does not matter, since $\otimes$ is both commutative and associative. Also note that by this definition the place-transition-combined arc label of unconnected places and transitions is the identity arc label, as there are no labels l_i. This is desired behaviour because Defintion 2 requires $\mathbb{I}$ to unconditionally do nothing, which is semantically equivalent to a place and transition being unconnected.

An example of reducing number of arcs to one by using the place-transition-combined arc label is given in Figure 8.

With place-transition-combined arc labels, we can now relatively easily give the formal definitions of enabledness and firing. After all, now that we can combine all arcs between any given place and any given transition into a single arc, we only have to consider a single arc for each part of place and transition.

Since the identity arc label is required to always make both the range function evaluate to true and the effect function to not change the marking, this arc label may be used when no arcs connect a place and transition. This has exactly the same effect as simply leaving the markings for those places unchanged.

A transition is enabled when the range function is `true` for the place-transition-combined arc label of each connected place:

Definition 5 *In a marked Petri net $N = ((P, T, A), (D, f_r, f_e, L, \otimes, \mathbb{I}), M)$ a transition $t \in T$ is* enabled *when for all $p \in P$, $f_r(l_r, M(p)) =$ `true`, where $l = (l_r, l_e)$ is the place-transition-combined arc label of p and t.*

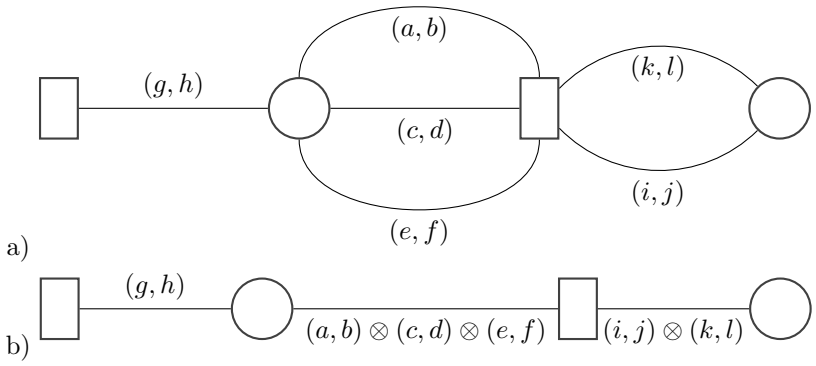

Figure 8: A simple net with several combinable arcs, showing both before (a) and after (b) using the place-transition-combined arc label to transform the net for all arcs where this is possible.

The firing of a transition defines a new marking with the result of the effect function for the place-transition-combined arc label of each connected place, leaving unconnected places unaffected.

Definition 6 *In a marked Petri net* $N = ((P, T, A), (D, f_r, f_e, L, \otimes, \mathbb{I}), M)$ *the* firing *of an enabled transition* $t \in T$ *changes* M *into* M' *so that* $\forall p \in P$, $M'(p) = f_e(l_e, M(p))$, *where* $l = (l_r, l_e)$ *is the place-transition-combined arc label of* p *and* t.

Note that since only the marking of a marked Petri net is affected by firing a transition and the marking is transformed using the effect function (f_e), the new marking is guaranteed to yield a permissible marked Petri net.

4.2 Arc types

In the notation used for Petri nets in this paper, places can be considered variables, merely holding values and playing no other role. Transitions can be seen as identifiers for a collection of arcs. As a result of this, it is the arcs that do all the work and are the most interesting to analyse. In this subsection, we will explore the theoretically possible types of arcs in marked Petri nets. We will do this through analysis of the possible values the arc labels can take.

Let us see if we can classify some major arc label types. Each arc label consists of two parts: the arc range and arc effect. Since we know there is an identity arc label ($\mathbb{I} = (\mathbb{I}_r, \mathbb{I}_e)$), both of these can either hold a value that does something or doesn't do something. For the arc label range "doing nothing" means being unconditional ($f_r(l_r, i) = \texttt{true}$ for all $i \in D_m$, where l_r is the arc label range). For the arc label effect it means not changing the marking ($f_e(l_e, i) = i$ for all $i \in D_m$, where l_e is the arc label effect). Since all ranges that do nothing are indistinguishable from $\mathbb{I}_r$ by the range function, we shall

use $\mathbb{I}_r$ to refer to all of them. Similarly, we shall use $\mathbb{I}_e$ to refer to all effects that do nothing. This leads to four major arc label types (or arc classes):

- No-operation arc labels $(\mathbb{I}_r, \mathbb{I}_e)$

  These arc labels have a range that is unconditional and an effect that does not alter the marking. In other words: an arc label that is always enabled and does nothing.

  There is at least one arc label part of this classification: the identity label, $\mathbb{I}$.

- Check arc labels $((r, \mathbb{I}_e)$, where $r \neq \mathbb{I}_r)$

  These arc labels have a range with conditions and an effect that does nothing.

  This class of arc labels does not affect the marking of the place it is connected to, but only checks the current marking.

- Radical arc labels $(((\mathbb{I}_r, e)$, where $e \neq \mathbb{I}_e)$

  These arc labels have an unconditional range and are able to change the marking through their effect.

  This class of arc labels can affect the marking of the place it is connected to in some way, without checking the current marking.

- Conditional arc labels $((r, e)$, where $r \neq \mathbb{I}_r$ and $e \neq \mathbb{I}_e)$

  These arcs both check the current marking of its connected place and may affects that marking.

  All arcs that have any range other than $\mathbb{I}_r$ and have any effect other than $\mathbb{I}_e$ are conditional arcs, making this the largest (and most interesting) class by far.

Until we define D_r, D_e, L, this is the only classification we can make on arcs, as all we know is that there is a r and an e, and both may contain an identity value or some other value. Later, when we have defined our first `PetriClass`, we shall define some subclasses of arcs as well.

4.3 Steps

A step is an execution of actions. In its most simple form, it is nothing more than firing a single enabled transition in a given marked Petri net. Doing so results in a new marked Petri net, but does not change the `PetriStructure` or `PetriClass`.

A *firing sequence* is a sequence of consecutively executed actions, and as such may be seen as a list of steps. By performing steps in sequence, a firing sequence is created out of the transitions that are chosen to be executed.

In the following subsections, we will use a method similar to the "policies" of [DKPY08] to extend the singleton step into more complex steps, by changing the net structure so that it represents these new steps as single steps.

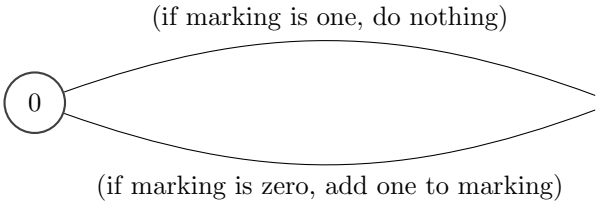

Figure 9: Two arcs that would allow their connected transitions to fire in sequence, but not simultaneously: the upper arc first, then the lower arc.

4.4 Concurrency and auto-concurrency

Nets may also express concurrent behaviour, where multiple transitions may fire simultaneously in a single step. The concept of "simultaneous" was already used before (in Demarcation 9) and the combination operator is defined in terms of it. In other words, the contextual meaning of "firing simultaneously" in a given `PetriClass` is fully defined by the combination operator of that `PetriClass`.

Auto-concurrent behaviour, where a given transition may fire simultaneously with itself, is possible as well.

Going into the territory of concurrency, conflicts become important. These are a type of resource contention, where the range of two arc labels each separately is enabled, but their combined arc label may not be. Combining certain arc labels together may prevent further arc labels from being added to this combination and the other way around, creating a choice between them. When a choice like this needs to be made, the arcs are said to be in conflict.

Simultaneous and auto-simultaneous behaviour are similar to transactions. In transactions multiple steps are executed in sequence, but considered as if they were a single step. This is relevant, because transactions are still a subject of research. In [BM97, BM00b, BM00a, BM01] a lot of groundwork is laid to generalize transactions.

Transactions and simultaneous behaviour are not the same, however: it is possible for transitions connected to two arcs to be able to fire in sequence, but not simultaneously (example in Figure 9). Similarly, it is possible for transitions connected to two arcs to be able to fire simultaneously but not in sequence (example in Figure 10).

The principles used in this section are heavily based on those used for transactions in the before mentioned [BM97, BM00b, BM00a, BM01]. Specifically, the methods used to "grow" transitions in Algorithms 4, 5 and 6 are similar to how transactions can be created.

The combination operator of a `PetriClass` underlies the contextual meaning of simultaneous firing, and as such, we can actually use the combination operator to generalize the simultaneous firing of multiple transitions to new singleton transitions. This allows us to use the same method on all step types, turning each step type into a single transition being fired.

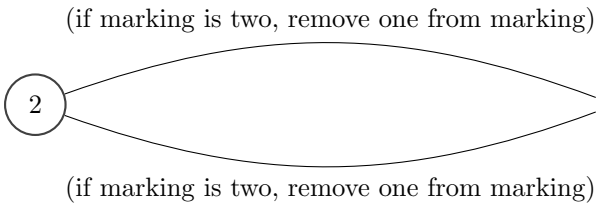

Figure 10: Two arcs that would allow their connected transitions to fire simultaneously, but not in sequence.

Demarcation 12 *The* (auto-)concurrent enabledness and firing of any combination of transitions *can be derived from a single "super transition" where all arcs connected to the original transitions are copied and these copies connected to this new super transition; if a single transition is present multiple times, the arcs connected to this transition must also be copied that same amount of times.*

Because both the enabledness check (Definition 5) and firing (Definition 6) of transitions combine arcs going to the same place into a single arc, these super transitions end up using the combination operator to resolve multiple arcs going into the same place. Since simultaneous behaviour for any specific `PetriClass` is defined in terms of its combination operator, this means a super transition thus behaves equal to the simultaneous firing of the transitions it was created from.

Definition 7 *Let $N = (P, T, A)$ be a `PetriStructure` and $\mathbb{S} : T \to \mathbb{N}$ be a multiset of transitions. Then $\mathbb{T}(\mathbb{S})$ is a new* super transition. *For each arc $a \in A$ connecting a transition $t \in \mathbb{S}$ to a place $p \in P$, we add $\mathbb{S}(t)$ new arcs b, such that:*

- $b_p = a_p$
- $b_t = \mathbb{T}(\mathbb{S})$
- $b_r = a_r$
- $b_e = a_e$

An example of the creation of a super transition from two transitions is shown in Figure 11.

Definition 7 describes how to combine multiple transitions into a single super transition. The resulting super transition itself is also a transition, and thus can be used to create further super transition. Since the combination operator ($\otimes$) used on the arcs in this process is both commutative and associative, the order in which transitions are combined does not matter as the resulting super transition will be the same. We will now introduce notation that allows us

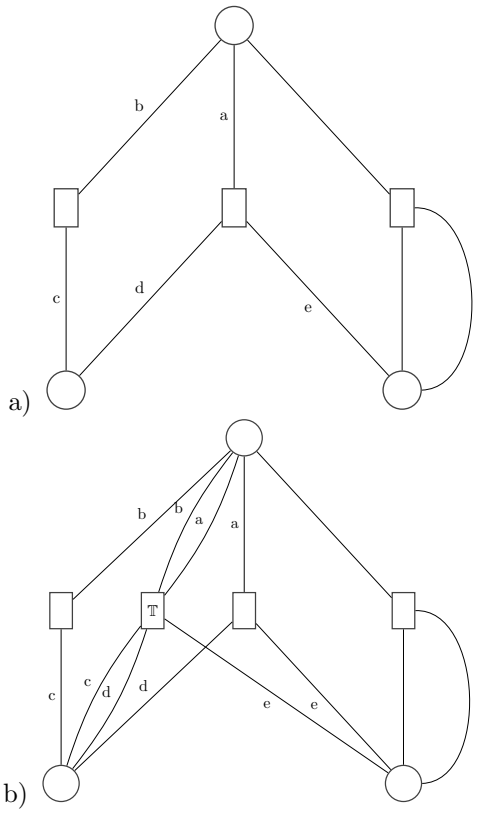

Figure 11: The leftmost two transitions in *(a)* are combined into a single super transition $\mathbb{T}$ in *(b)* that has the same behaviour as those two transitions would have when fired simultaneously. The arc labels have been simplified to single letters to condense space and preserve the ability to clearly distinguish the various labels of the involved arcs.

to easily combine transitions into a new super transition. This is a transition analogue to the combination operator for arcs.

Demarcation 13 *The expression $a \oplus b$ where $a, b \in T$ means to create a new super transition, using the method to create super transitions as described in Definition 7 with $\mathbb{S}(a) = 1$ and $\mathbb{S}(b) = 1$ and $\mathbb{S}(c) = 0$ for all $c \in T$ where $c \neq a$ and $c \neq b$.*

5 Algorithms for experimental analysis of marked Petri nets

First let us be clear about what is meant here by experimental analysis. In general, two types of analysis can be done on (marked) Petri nets:

- Experimental

  This involves "running" the net under consideration (simulating its behaviour) by observing the effects on the marking of a firing sequence. This approach generally requires a great number of simple steps to be performed, where not much (if any) insight into the inner workings of the net is required. In particular, when creating a firing sequence any conflicts are resolved automatically as they turn into choices.

- Theoretical

  This involves identifying and classifying the net under consideration, making statements about the structure of its (potential) behaviour. It includes things such as reachability, locating and/or identifying deadlocks and invariants plus anything else that doesn't involve running the net at any point. This approach generally requires more insight into the inner workings of a net, as opposed to sheer computational power.

While both approaches can be automated, the experimental approach is more generally applicable, more repetitive and requires the least amount of understanding about what is being done. This makes it a great candidate for automation in a general sense. Additionally, the theoretical approach is already covered by the vast majority of existing literature on Petri nets.

That is why in this paper, we focus on the experimental approach. After all, the goal of the methods described in this paper is to achieve a high degree of automation and speed.

For the purpose of experimenting, using all the principles and definitions from this paper so far, we can now construct algorithms that extend steps (see Subsection 4.3). We shall extend steps from single transitions to (multi)sets of transitions, thus covering simultaneous (auto-concurrent) firing.

5.1 Single transition steps

The step consisting of a single transition is the simplest case, not requiring any special treatment. While its algorithm may be straightforward from the explanation of steps (Subsection 4.3), we have not yet formally written it down. In order to be complete, this is the basic algorithm for single transition steps:

Algorithm 1 *To perform a* step consisting of a single transition *for a given marked Petri net* $N = ((P, T, A), C, M)$ *with finite* T*, do the following:*

1. *Create a set E of all enabled transitions, using Definition 5 to determine enabledness for all $t \in T$.*

2. *Pick any transition $t \in E$ and fire it using Definition 6.*

This method requires that T is finite. Note that conflicts are hidden because in step two, "any" transition is picked, and thus a choice is made randomly, avoiding conflicting situations and making simultaneous firing impossible.

A less efficient algorithm that only has the requirement that T is countable (so a random t can be picked reliably) is the following:

Algorithm 2 *To perform a* step consisting of a single transition *for a given marked Petri net $N = ((P, T, A), C, M)$ with countable T, do the following:*

1. *Pick a random transition $t \in T$.*

2. *Using Definition 5, determine enabledness for t.*

3. *If t is enabled, fire it using Definition 6 and stop.*

4. *Otherwise, start over from 1.*

This algorithm does not create a list of enabled transitions to pick from, but instead picks a random transition and checks if it is enabled. Sadly, many non-enabled transitions can be picked before an enabled one is found. The complexity of Algorithm 1 is $O(|T|)$ (because each transition is checked exactly once), while the complexity of Algorithm 2 is potentially unknown and the algorithm may never finish at all (for example when there are infinitely many transitions in T, none of them enabled).

5.2 Concurrent transition steps

Next we will consider the case of transitions firing simultaneously with other transitions (but not with themselves).

Note that from this point onward, we are making an important assumption: we assume that combining a disabled arc with any other arc will always produce a disabled arc. In other words, for the below algorithms we require that any combination of arcs never has a wider range than any of the arcs it was created from. As such, the algorithms from this point onward can only be used if this condition holds for the combination operator of the `PetriClass` of the marked Petri net they are being used on.

To calculate concurrent steps, the `PetriStructure` is changed by adding a new super transition $\mathbb{T}$ for each choice of transitions in T, thus defining a new net. This new net simulates concurrent steps in the original net, and we can use the same singleton step methods from Algorithms 1 or 2.

Algorithm 3 *To create a new marked Petri net* $N' = ((P, T', A'), C, M)$ *representing* concurrent transition steps *for a given marked Petri net* $N = ((P, T, A), C, M)$ *with finite* T*, let* T' *and* A' *be defined by for every possible finite subset* $\mathbb{S}$ *of* T*, extending* T' *with a new super transition* $\mathbb{T}(\mathbb{S})$ *as described in Definition 7.*

Note that (as mentioned in the preliminaries) (sub)sets can be seen as multisets, so Definition 7 may be used to construct super transitions even though it only works on multisets and we are dealing with a set here.

If T is finite, then T' is also finite because it is constructed from all possible combinations of elements from T. Then the number of transitions in T' is at most $2^{|T|}$ (each transition has two possibilities: it is either in the set, or it is not).

The main advantage of using Algorithm 3 is that it need only be done once, during initialization, before the net is used. Afterwards, Algorithm 1 can be used to calculate the steps. This makes it a very generic method to handle concurrency with a one-time cost regardless of the amount of steps that are run.

Alternatively to using Algorithm 3, it is possible to change Algorithm 2 to calculate concurrent transition steps. This is less efficient than using Algorithm 3 but can be used when T is infinite but countable.

This method works by "growing" super transitions, using the $\oplus$ operator as mentioned in Demarcation 13. We keep adding transitions to a super transition that do not disable the super transition, until at a randomly chosen point we decide to fire it:

Algorithm 4 *To calculate a* concurrent transition step *for a given marked Petri net* $N = ((P, T, A), C, M)$ *with infinite but countable* T*, do the following:*

1. $T' = \emptyset$
2. *Pick a random transition* $t \in (T \setminus T')$*. (If* $T \setminus T' = \emptyset$*, stop.)*
3. *Add* t *to* T'*.*
4. *Determine if* t *is enabled, using Definition 5.*
5. *If* t *is not enabled, continue from 2 onward.*
6. *Randomly, either fire transition* t *and stop, or continue.*
7. *If* $T \setminus T' = \emptyset$*, fire transition* t *and stop.*
8. *Pick a random transition* $t' \in (T \setminus T')$*.*
9. *Add* t' *to* T'*.*

10. *Determine if $t \oplus t'$ is enabled, using Definition 5.*

11. *If $t \oplus t'$ is not enabled, continue from 6 onward.*

12. *Set $t := t \oplus t'$.*

13. *Continue from 6 onward.*

Note that because of the nature of infinite sets, the above algorithm has a tendency to prefer combinations of smaller amounts of transitions over combinations of higher amounts of transitions, and is not guaranteed to pick fairly from the various possible transitions that are enabled. This is because it is impossible to iterate over an infinite set. As such, using Algorithm 3 to create a new net and then using Algorithm 1 is preferred and should be used when T is not infinite.

5.3 Auto-concurrent transition steps

Now we consider the case of transitions firing concurrently with other transitions including itself.

We would simply apply Algorithm 3 again, using a multiset $\mathbb{S}$ on T instead of a subset. There is a problem with this, that the number of multisets definable for a given non-empty set is infinite even if the set is finite. This means step 2 of Algorithm 3 would never complete.

Instead, we will use a modified version of Algorithm 4, changed to deal with multiple occurrences of transitions by removing the set that keeps track of used transitions (T') from the algorithm:

Algorithm 5 *To calculate an* auto-concurrent transition step *for a given marked Petri net $N = ((P, T, A), C, M)$ with countable T, do the following:*

1. *Pick a random transition $t \in T$.*

2. *Determine if t is enabled, using Definition 5.*

3. *If t is not enabled, start over from 1.*

4. *Randomly, either fire transition t and stop, or continue.*

5. *Pick a random transition $t' \in T$.*

6. *Determine if $t \oplus t'$ is enabled, using Definition 5.*

7. *If $t \oplus t'$ is not enabled, continue from 4 onward.*

8. *Set $t := t \oplus t'$.*

9. *Continue from 4 onward.*

Since this algorithm is based on Algorithm 4, it has the same disadvantage

of having a bias towards smaller super transitions, and we'd like to avoid it if at all possible.

Luckily, it is possible to alter the algorithm somewhat to make it fully consider all options equally, under the assumptions that T is finite and no transition can be fired auto-concurrently infinitely. If these assumptions are true, we can do a full depth search to list all possible enabled super transitions created from T, and then pick one from that list, giving them all an equal chance to occur without bias. This is possible because we know in advance that we will not be able to keep adding transitions indefinitely and thus will not enter an infinite loop.

This is a true limitation (but a natural one) on the marked Petri nets it can be used on, as it means each transition must have at least one place connected to it with an arc of limited range and an arc that causes a non-identity effect on the marking of this same place (this may be the same arc, but need not be). In all other cases it would be possible to fire this transition infinite times once enabled.

The new algorithm looks as follows:

Algorithm 6 *To calculate an* auto-concurrent transition step *for a given marked Petri net $N = ((P, T, A), C, M)$ with finite T and each transition in T having a connected arc a where $a_r \neq \mathbb{I}_r$ and a connected arc b where $b_e \neq \mathbb{I}_e$ and $a_p = b_p$, do the following:*

1. *$A = \emptyset$, $B = \emptyset$, $C = \emptyset$*
2. *For each transition $t \in T$:*
   - *Determine if t is enabled, using Definition 5.*
   - *If t is enabled, add t to A, B and C.*
3. *For each transition $t \in A$:*
   - *For each transition $t' \in B$:*
     - *Determine if $t \oplus t'$ is enabled, using Definition 5.*
     - *If $t \oplus t'$ is enabled, add $t \oplus t'$ to C and A.*
   - *Remove t from A*
4. *If $A \neq \emptyset$, go to 3.*
5. *Fire a random transition in C.*

This algorithm simply creates all possible enabled super transitions and then picks one from the list when all possibilities have been determined.

5.4 Maximal stepping

An additional requirement on (auto-)concurrent steps that are *maximal* (as specified in [Bur83]) is that it will only ever fire an enabled (super or regular)

transition if it cannot be extended with more transitions in the net (using $\oplus$) without disabling it.

To do so, we can alter Algorithm 4 as follows to calculate maximal concurrent steps:

Algorithm 7 *To calculate a* maximal concurrent transition step *for a given marked Petri net* $N = ((P, T, A), C, M)$*, do the following:*

1. *Create a new empty set of transitions* T'*.*
2. *Pick a random transition* $t \in (T \setminus T')$*.*
3. *Add* t *to* T'*.*
4. *Determine if* t *is enabled, using the method from Definition 5.*
5. *If* t *is not enabled, continue from 2 onward.*
6. *Set* $\mathbb{T} := t$
7. *If* $T \setminus T' = \emptyset$*, fire transition* $\mathbb{T}$ *and stop.*
8. *Pick a random transition* $t' \in (T \setminus T')$*.*
9. *Add* t' *to* T'*.*
10. *Determine if* $\mathbb{T} \oplus t'$ *is enabled, using the method from Definition 5.*
11. *If* $\mathbb{T} \oplus t'$ *is enabled, set* $\mathbb{T} := \mathbb{T} \oplus t'$*.*
12. *Continue from 7 onward.*

And as follows to calculate maximal auto-concurrent steps:

Algorithm 8 *To calculate a* maximal auto-concurrent transition step *for a given marked Petri net* $N = ((P, T, A), C, M)$*, do the following:*

1. *Create a new empty set of transitions* T'*.*
2. *Pick a random transition* $t \in (T \setminus T')$*.*
3. *Determine if* t *is enabled, using the method from Definition 5.*
4. *If* t *is not enabled, add* t *to* T' *and continue from 2 onward.*
5. *Set* $\mathbb{T} := t$
6. *If* $T \setminus T' = \emptyset$*, fire transition* $\mathbb{T}$ *and stop.*
7. *Pick a random transition* $t' \in (T \setminus T')$*.*
8. *Determine if* $t \oplus t'$ *is enabled, using the method from Definition 5.*

9. *If* $\mathbb{T} \oplus t'$ *is not enabled, add* t' *to* T' *and continue from 6 onward.*

10. *Set* $\mathbb{T} := \mathbb{T} \oplus t'$.

11. *Continue from 6 onward.*

Being maximal steps, neither of these two algorithms suffer from the problem their non-maximal variants have, where there is a bias towards smaller transition combinations. After all, these smaller combinations are not allowed to be fired at all in maximal steps. Hence it is not necessary to determine all possible enabled transition combinations and we can simply grow a single arbitrary combination, instead.

6 Natural Petri nets

The previous sections go about as far as possible into the theory of marked Petri nets without defining C in the tuple $N = (S, C, M)$. We will now define a class of Petri nets (i.e.: nets that share a single `PetriClass`, as mentioned in Definition 2) that we shall call *natural Petri nets*. All of these nets share the same `PetriClass` that we shall call `NatNetClass`.

In the `NatNetClass` `PetriClass` the domain for the markings is represented by natural numbers, the domain for range is $\mathbb{N}_\infty$, the domain for effect is $\mathbb{Z}$, the range function is a simple bounds check, the effect function is an addition and the combination operator consists of a near-trivial hybrid between addition and taking the minimum and maximum.

`NatNetClass` can be seen as the most immediately intuitive instantiation of `PetriClass`, and represents one of the most commonly accepted definitions of a Petri net: an extended net as described in [Pet77] and later further explained in more detail in [Mur89] and [DR98]. It fully expresses extended Petri nets as described, with multiplicity on the arcs (and even with the inhibitor arc extension).

The following definition is quite a bit of theory all at once, and thus is explained step by step immediately afterwards.

Definition 8 *The* `PetriClass` `NatNetClass` $= ((D_r, D_e, D_m), f_r, f_e, L, \otimes, \mathbb{I})$ *where:*

- D_r *consists of tuples* (u, l, h)*, where* $l, h \in \mathbb{N}_\infty$ *and* $u \in \mathbb{N}$.
- D_e *is* $\mathbb{Z}$.
- D_m *is* $\mathbb{N}$.
- $f_r((u, l, h), v) = \left\{ \begin{array}{c} \texttt{true if } l \leq v \leq h \texttt{ and } u \leq v \\ \texttt{false otherwise} \end{array} \right\}$, *where* $(u, l, h) \in D_r$ *and* $v \in D_m$
- $f_e(e, v) = v + e$*, where* $e \in D_e$ *and* $v \in D_m$
- L *comprises those* $((u, l, h), e)$ *where* $((u, l, h), e) \in D_r \times D_e$ *and* $u \geq -e$.
- $((u_1, l_1, h_1), e_1) \otimes ((u_2, l_2, h_2), e_2) = (u_1 + u_2, \texttt{max}(l_1, l_2), \texttt{min}(h_1, h_2)), e_1 + e_2)$
- $\mathbb{I} = ((0, 0, \infty), 0)$

For convenience, we will define a shorthand for Petri nets with the `NatNetClass` `PetriClass`:

Definition 9 *A* natural Petri net *or* `NatNet` $= (S, M)$ *is a marked Petri net* (S, C, M) *where* $C =$ `NatNetClass`*.*

Natural Petri net domain

The marking domain of a `NatNet` is the natural numbers ($\mathbb{N}$). In other words: all positive whole numbers. This means markings may never become negative in `NatNet`s (since negative numbers are not in $\mathbb{N}$).

The range domain of a `NatNet` is a tuple (u, l, h), where $u, l, h \in \mathbb{N}_\infty$. The formal meaning is defined in subsection 6. l and h represent the lower and higher bounds of the range, with the intended meaning that it is connected to a transition that is enabled if the markings corresponding to the connected places are within the bounds. u represents the amount of tokens "used" by the transition. We need to differentiate between the lower bound and the actually used amount to accommodate the difference between checking tokens without using them and arcs that use tokens. An example of this can be found in subsection 6. The lower bound may be higher than the upper bound. Such labels permanently disable any connected transitions. As an example, the nonsensical range $(3, 2, 1)$ will never enable any transitions, as there are no values v that can make the statement $2 \leq v \leq 1$ true. The addition of infinity to the domain of the range means we can also define ranges that have no upper bound (when $h = \infty$), or are never enabled (when $l = \infty$ or $u = \infty$) without the need to resort to nonsensical values like above.

The effect domain of a `NatNet` is $\mathbb{Z}$ instead of $\mathbb{N}$ to allow the marking to go either up or down with all whole numbers. Since there is no upper limit (capacity) on the markings of places, any zero or positive effect is always a valid. Negative values could bring the marking outside of $\mathbb{N}$, however — so through the restrictions placed on the set of arc labels we only allow effects that are guaranteed to not make any marking negative (see subsection 6 for details).

Range and effect functions in `NatNet`s

The range and effect functions of a `NatNet`are both near-trivial, and have been chosen to most closely match their intuitive meaning:

The range function of a `NatNet` is:

$f_r((u, l, h), v) = \begin{Bmatrix} \texttt{true if } l \leq v \leq h \texttt{ and } u \leq v \\ \texttt{false otherwise} \end{Bmatrix}$, where $(u, l, h) \in D_r$ and $v \in D_m$.

In other words, only when $l \leq v \leq h$ and $u \leq v$ will arcs enable connected transitions. In plain English: the current value of the marking must be between the lower and upper bound of the range, inclusive, and the current value of the marking must be at least the amount of used tokens. A quite straightforward requirement which should not need further explanation.

The effect function of a `NatNet` is $f_e(e, v) = v + e$. In plain English: add the effect of the arc to the connected place's current value of the marking to

produce the new value of the marking for that place.

Allowed arc labels in `NatNets`

The requirement that $((u, l, h), e) \in D_r \times D_e$ is already a requirement for all marked Petri nets. It is not specific to `NatNets`.

`NatNet` add one other requirement to the set of arc labels, however. $u \geq -e$ must hold, to make sure the effect of an arc will never bring any given marking outside of D_m. We're requiring the negative effect of an arc to never be larger than the lower bound on the range. This means the effect function will never reduce the connected place's marking below zero for any enabled transition, as it can only be enabled when $l \leq$ the current marking while $u \leq l$ and $u \geq -e$, thus $u - e \geq 0$.

The combination operator of `NatNets`

As stated in Demarcation 9, the combination operator defines how any two arcs connected to a shared place may be combined into a single arc with equal behaviour as the two arcs behaving simultaneously. In order to keep the definition of Petri nets generic, the concept of "simultaneous behaviour" is not covered at all by Petri nets. It is mentioned, but never defined or even explained. This is with good reason: by tying the term to the combination operator, we move the task of defining what simultaneous behaviour is to the combination operator itself. To clarify: what for any given `PetriClass` is considered simultaneous behaviour, is defined by the semantics of that `PetriClass`'s combination operator.

In the specific case of `NatNets`, the combination operator formalizes the simultaneous behaviour as seen in a traditional extended Petri net (as in [Pet77, Mur89, DR98]): the used tokens and the effects are combined by accumulating them, the ranges are combined by taking their intersection.

Taking the above quite literally, the following combination operator can be formally written down:

$$\begin{gathered}((u_1, l_1, h_1), e_1) \otimes ((u_2, l_2, h_2), e_2) = \\ (u_1 + u_2, \mathtt{max}(l_1, l_2), \mathtt{min}(h_1, h_2)), e_1 + e_2)\end{gathered}$$

Intuitively, this already looks correct, and using only the $+$, `min` and `max` operators without mixing them guarantees that this operator is both commutative and associative (since each of the operators has those properties), which are the requirements for the combination operator in `PetriClass`es.

Since $u \geq -e$ is already known to be true for both arcs going into the combination operator, that means $u_1 + u_2 \geq -(e_1 + e_2)$, and thus $u \geq -e$ holds in the new arc as well.

Some examples of arcs being combined are given in Figures 12 through 15, with a table of further examples given in Figure 16.

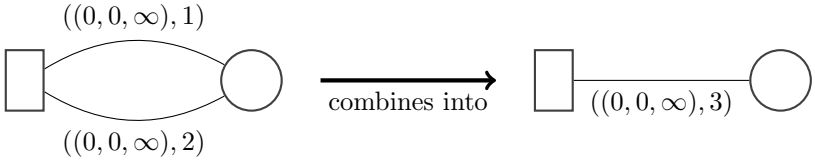

Figure 12: Combination operator on two arcs that add tokens

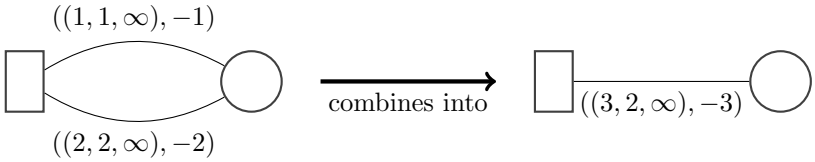

Figure 13: Combination operator on two arcs that remove tokens

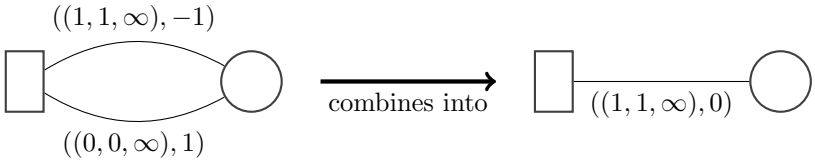

Figure 14: Combination operator on an arc that removes tokens and an arc that adds tokens

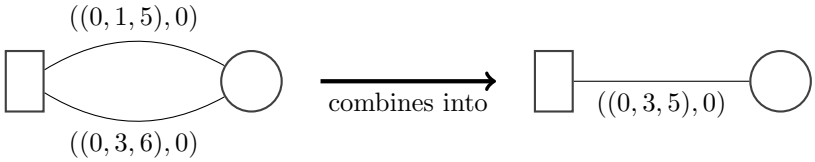

Figure 15: Combination operator on two arcs that test for specific token counts.

a_1	a_2	$a_1 \otimes a_2$
$((1,1,\infty),0)$	$((0,1,\infty),0)$	$((1,1,\infty),0)$
$((1,1,\infty),0)$	$((0,0,\infty),1)$	$((1,1,\infty),1)$
$((0,5,6),0)$	$((0,2,3),0)$	$((0,5,3),0)$
$((5,5,6),0)$	$((2,2,3),0)$	$((7,5,3),0)$
$((0,5,6),8)$	$((0,2,3),3)$	$((0,5,3),11)$
$((0,5,6),0)$	$((0,0,\infty),0)$	$((0,5,6),0)$
$((100,\infty,\infty),1)$	$((0,0,\infty),1)$	$((100,\infty,\infty),2)$
$((5,10,20),0)$	$((0,0,\infty),5)$	$((5,10,20),5)$
$((5,10,20),0)$	$((2,0,\infty),5)$	$((7,10,20),5)$

Figure 16: Combination operator examples in table form.

$(L, \otimes, \mathbb{I})$ is a commutative monoid

The requirement that $(L, \otimes, \mathbb{I})$ is a commutative monoid is met:

- $\otimes$ can never bring any two values that are in L to result in a value not in L, since $l_1 + l_2 \geq -(e_1 + e_2)$ is true for all values where $l_1 \geq -e_1$ and $l_2 \geq -e_2$.
- $\otimes$ is commutative and associative (see the paragraph preceding this one for details).
- $\mathbb{I}$ is the identity for $\otimes$ since $i + 0 = i$, $\texttt{max}(i, 0) = i$ and $\texttt{min}(i, \infty) = i$ within the respective domains where these operators are used.

7 Properties of NatNets

NatNets are a way to write down existing Petri net classes (extended and non-extended Petri nets as in [Pet77, Mur89, DR98]) in a new generalized manner, combined with a calculation method that works on this notation.

As such, the properties of these nets are quite interesting. So in this section we will explore some of the properties of Petri nets in this notation, and NatNets specifically.

7.1 Limitations to the expressive abilities of a single arc

Both Petri nets and NatNets focus on local effects only, meaning markings can only be affected by transitions and arcs that are connected to their respective place. This does not actually limit expressive abilities of the net itself, only the expressive abilities of a single arc. By using multiple arcs and/or multiple transitions, non-local behaviour can still be expressed indirectly.

Similarly, NatNets have a few restrictions that limit the expressive abilities of a single transition, while not limiting the expressive abilities of the net itself:

- The effect of an arc deals with a known static (non-variable) amount of tokens (thus reset or set arcs are not possible). This can be replaced by a construction that adds or removes tokens combined with an arc that checks for a specific range.
- Ranges may not have "gaps" or consist of multiple non-contiguous ranges. This can be accomplished by using multiple transitions, one for each contiguous section of the wanted non-contiguous range.

That first limitation is only there to keep NatNets straightforward and simple. It is quite possible to extend NatNets with reset and set arcs, and this is actually done later in section 8.

The second limitation is purely to keep arc labels readable, as allowing non-contiguous ranges would make them very hard to note down while not increasing the computational power of NatNets.

7.2 Arc label subtypes

In subsection 4.2 we already introduced the four main arc label types. Now that we have domains for the range and effect, we can extend these types with subtypes.

A full list in table form of all arc label subtypes is shown in Figure 17.

Since the range in a NatNet is a 3-tuple (u, l, h), this gives us eight possibilities for check arc label subtypes (which have a range but no effect), following the original reasoning from subsection 4.2: used tokens or not, lower bound or not, upper bound or not. That is $2^3 = 8$ combinations.

However, the case where there is no used token amount, no lower bound and no upper bound is not a range arc label (but rather an identity arc label) and

the option where there is a used token count but no lower bound effectively does have a lower bound because of the used token count. This leaves six subtypes for check arc labels:

- Lower-bounded check arc labels: $((0, l, \infty), 0)$ with $l > 0$

  These arc labels do not use tokens, but do have a lower bound on the activating token range. These arc labels are analogous to activator arcs in extended Petri nets.

- Upper-bounded check arc labels: $((0, 0, h), 0)$ with $h < \infty$

  These arc labels do not use tokens, but do have an upper bound on the activating token range. These arc labels are analogous to inhibitor arcs in extended Petri nets as described in [Pet77].

- Range arcs $((0, l, h), 0)$ with $0 < l \leq h < \infty$

  These arc labels do not use tokens, but do have both an upper bound as well as a lower bound on the activating token range. These arc labels represent the combination of an activator arc and an inhibitor arc, and this combination has already been described as a range in [KK10].

- Lower-bounded self-loops $(((u, l, \infty), 0)$ with $u > 0$ and $l > 0)$, upper-bounded self-loops $(((u, 0, h), 0)$ with $u > 0$ and $h < \infty)$, and range self-loops $(((u, l, h), 0)$ with $u > 0$ and $l > u$ and $h < \infty)$.

  These arc labels are a form of self-loop: an equal amount of tokens goes in and goes out, giving a net effect of zero. They can have a lower bound for their activating token range, and may or may not have an upper bound for their activating token range. As such, there can be lower-bounded self-loops, upper-bounded self-loops and range self-loops.

Radical arc labels (which have effect, but no range) can be divided into source arc labels $(((0, 0, \infty), e)$ with $e > 0)$ and sink arc labels $(((0, 0, \infty), e)$ with $e < 0)$. Since removing tokens unconditionally can lead to a negative amount of tokens, sink arc labels may not appear in `NatNets`. This is enforced by the $u \leq -e$ requirement on L.

Conditional arc labels can be broadly split into two main categories, one based on arc labels with positive effect and one based on arc labels with negative effect. Within these one can further classify the various check arc label types, but it more interesting to base the classification on the amount of tokens being used, since this tells us more about what the contextual effect of an arc label really is. This leads to the following conditional arc types:

- Amplification arc labels $(((u, l, h), e)$ with $u > 0$ and $e > 0)$

  These arc labels use tokens to add tokens to a place, thus amplifying the token count.

Major type	Subclass	Range	Effect
Identity	—	$(0, 0, \infty)$	0
Check	Lower-bounded	$(0, l, \infty)$, $l > 0$	0
	Upper-bounded	$(0, 0, h)$, $h < \infty$	
	Range	$(0, l, h)$, $l > 0$, $h < \infty$	
	L-bounded self-loop	(u, l, ∞), $u > 0$, $l > 0$	
	U-bounded self-loop	$(u, 0, h)$, $h < \infty$, $u > 0$	
	Ranged self-loop	(u, l, h), $u < 0$, $l > 0$, $h < \infty$	
Radical	Sink	$(0, 0, \infty)$	< 0
	Source		$e > 0$
Conditional	Amplification	(u, l, h), $u > 0$	> 0
	Bootstrap	$(0, 0, h)$, $h < \infty$	> 0
	Clearing	$(-e, -e, -e)$	< 0
	Dampening	(u, l, h), $u \geq -e$	< 0
	Negating	(u, l, h), $u < -e$	< 0

Figure 17: A full list of all theoretically possible arc label types in `NatNet`s. The sink and negating arc label types are not allowed in `NatNet`s, but included for completeness.

- Bootstrap arc labels $(((u, l, h), e)$ with $u = 0$ and $e > 0)$

  These arc labels add tokens to a place without using tokens, thus bootstrapping the token count.

- Clearing arc labels $(((u, l, h), e)$ with $u = l = h = -e$ and $e < 0)$

  These arc labels decrease the token count to exactly zero.

- Dampening arc labels $(((u, l, h), e)$ with $u \geq -e$ and $e < 0)$

  These arc labels decrease the token count by a set amount.

- Negating arc labels $(((u, l, h), e)$ with $u < -e$ and $e < 0)$

  These arc labels can decrease the token count to below zero. This arc label type is not allowed in `NatNet`s because it violates the $u \geq -e$ requirement on arc labels.

8 PetriCalc: Implementation in code

In order to to show the ability to do fast calculations of the model that is presented in this paper, a practical implementation has been written in C++ that was named PetriCalc. PetriCalc is a NatNet calculation tool that simulates running NatNets, following the ideas from this paper as closely as possible.

If reading this paper in digital PDF format, the full source code to the latest version of PetriCalc at time of writing should be attached as a PDF attachment. Alternatively, the latest version can be obtained at https://github.com/Thulinma/PetriCalc. Some of the most interesting parts will be highlighted in the rest of this section, but having a copy of the full source available makes it easier to follow along.

Converting (extended) Petri nets as used in the Snoopy tool

In order to facilitate input (PetriCalc does purely simulation and nothing else — there is no interface whatsoever), PetriCalc reads Snoopy ([HHL+12]) XML files containing Snoopy-style extended Petri nets (.spept file extension) or Snoopy-style Petri nets (.spped file extension). These files are converted to a NatNet in-memory during load.

The Snoopy tool ([HHL+12]) defines five arc types for extended Petri nets. By converting these arc types into NatNet arcs, we can translate any Snoopy-created extended Petri net into a NatNet as described in the earlier sections. This is a description of the behaviour of the arc types available in Snoopy and how to write them as natural net arcs:

- An "edge" will either increase or decrease the marking of a place, where possible. With weight x these can be labelled either $((0, 0, \infty), x)$ (when coming out of a transition) or $((x, 0, \infty), -x)$ (when going into a transition).
- A "read edge" does not consume or produce any tokens and may only fire if the tokens that would have been consumed otherwise are available. With weight x these can be labelled as $((0, x, \infty), 0)$.
- An "inhibitor edge" does not consume or produce any tokens and may only fire if the tokens that would have been consumed otherwise are *not* available. With weight x these can be labelled as $((0, 0, x - 1), 0)$.
- An "equal edge" does not consume or produce any tokens and may only fire if there are exactly its weight in tokens available in the connected place. With weight x these can be labelled as $((0, x, x), 0)$.
- A "reset edge", when fired will consume all available tokens instead of the amount indicated by its weight. These cannot be modelled within NatNets, so we have extended NatNets to allow for reset arcs.

In order to extend NatNets to allow reset arcs to be used, D_e had to be changed from $\mathbb{Z}$ to a tuple (e, a) where $e \in \mathbb{Z}_S$ and $a \in \mathbb{N}$. Here $\mathbb{Z}_S$ is $\mathbb{Z}$ extended with Si where $i \in \mathbb{Z}$, with the S-version of any number being a "setter". To calculate addition for $\mathbb{Z}_S$ the rules $n+Si = S(n+i), Si+n = S(i+n), Si+Sn = S(i+n)$ apply, where $n, i \in \mathbb{Z}$.

The effect and range functions do not need to be changed and can be used as-is. The combination operator requires a change though:

$\otimes(((u_1, l_1, h_1), (e_1, a_1)), ((u_2, l_2, h_2), (e_2, a_2))) =$
$((u_1 + u_2, \texttt{max}(l_1, l_2), \texttt{min}(h_1, h_2)),$
$(\texttt{if } e_1 + e_2 \notin \mathbb{Z} \texttt{ then } \texttt{max}(S0, a_1) + \texttt{max}(S0, a_2)) \texttt{ else } e_1 + e_2,$
$\texttt{max}(0, a_1) + \texttt{max}(0, a_2))$
$)$

The reason we need the extra effect value is to keep track of the sum of all positive effects so far. When a reset arc is used, we need to change this into a set-arc to the value of the sum of all positive effects. Since the normal effect only keeps track of the total effect as opposed to all changes, we need an extra value that keeps track of the positive changes only. The S symbol is used to represent set-arcs and their behaviour.

The code used to load Snoopy arcs into this NatNet-based representation is the following:

```
1  /// \brief Adds a single arc to the net, from a Snoopy XML ↩
       file.
2  ///
3  /// This function combines arc labels using the combination ↩
       operator if an arc between the same place and transition↩
        already exists.
4  /// The result of this is that arc labels never need be ↩
       combined later, as they have been combined right here ↩
       during net load already.
5  void PetriNet::addEdge(TiXmlNode * N, unsigned int E){
6    TiXmlElement * e;
7    e = N->ToElement();
8    const char * id = e->Attribute("id");
9    if (!id){return;}
10   unsigned long long SOURCE = atoi(e->Attribute("source"));
11   unsigned long long TARGET = atoi(e->Attribute("target"));
12   unsigned long long transition;
13   unsigned long long place;
14   long long multiplicity = 1;
15   if (transitions.count(SOURCE)){
16     transition = SOURCE;
17     place = TARGET;
18   }
19   if (places.count(SOURCE)){
20     transition = TARGET;
21     place = SOURCE;
```

```
22    }
23    TiXmlNode * c = 0;
24    while ((c = N->IterateChildren(c))){
25      e = c->ToElement();
26      if (!e){continue;}
27      if (!e->Attribute("name")){continue;}
28      std::string name = e->Attribute("name");
29      if (name == "Multiplicity"){
30        multiplicity = atoi(e->GetText());
31      }
32    }
33
34    //If the place is the source, everything is negative
35    if (place == SOURCE){
36      multiplicity *= -1;
37    }
38
39    unsigned long long aRLow, aRHigh, aRUsed;
40    long long aEffect;
41    bool aEffectSetter = false;
42
43    if (E == EDGE_NORMAL){
44      if (multiplicity < 0){
45        aRLow = -multiplicity;
46      }else{
47        aRLow = 0;
48      }
49      aRUsed = aRLow;
50      aRHigh = INFTY;
51      aEffect = multiplicity;
52    }
53    if (E == EDGE_ACTIVATOR){
54      if (multiplicity < 0){
55        aRLow = -multiplicity;
56      }else{
57        aRLow = multiplicity;
58      }
59      aRUsed = 0;
60      aRHigh = INFTY;
61      aEffect = 0;
62    }
63    if (E == EDGE_INHIBITOR){
64      if (multiplicity < 0){
65        aRHigh = -multiplicity - 1;
66      }else{
67        aRHigh = multiplicity - 1;
68      }
69      aRUsed = 0;
70      aRLow = 0;
71      aEffect = 0;
```

```
72    }
73    if (E == EDGE_EQUAL){
74      if (multiplicity < 0){
75        aRLow = -multiplicity;
76        aRHigh = -multiplicity;
77      }else{
78        aRLow = multiplicity;
79        aRHigh = multiplicity;
80      }
81      aRUsed = 0;
82      aEffect = 0;
83    }
84    if (E == EDGE_RESET){
85      aRLow = 0;
86      aRUsed = 0;
87      aRHigh = INFTY;
88      aEffect = 0;
89      aEffectSetter = true;
90    }
91
92    if (arcs.count(transition) && arcs[transition].count(place↩
          )){
93      #if DEBUG >= 9
94      std::cerr << "Combining arc " << transitions[transition]↩
            << "<->" << places[place] << ": ";
95      #endif
96      arcs[transition][place].combine(PetriArc(aRUsed, aRLow, ↩
          aRHigh, aEffect, aEffectSetter));
97    }else{
98      arcs[transition][place] = PetriArc(aRUsed, aRLow, aRHigh↩
          , aEffect, aEffectSetter);
99      #if DEBUG >= 10
100     std::cerr << "Inserting arc " << transitions[transition]↩
            << "<->" << places[place] << ": " << arcs[↩
          transition][place].label() << std::endl;
101     #endif
102   }
103 }
```

It works by looping over all the edges declared in the Snoopy XML file, parsing out the related place, transition and multiplicity as well as edge direction. Then this information is used together with the type of edge to come up with the range and effect values for the arc label.

Something particularly interesting happens when an arc between a given place and transition already exists. The combination operator is used to combine the arc labels of the existing arc and the new arc together:

```
1     arcs[transition][place].combine(PetriArc(aRUsed, aRLow, ↩
          aRHigh, aEffect, aEffectSetter));
```

Since the combination operator is used to combine arc labels together that connect the same place and transition before either the effect function or the range function is used on any arc, no information or functionality is lost if we do this ahead of time. It also results in a nice speed-up, as we no longer have to do this combining at run time.

The combination operator as given earlier in this section is translated almost literally to code:

```
1  /// \brief The combination operator.
2  ///
3  /// When called, this PetriArc and given PetriArc are ↩
       combined into this PetriArc (irreversibly).
4  void PetriArc::combine(PetriArc param){
5    //Set u to the sum of u1 and u2
6    rangeUsed += param.rangeUsed;
7    //Set l to the maximum of l1 and l2
8    if (param.rangeLow > rangeLow){rangeLow = param.rangeLow;}
9    //Set h to the minimum of h1 and h2
10   if (param.rangeHigh < rangeHigh){rangeHigh = param.↩
         rangeHigh;}
11   //Set a to the sum of a1 and a2
12   effectAdded += param.effectAdded;
13   //If either is a setter-arc, e is the new a. Otherwise, it↩
         is e1+e2.
14   if (!effectSetter && !param.effectSetter){
15     effect += param.effect;
16   }else{
17     effectSetter = true;
18     effect = effectAdded;
19   }
20 }
```

Similarly, the code contains nearly verbatim copies of the range and effect functions from Definition 8.

The range fuction $f_r((l,h),m) = \begin{Bmatrix} \texttt{true if } l \leq m \leq h \\ \texttt{false otherwise} \end{Bmatrix}$:

```
1  bool PetriArc::rangeFunction(unsigned long long m){
2    return (rangeLow <= m && m <= rangeHigh && rangeUsed <= m)↩
         ;
3  }
```

And the effect function $f_e(e,m) = e + m$:

```
1  void PetriArc::effectFunction(unsigned long long & m){
2    if (effectSetter){
3      m = effect;
4    }else{
```

```
5       m += effect;
6     }
7   }
```

To then do single steps on this in-memory `NatNet`, again a nearly verbatim copy of a definition from this paper is used. In this case, the steps from Algorithm 1:

```
1   if (stepMode == SINGLE_STEP){
2     //Every transition is checked for enabledness, and made ↩
          part of a subset consisting of only enabled ↩
          transitions.
3     for (T = arcs.begin(); T != arcs.end(); T++){
4       if (isEnabled(T->first)){enabled.insert(T->first);}
5     }
6
7     #if DEBUG >= 5
8     fprintf(stderr, "Single-stepping: %u transitions enabled↩
          \n", (unsigned int)enabled.size());
9     #endif
10    //Nothing enabled? We are done. Cancel running net.
11    if (enabled.size() == 0){return false;}
12    //pick a random enabled transition
13    selector = enabled.begin();
14    std::advance(selector, rand() % enabled.size());
15    #if DEBUG >= 4
16    fprintf(stderr, "Single-stepping: picked transition %s\n↩
          ", transitions[*selector].c_str());
17    #endif
18    //Run the effect function on each arc of the chosen ↩
          transition.
19    //We do not calculate the pt-combined arc label here, ↩
          since it has been pre-calculated during net load ↩
          already for each transition
20    std::map<unsigned long long, PetriArc> & selected = arcs↩
          [*selector];
21    for (A = selected.begin(); A != selected.end(); A++){
22      A->second.effectFunction(marking[A->first]);
23    }
24    //Step completed.
25    return true;
26  }
```

Maximal auto-concurrent stepping has also been implemented, following Algorithm 8:

```
1   if (stepMode == MAX_AUTOCON_STEP){
2     //Every transition is checked for enabledness, and made ↩
          part of a subset consisting of only enabled ↩
```

```
        transitions.
3       for (T = arcs.begin(); T != arcs.end(); T++){
4         if (isEnabled(T->first)){enabled.insert(T->first);}
5       }
6
7       #if DEBUG >= 5
8       fprintf(stderr, "Maximal auto-concurrent stepping: %u ↩
            transitions enabled\n", (unsigned int)enabled.size()↩
            );
9       #endif
10      //Nothing enabled? We are done. Cancel running net.
11      if (enabled.size() == 0){return false;}
12      //prepare empty list of chosen transitions and empty ↩
            PetriSuperTrans
13      std::map<unsigned long long, unsigned long long> ↩
            chosenTrans;
14      PetriSuperTrans super;
15
16      //pick a random enabled transition
17      selector = enabled.begin();
18      std::advance(selector, rand() % enabled.size());
19      chosenTrans[*selector]++;//increment chosen transition ↩
            counter
20      super.combine(arcs[*selector]);//combine the chosen ↩
            transition into the PetriSuperTrans
21
22      //keep going until no enabled transitions are left to ↩
            add
23      while (enabled.size()){
24        //pick a random enabled transition
25        selector = enabled.begin();
26        std::advance(selector, rand() % enabled.size());
27        //would super still be enabled if this transition was ↩
              added?
28        if (super.isCombinedEnabled(arcs[*selector], marking))↩
              {
29          //if so, add it
30          chosenTrans[*selector]++;//increment chosen ↩
                transition counter
31          super.combine(arcs[*selector]);//combine the chosen ↩
                transition into the PetriSuperTrans
32        }else{
33          //if not, remove it from the list of enabled ↩
                transitions
34          enabled.erase(selector);
35        }
36      }
37
38
39      #if DEBUG >= 4
```

```
40      std::cerr << "Maximal auto-concurrent stepping: picked ↩
            transitions:";
41      std::map<unsigned long long, unsigned long long>::↩
            iterator pckd;
42      for (pckd = chosenTrans.begin(); pckd != chosenTrans.end↩
            (); pckd++){
43        std::cerr << " " << transitions[pckd->first];
44        if (pckd->second > 1){
45          std::cerr << " (X" << pckd->second << ")";
46        }
47      }
48      std::cerr << std::endl;
49      #endif
50      //Run the effect function on each arc of super.
51      std::map<unsigned long long, PetriArc> & selected = ↩
            super.myArcs;
52      for (A = selected.begin(); A != selected.end(); A++){
53        A->second.effectFunction(marking[A->first]);
54      }
55      //Step completed.
56      return true;
57    }
```

For completeness, here is the `isEnabled` function as used above as well. It's a near verbatim copy of Definition 5:

```
1  bool PetriNet::isEnabled(unsigned int T){
2    //We consider transitions without arcs to not be enabled, ↩
         since that is the only thing that makes sense.
3    if (!arcs[T].size()){
4      return false;
5    }
6
7    // Loop over all p in P such that p connected to t
8    std::map<unsigned long long, PetriArc>::iterator A;
9    for (A = arcs[T].begin(); A != arcs[T].end(); A++){
10     //Check fR(aR, M(p)), if false, return false
11     //We do not calculate the pt-combined arc label here, ↩
           since it has been pre-calculated during net load ↩
           already for each transition
12     if (!A->second.rangeFunction(marking[A->first])){return ↩
           false;}
13   }
14
15   //only if all fR(aR , M (p)) are true, return true
16   return true;
17 }
```

9 Discussion and conclusion

The straightforwardness of the implementation as shown in Section 8 shows that `NatNet`s are a good notation to use for Petri net simulations. During the writing and testing of the implementation, it became clear that the combination operator of `NatNet`s can also be used to simplify and classify collections of arcs into more generic behaviours, which could assist the interpretation of (larger or more complex) Petri nets by human eyes.

The extension to make `NatNet`s compatible with reset arcs (and, as a result, set arcs in general) only required minor changes, showing the extensibility of the method as well as producing the tool `PetriCalc`. `PetriCalc` was used in [BKH+15] to do calculations on nets with literally millions of tokens moving around, as a means to experimentally verify theoretical models. Existing tools at the time required days to weeks to complete those calculations only partially, where `PetriCalc` needed only hours to reach end results. This not only shows that `PetriCalc` (and the theory it is based on) has practical uses but also that it is significantly more efficient than existing tools that do Petri net simulation / animation.

However, [BKH+15] did not use `PetriCalc` in its current iteration. Both the theory from this paper as well as the practical implementation in the form of `PetriCalc` went through iterative changes over a period of several years. The calculations done using the earlier iteration were validated against the latest version of the theory and tool, and the results were identical. This is very good, because the earlier version of `PetriCalc` wasn't fully compatible with all extended nets as produced by the Snoopy tool and this latest version is. So, for nets that were compatible with the older version, the results were indeed expected to be identical.

Further work could still be done, both on the theory and `PetriCalc`:

It is not clear why `PetriCalc` is so much more efficient than existing tools, as it is hard to know what calculation methods those tools use and thus it is a complicated task to compare these. This could be an area of further research.

Supporting coloured Petri nets would be a sensible next step, but creating `PetriClass`es for timed Petri nets, stochastic Petri nets and continuous/hybrid Petri nets would also be very interesting as they all have their own particular semantics and challenges. The most readily visible class of net that could be made to fit a `PetriClass` is perhaps coloured Petri nets. In stead of `NatNet`s scalars, here markings, ranges and effects could be represented by matrices of scalars, for example.

Besides simply supporting more classes of nets, the algorithms could also be improved upon. A great speed-up is possible when it is known in advance that a net is deterministic, for example (as there is no need to check further transitions after finding the first enabled transition). More speed still could be gained in auto-concurrent maximal stepping mode, as a single transition can instantly be added as many times as it possibly could be without needing to worry about possible conflicts with other transitions. First experiments with

this method have shown an increase in simulation speed that is incrementally higher as token counts go up.

`PetriCalc` only has implementations for single steps and for maximal auto-concurrent steps, implementing (maximal) concurrent steps as well as non-maximal auto-concurrent steps are also good candidates for further work. The algorithms for these have already been outlined in this paper, but a practical implementation is still lacking.

A visualizer that simplifies a net to no more than one arc per place/transition combination and then highlights them depending on their arc label type could also be a very interesting analysis tool based on the work in this paper. The various arc types would make the effects of firing arcs immediately visible and thus much easier to analyse by human eye.

Finally, Snoopy is by far not the only Petri net tool out there, and `PetriCalc` could benefit from being able to read nets created by other tools as well.

References

[BKH+15] Laura M. F. Bertens, Jetty Kleijn, Sander C. Hille, Monika Heiner, Maciej Koutny, and Fons J. Verbeek. Modeling biological gradient formation: combining partial differential equations and petri nets. *Natural Computing*, pages 1–11, 2015.

[BM97] Roberto Bruni and Ugo Montanari. Zero-safe nets, or transition synchronization made simple. *Electr. Notes Theor. Comput. Sci.*, 7:55–74, 1997.

[BM00a] Roberto Bruni and Ugo Montanari. Executing transactions in zero-safe nets. In *ICATPN*, pages 83–102, 2000.

[BM00b] Roberto Bruni and Ugo Montanari. Zero-safe nets: Comparing the collective and individual token approaches. *Inf. Comput.*, 156(1-2):46–89, 2000.

[BM01] Roberto Bruni and Ugo Montanari. Transactions and zero-safe nets. In Hartmut Ehrig, Gabriel Juhás, Julia Padberg, and Grzegorz Rozenberg, editors, *Unifying Petri Nets, Advances in Petri Nets*, volume 2128 of *Lecture Notes in Computer Science*, pages 380–426. Springer, 2001.

[Bur83] Hans-Dieter Burhard. On priorities of parallelism: Petri nets under the maximum firing strategy. In A. Salwicki, editor, *Logics of Programs and Their Applications*, volume 148 of *Lecture Notes in Computer Science*, pages 86–97. Springer Berlin Heidelberg, 1983.

[DKPY08] Philippe Darondeau, Maciej Koutny, Marta Pietkiewicz-Koutny, and Alexandre Yakovlev. Synthesis of nets with step firing policies. In Kees M. van Hee and Rüdiger Valk, editors, *Applications and Theory of Petri Nets, 29th International Conference, PETRI NETS 2008, Xi'an, China, June 23-27, 2008. Proceedings*, volume 5062 of *Lecture Notes in Computer Science*, pages 112–131. Springer, 2008.

[DMM96] Pierpaolo Degano, José Meseguer, and Ugo Montanari. Axiomatizing the algebra of net computations and processes. *Acta Inf.*, 33(7):641–667, 1996.

[DR98] Jörg Desel and Wolfgang Reisig. Place/transition Petri Nets. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*, pages 122–173. Springer, Berlin / Heidelberg, June 1998.

[HHL+12] Monika Heiner, Mostafa Herajy, Fei Liu, Christian Rohr, and Martin Schwarick. Snoopy a unifying petri net tool. In Serge Haddad

and Lucia Pomello, editors, *Application and Theory of Petri Nets*, volume 7347 of *Lecture Notes in Computer Science*, pages 398–407. Springer Berlin Heidelberg, 2012.

[HMR05] Thomas A. Henzinger, Rupak Majumdar, and Jean-François Raskin. A classification of symbolic transition systems. *ACM Trans. Comput. Log.*, 6(1):1–32, 2005.

[KK10] Jetty Kleijn and Maciej Koutny. Petri nets with localities and testing. In Johan Lilius and Wojciech Penczek, editors, *Applications and Theory of Petri Nets, 31st International Conference, PETRI NETS 2010, Braga, Portugal, June 21-25, 2010. Proceedings*, volume 6128 of *Lecture Notes in Computer Science*, pages 19–38. Springer, 2010.

[KKPKR12] Jetty Kleijn, Maciej Koutny, Marta Pietkiewicz-Koutny, and Grzegorz Rozenberg. Step semantics of boolean nets. *Acta Informatica*, 50(1):15–39, 2012.

[MM90] Jos Meseguer and Ugo Montanari. Petri nets are monoids. *Information and Computation*, 88(2):105 – 155, 1990.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.

[Pet77] James L. Peterson. Petri nets. *ACM Computing Surveys*, 9:223–252, 1977.