Master thesis

Rasmus Bartholin

May 2023

Contents

1 Abstract

2 Introduction

3 Petri nets

Petri nets are a mathematical tool used for modelling many different types of problems. It has been used to analyze many different problems, such as concurrency problems, communication protocols and chemical pathways [11]. A Petri net can be viewed as a graph, with two distinct node types, transitions and places. The definition of a Petri net is:

Definition 1 Petri net: [11]. A Petri net is a 5-tuple, $PN = (P, T, F, W, M_0)$ where:

$P = \{p_1, p_2, ..., p_m\}$ is a finite set of places

$T = \{t_1, t_2, ..., t_n\}$ is a finite set of transitions

$F \in (P \times T) \cup (T \times P)$ is a set of arcs (flow relation)

$W{:}F \Rightarrow \{1, 2, 3, ...\}$ is a weight function

$M_0{:}P \Rightarrow \{0, 1, 2, 3, ...\}$ is an initial marking

$P \cap T = \emptyset$ and $P \cup T \neq \emptyset$

A Petri net structure $N = (P, T, F, W)$ without any specific initial marking is denoted N and is called a net.

A Petri net with the given initial marking is denoted by $PN = (N, M_0)$.

As the definition states, a Petri net consists of a finite set of places and transitions. The arcs between these node types can have weights ranging from 0 to $+\infty$. The places and transitions have different attributes. A place can contain 0 to $+\infty$ tokens. A token is some sort of abstract resource, that can be moved around the Petri net. This is done by firing transitions. Before we can talk about firing transition, we need a few more definitions:

Definition 2 *Postset and preset definitions [4]:*

For $x \in P \cup T$

$\cdot x{:} = \{y \mid yFx\}$ is called a preset of x

$x \cdot{:} = \{y \mid xFy\}$ is called a postset of x

In plain words, a preset of a transition or place, is the nodes that have arcs going into that place or transition. The postset is the reverse of a preset. It is the nodes that a place or transition have outgoing arcs to. The places and transition have a specific flow relation. A place can only have outgoing and ingoing arcs from transitions. The reverse is true for transitions. They can only have arcs to and from places. An example of a valid and non-valid Petri net, can be seen in Figure 1. Figure 1a depicts how Petri nets are usually drawn. The circles are places, and the boxes are transitions. The small dots inside the places are the tokens of that place. The numbers above the edges are the weights. If an

edge has a weight of 1, then that edge's weight is often not drawn. This will also be the case in this thesis. Now we are ready to define, what happens when a transition is *fired*.

(a) Valid Petri net (b) Non-valid Petri net

Figure 1: Example of a valid Petri net and one that is not valid. The Petri net that is not valid, has two places with outgoing arcs, connecting to another place. This is not allowed.

When a transition is fired, tokens from the transition's preset are removed, and tokens are added to its postset. How many tokens are taken and put in each place, is determined by the weights of the arcs. For the places in the transition's preset, the weight of the arc (p, t) determines how many tokens are removed from p. This action is often referred as the token from p has been *consumed*. For the places in the transition postset, the weight of the arcs from t to all places p in the postset determines how many tokens is put in those places. This is the reverse of consuming tokens from a place, and is often called *producing* tokens in place p [11]. This means, that the number of tokens that is consumed in a transition's preset, does not have to match the number of tokens produced in its postset when that transition is fired. Whether a transition can be fired or not depends on the number of tokens in each place in the preset of the transition, and the weight of the arcs going from those places to that transition. If every place in a transition preset has as many or more tokens than the weight of its arc to that transition, the transition can be fired and is said to be *enabled*. If a transition can not be fired, it is said to be *disabled* [11].

(a) (b)

Figure 2: A small example of firing a transition in a Petri net. The arcs (O_2, t) and (t, H_2O) does not have a depicted weight on them, because their weight is 1. In (a) the transition is enabled. (b) depicts the same Petri net as (a), but after t is fired. In (b), t is no longer enabled [11].

Thus, when a transition is fired, the Petri net changes state, as we can see in Figure 2. Tokens are moved or disappear, and transitions can both be enabled or disabled. There is an important distinction in the definition of the Petri net, that was omitted until now. It is not one single entity, but consists of two objects. A *Net* denoted as N and a *marking* denoted as M. If we combine them, we get a Petri net $PN = (N, M_0)$. Here M_0 is the *initial marking*. A marking is something we place on top of a net and is the placement of the tokens. Note, that since the net N does not have a marking, it is not possible to talk about if a transition is enabled or not, since the notion of tokens is not yet introduced. This distinction becomes important later. A Petri net is thus comprised of two components. The structure of it in form of the net N, and its marking or *state* M. As an example, In Figure 2a the Petri net is in the *initial marking*, denoted as M_0. Then t is fired, and we reach a new marking we can call M_1. Figure 3 is the Net N of 2a and 2b. A special case of a marking that will become important later is the *null marking*. This is the marking, where all places have 0 tokens[11].

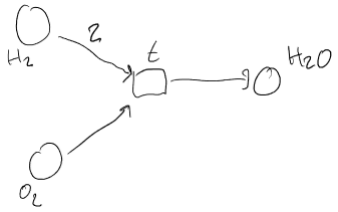


Figure 3: The net N of 2a and 2b without the marking. The null marking would be depicted in the same way, since no tokens would be at any of the places.

From this, we can move on to the concept of a *reachabilty graph*. Since they will be at the core of this thesis, this subject will be covered in detail.

3.1 Reachabilty graphs

Figure 2a depicts a Petri net in an initial marking. From this initial marking, we can fire t and reach another marking, depicted in 2b. Since drawing the entire Petri net for every firing is quite space inefficient, and is also hard to read if the Petri net is large, this process of going from one marking to another possible marking is often presented as a graph called a *reachability graph*. In a reachability graph, the nodes corresponds to the different markings and the directed arcs equates to a specific transition firing. The nodes can be depicted in a few different ways. One method is that each node is an array of numbers, where each entry corresponds to a specific place. The numbers on each entry, is the number of tokens in that place. This is the method used in Figure 4. A reachability graph can thus be used, to get an overview of all reachable marking from some initial marking M_0 [11].

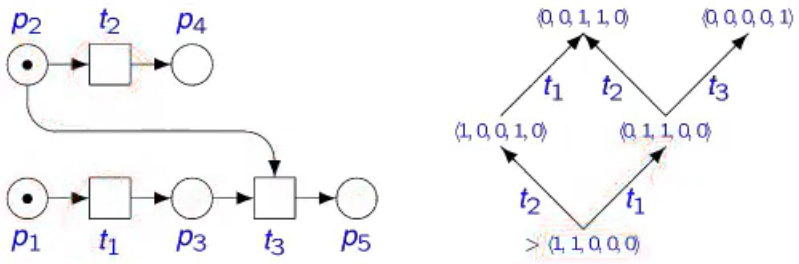


Figure 4: A Petri net and the corresponding reachability graph. The node in the bottom is the initial marking M_0. The order of the places in the nodes array are: p_1,p_2,p_3,p_4,p_5.

One important thing to note, is that a reachability graph can only be created, if the Petri net it is created from is *bounded* also called *k-bounded*. A Petri net is bounded, if it is impossible, from the initial marking M_0, to put more than k tokens in any place p in the Petri net, where k is some positive integer number. No matter what transitions is fired, or in what order they are fired in. Conversely, a Petri net is *unbounded* or *k-unbounded* if there exists a place p, where one can put an infinite number of tokens. If this is the case, then there cannot be created a reachability graph for that Petri net. Then a *coverability graph* can be produced instead [11]. The reason that a reachability graph cannot be produced if a place is unbounded is simple. Since there needs to be a node for each reachable marking, a Petri net with a place where it is possible to pump an infinite number of tokens into a place p, would obviously result in an infinitely large reachability graph which cannot be computed [11]. The smallest possible example of this, can be seen in Figure 5.

Figure 5: A Petri net, that would result in an unbounded reachability graph. Since there t has no preset, it will always be enabled. We can thus always fire it, and create a new marking with one more token, then the previous one.

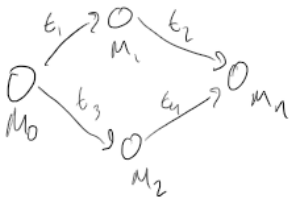
Now that we have defined what a reachability graph is, we will explore what aspects of the reachability graph we are interested in for this thesis.

3.2 Firing sequences and paths

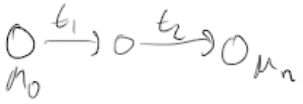
With a reachability graph created from a given Petri net, one can answer a lot of different questions about the Petri net, such as *liveness* and *reversibility*, which will not be explored in this project. The question that is relevant for this thesis, is if it is possible to reach a certain marking from M_0. We will call this marking the *end marking*, *goal marking* or the *target marking* in this thesis, and is denoted as M_n. This problem is known as the *reachability problem* [11]. This is written as $M_0[> \sigma M_n$. In this expression σ signifies a *firing sequence*. A firing sequence is as the name suggest a sequence of firings that can be performed from an initial marking to a target marking. What a firing sequence σ contains can vary depending on who you ask [11]. In this thesis, it will be a series a transitions $[t_1, t_2, t_3, ..., t_n]$, that can be fired in the order of the sequence, to go from M_0 to M_n. Just for clarification, LoLA which we will explore more later, uses the term *reachability query* for the reachability problem, while σ is called a *witness* of the reachability query [12].

An important distinction needs to be made between a *path* in the reachability graph and a firing sequences. A path is a monomorphism of the reachability graph, and can be denoted as $M_0, t_1, M_1, t_2 ... t_n, M_n$ [11]. In other words, a path will be a subgraph of the reachability graph, where a firing sequence is just the edges on that path. An example of the differences between a path and a firing sequence is depicted in Figure 6.

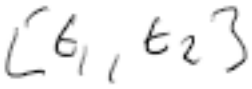
Not defined yet. Also seems a very big word here? Isn't subgraph fine (as used in the



(a)



(b)



(c)

Figure 6: An example of the difference between a path of a reachability graph, and a firing sequence σ. (a) is a reachability graph, where the nodes are markings. The number of tokens on each of them are hidden, since it is not relevant for this example. (b) is an example of a path from M_0 to M_n. It contains both the markings and the transitions. The corresponding σ can be seen in (c).

The two have a bijective relation to each other. However, each individual firing sequence can be saved on less space since it only needs to store the transitions, while a path needs to store the transitions and nodes that represent the different markings. In Section 8.5 we will discuss the advantages and disadvantages of storing either one while exploring the reachability graph. Just for completeness the strategy to go from a firing sequences to a path and vice versa is as follows.

If we have a firing sequences fire the transition from first to last using the initial marking as a starting point. After each transition is fired, record the marking that has just been reached. Now simply build the path. Retrieving the firing sequences from a path is even simpler. Start from the node that represents the initial marking, and for each edge encountered on the path, put the transition stored there in the back of a list. Continue this process until the end of the path has been reached.

Now that we have the definition of a path in a reachability graph and a firing sequences, we can finally discuss how Petri net will be used in detail in this thesis, and what will be a new entry to what can be done with Petri nets.

3.3 How Petri net can be used to model Chemical Reaction Networks

As discussed in the introduction, we want to use Petri nets to model chemical reaction networks, and find pathways between them. These networks can be expressed as Petri nets. How this transformation is done, won't be explored in this project. In practice, these will be produced by MØD [2].

Maybe a little sad that we get zero examples of how a chemical network can be expressed as a petri net (which is also in contradiction to that fact that you already tried to use molecular names for nodes!) Make a figure with an example (chemical network and corresponding petri net)?

Assuming we have such a Petri net that models a chemical reaction network, what question chemist we like answered, is if it is possible to go from an initial marking M_0 to a target marking M_n. Furthermore, they would like to know how they can reach M_n. As we have already discussed, there already exist programs that can do this such as LoLA [13]. What has not yet been created, is a program that can retrieve *all* possible firing sequences from M_0 too M_n. Why are we interested in this feature? Even though that it has been possible for a long time, to get chemical pathways from a chemical reaction network, there are a lot of questions about these pathways that are not known. Are there more than 1? If there are, how do they differ? Are some of them "better" in a chemical sense than others, or are they all equivalent? If there can be more than 1, are there just a few typically or are there many for a most chemical reaction networks? If it is possible for networks to have more than 1, are networks with only 1 path special in some way? This thesis will not answer these questions. I do not have the required background for it. But it will explore how such a tool to produce all of these paths can be designed and implement it, so that others will be able to.

Wonderful text for the introduction part. Then not needed to be repeated here in full detail.

However, before we go into how such a tool can be designed, we need some more background. Getting all firing sequences from M_0 to M_n is not all that the chemist will need. In Section 4 we will cover the last parts we need.

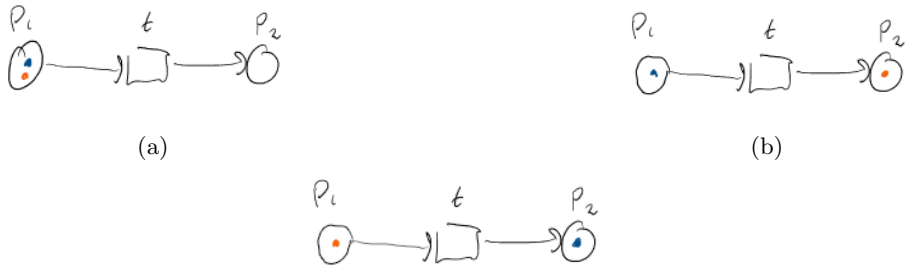
4 Individual versus Collective Token Interpretation and Occurrence Nets

As just stated, chemist will need a bit more, before they will have a solution they can use for their analysis. A Petri net as defined in Section 3 can be used to model chemical reaction. We saw a small example of this in Figure 2. However, they do not quite capture the whole story of a firing, because of the *collective token interpretation*. To explain this term, we need to look at what happens when a transition is fired, as a Petri net was defined in Section 3.

When a transition is fired, of the transitions preset are consumed, and tokens in the postset are produced. But as these definitions indicate, it is not the same tokens in the preset and postset. When we say, that a token is consumed at a place, we literally mean, that they disappear from that place. When tokens are produced for a place, they are created for that place, and have no connection to the tokens that were removed from the preset. So for this model, there is no way to discuss where a specific token ended up after or during a firing sequence [9]. However, this can be important information for a chemist. What is needed, is an *individual token interpretation*.

In an individual token interpretation, each token is unique. When a transition is fired, a token is not consumed, but is moved from one place to another, though the arcs of the transition that was fired. This makes it possible to track a specific token, as transitions are fired in a firing sequence[9]. An example of this can be seen in Figure 7. 7a depicts the initial marking of that Petri net.

The colors of the tokens, signifies that the two tokens are unique. Both 7b and 7c represents the two possible outcomes when firing t once. For contrast, Figure 8 depicts the same Petri net, before and after firing t. Here, it is not possible to know which token now is in p_2.



(a) (b)

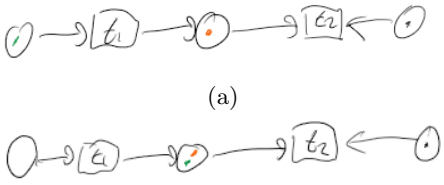
(c)

Figure 7: Individual token interpretation of a Petri net. (b) and (c) are two different outcomes that are possible from (a), when firing t once. The individuality of the nodes are represented by coloring the tokens.

(a) (b)

Figure 8: Collective token interpretation of the same Petri net as Figure 7. Here there is only one interpretation after firing t, as we can see in (b).

This can make a huge difference, when talking about the casualty of a firing sequence. When the tokens are individualized, we can establish a casual link between the firings of a σ. This happens when a transition is using a token, that was moved to its preset by a previous firing. Ron van Glabbek has a brilliant little example, that I will borrow [9]. The figures can be seen in Figure 9.



(a)

(b)

Figure 9: (a) is the initial marking of the Petri net. (b) is the Petri nets state after the transition t_1 was fired.

In Figure 9b we can see the Petri net after the transition t_1 was fired. There are now two options, when we are at the state depicted in 9b. If the orange token is used to fire the transition t_2, then the firing of t_2 is independent of the firing of t_1. If the green token is used, then the firing of $t2$ is dependent on the firing of t_1. If we used the collective token interpretation, then this firing of t_2 would always be independent of the firing of t_1. This means, in the individualized token interpretation, we can establish a partial order of the firing of the transitions. We can in a meaningful way describe in all cases if a firing of a transition dependent on another transition [9]. This is often not the case for the collective token interpretation, as Figure 9b is an example of.

We have now defined, what an individualized token interpretation is. We have yet to discuss how or if we are going to change our definition of a Petri net, in order to accommodate it. This will be discussed in the next section

4.1 Occurence net

It is possible to redefine the firing rule from Section 3, so that it has an individual token interpretation [9]. This is not the approach that was used in this project.

Instead, we will derive the individualized token interpretation by creating what is called an *occurrence net*. The definition of an occurrence net, is sadly also dependent on which source you use to define it. However, they all reflect the same idea. They are an altered version of the definition of a net, that we saw in definition 1. What makes them interesting for us, is that given an initial marking M_0 and a firing sequence σ we can create an occurrence net, that reflects what happened with the Petri net, as the transitions in σ were fired. What we need from it, is that we can use it to tell where each individual token was, after each firing. We will first present the definition that was used in this thesis, and then show a few examples of how one can draw an occurrence net from an M_0 and σ. However, before we can define what an occurrence net is, we need to revisit the definition of net.

Bedre sammenhæng med det foregående (mindre "modstrid" i formulering).

The definition of a net, is something we saw in Section 3. As a reminder, a Net is the structure of the Petri net. It is the Petri net, without the marking. An occurrence net is a net, but with some further restrictions on it:

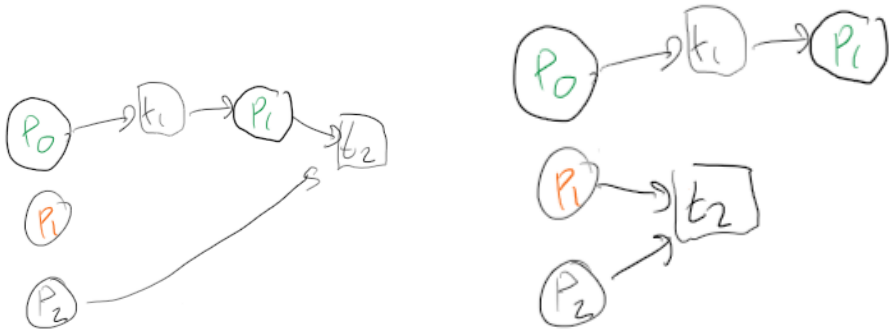
Definition 3 *Occurrence net definition: [4]. A net K is an occurrence net iff*

Aha, the relation view may after all have its uses(?)

(a) $\forall x, y \in K\ xF_K^+y \Rightarrow \neg(yF_K^+x)$ F_K^+ denoting the transitive closure of F_k.

(b) $\forall p \in P_K |^{\cdot}p| \leq 1 \wedge |p^{\cdot}| \leq 1$.

Let us examine the definition. The first restriction (a) simply states, that there shall be no cycles in an occurrence net. If there is a flow relation from x to y, then there is no flow relation to y to x. The second restriction (b) states that no arc going from or to a place can have a weight larger than 1. Another way to say this, is that no arcs have a weight larger than 1.



(a) (b)

Figure 10: The two possible occurrence nets that can be built from the Petri net in Figure 9a after firing t_1 and t_2. The colors of the places indicates, what tokens they were built from. The names of the O-places indicates which place the token was taken from, when firing the transition it is has an outgoing arc to.

An example of an occurrence net can be seen in Figure 10. Exactly how one would build an occurrence net from the Petri net (N, M_0) from Figure 7a and the firings sequence $[t_1, t_2]$ will be explored later. For now, we just need to understand what an occurrence net is comprised off. Since an occurrence net is a net, it still has both places and transitions. These are drawn in the same manner as in a Petri net, with circles indicating that the node is a place, and a square indication that the node is a transition. The arcs also have to comply to the same rules as a net. Places cannot have an arc's to and from another place, and a transition cannot have arcs going to or from another transition. Since an occurrence net does not have a marking, the places do not contain tokens, and is therefor never drawn with them either [4]. To avoid confusion going forward, when places and transitions are discussed that are a part of an occurrence net, they will be called *O-places* and *O-transitions*.

The colors of the O-places, indicates which token they are tied to. The names indicate what place the O-place was build from. Coloring the places is not normally done, and will not be done throughout the thesis. This is just to

make the first few examples easier to understand. Each occurrence net that can be built from a Petri net $PN = (N, M_0)$ and a firing sequence σ corresponds to a choice, to what token to use for each firing. In Figure 10a the green token was used for the firing of t_2, while in Figure 10b, the orange token was used to fire t_2. We can thus use the different occurrence nets that can be generated from a (N, M_0) and a σ, to analyze how the individualized tokens can move though the Petri net. Note that given a (N, M_0) and a σ, one might get multiple "unique" and/or "different" occurrence nets. How occurrence net are defined to be either equal to or different to one another, will be discussed in Section 4.3. To go forward, we need a proper definition of how the Petri net, firing sequences and occurrence nets relates to each other. This will be done in the next section.

4.2 Process

Before we can define how a Petri net with an σ is tied to an occurrence net, we need some definitions. The first relates to how we can read the different markings, that can be reached from (n, M_0) and firing the transitions of σ in the occurrence net itself. Here we can use what Goltz and Reisig calls a *slice* of an occurrence net 3:

Definition 4 *Let K be an occurrence net.*

(a) $<_K \; := F_K^+$ is the *order relation of K*. The index K is omitted if it is obvious from the context.

(b) Let $\mathbf{li} \subseteq K \times K$ and $\mathbf{co} \subseteq K \times K$ be given by

$$x\mathbf{li}y :\Leftrightarrow x < y \vee y < x \vee x = y,$$

$$x\mathbf{co}y :\Leftrightarrow \neg(x\mathbf{li}y) \vee x = y.$$

(c) $M \subseteq K$ is a *cut* iff $\forall x, y \in M \; x\mathbf{co}y \wedge \forall z \in K \setminus M \; \exists x \in M \; \neg(x\mathbf{co}z)$.

(d) A cut $M \subseteq K$ is a *slice* iff $M \subseteq P_K$.

Definition 5 defines the order of the nodes in an occurrence net. Given two nodes x and y, they can either be ordered or the same element (denoted with $x\mathbf{li}y$), or we cannot compare the order of x and y or they are the same element (denoted with $x\mathbf{co}y$) [4]. A cut, is a subset of the occurrence net nodes, where order is not defined. A slice, which we are particularly interested in, is a cut that consists only of O-places. Figure 13 has some examples of slices of an occurrence net, and the difference between a cut and a slice.

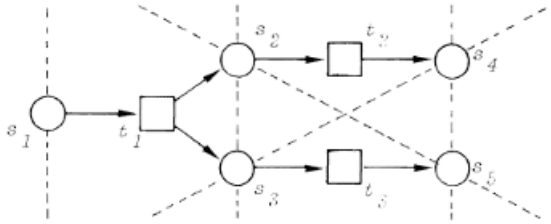
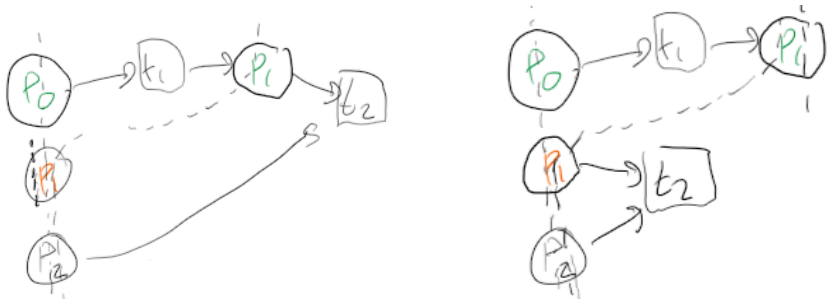


Figure 11: Example of slices from [4] The dotted lines are valid slices. $[p_2, t3]$ is a cut but not a slice, because of the presence of t_3.

Now we can finally discuss how one can find all the markings visited from firing the transition of σ from M_0 by examining the occurrence net. Let us revisit the occurrence nets of Figure 10, but with some more detail that can be seen in Figure 12.



(a) (b)

Figure 12: The two possible occurrence nets that can be built from the Petri net in Figure 9a after firing t_1 and t_2. The dotted lines are the slices of the two occurrence nets.

The slices of the occurrence nets in Figure 12a directly corresponds to the different markings that were reached from M_0 while firing $[t_1, t_2]$. There are in total 3 slices for both occurrence nets since there were 3 markings visited, M_0, the marking after firing t_1 and the marking after firing t_2. M_0 consists of the leftmost p_0, p_1 and p_2 in both 12a and 12b. After firing t_1, we will reach the marking M_1 that corresponds to the slice p_1, p_1, and p_2. The last marking M_n consists of the p_1 that was not used. This is orange p_1 for the occurrence net in Figure 12a, and the green p_1 in Figure 12b. Note that a special case is the null marking. Here, the slice of any occurrence net would be empty.

The astute reader may have noticed, that it is not possible to always find only the markings that were reached from M_0 by firing σ in order, unless σ is also provided. The initial marking, can always be created from the slice of the O-places that has no ingoing arcs. However, finding the next markings in order is not always possible to determine by the occurrence net alone. The occurrence

net of Figure 12b is an example of this. If we do not have σ, whether t_1 or t_2 was fired first is not known to us, since the firing of t_1 and t_2 are independent in the occurrence net. It is possible that t_2 was fired first, which would result in a slice that consists of p_0 alone. But this marking was never reached from M_0 using $\sigma = [t_1, t_2]$. Thus, in general we need the σ as well as the occurrence net, to find the markings that were reached from the Petri net and the firing sequences. Note that we do not have this problem in Figure 12a since the firing of t_2 is dependent on the firing of t_1 in this case. If σ is provided, getting the correct slices is trivial. Since we will not need to do this (more on this later), the algorithm to find the correct slices from (N, M_0) and σ has been omitted.

That it is not possible to find the markings that the Petri net went through as the transitions of σ was fired, signifies that it is not always possible to find the σ that create the occurrence net either. The reason for this is simple. For a given Petri net and an occurrence net, several σ could have been used to create that occurrence net. Again we can use Figure 12b as an example. Both the firing sequences $[t_1, t_2]$ and $[t_2, t_1]$ could have created that occurrence net.

Now we are ready for the formal definition, of how a piticular Petri net can be mapped to an occurrence net.

Definition 5 Process [4] (adapted). Let $PN = (P_N, T_N, F_N, W_N, M_0)$ be a Petri net, and let M_0 be the initial marking of PN. A *process* is a triplet (K, q, M_0) of an occurrence net $K = (P_K, T_K, F_K)$ and a mapping $q{:}K \to PN$ that satisfies the following properties.

(a) $q(P_K) \subseteq P_N$ and $q(T_K) \subseteq T_N$

(b) C is a slice and $\forall p \in P_N \; M_0(p) = |q^{-1}(p) \cap C|$

(c) $\forall t \in T_K$ and $\forall p \in P_K$

$$W_N(p, q(t)) = |q^{-1}(p) \cap \circ t|$$

$$W_N(q(t), p) = |q^{-1}(p) \cap t \circ |$$

Definition 5 describes the relationship between a Petri net and an occurrence net, that will be necessary for the occurrence net to be built from the Petri net. A *process* is a triplet (K, q, M_0), where K is the occurrence net itself, while q is a mapping function and M_0 is the initial marking of the Petri net. This function q maps K back to the Petri net. For this mapping to be valid a number of requirements must be met. Firstly, the mapping and occurrence net needs to respect the transitions and places of the Petri net. There cannot be places or transitions, that cannot be mapped back to the Petri net. If the Petri net for example, had the places (p_1, p_2, p_3), we cannot have a O-place in the occurrence net build from p_4. Second, when we create a slice of the Petri net, for each place in the Petri net, there can be no more O-places than there were tokens in that place. This means that $q(p)$ maps many O-places back to a single place,

while $q^{-1}(p)$ maps a single place to one or more O-places. How many O-places, that should exist for a specific place, depends on the number of tokens in that place. Third, the weights of the edges of the Petri net needs to be respected. The weight of an edge from a place to a transition, needs to match with a set of O-places, that is connected to an O-transition, that is mapped to a transition of the Petri net. The same requirement is of course in place for edges from transitions to places.

Figure 13: Example of requirement (c) of definition 5 from [4]. The figure on the left is the Petri net, and the figure on the right is the occurrence net, that can be build from it.

The figures occurrence nets of the figures 12a and 12b has a valid mapping q with the Petri net of Figure 9a. Note that the firing sequence, is **not** a part of a process. As we have already seen with the occurrence net of Figure 12b, several σ can have been used to create this occurrence net. This means, that the definition of a process is closely related, but not quite what we need. It tells us, if a Petri net and an occurrence net has a valid mapping or not. If the occurrence net is tied to the Petri net. However, it does not tell us how we create these occurrence nets from a Petri net. What we need is to be able to build a valid occurrence net given a (N, M_0) and a σ, that respects the requirements of a process. Since this algorithm does not exist, we need to create it ourselves. This will be done in Section 10. For now, there is another question that has been left unanswered, that does have established theory we can use.

As we mentioned earlier, for a (N, M_0) and a σ, it is possible to produce many occurrence nets, that are either identical or unique. But we have not yet defined, what this means mathematically. This will be done in the next section. Then, we will start to build our solution.

4.3 Occurrence nets and isomorphism

As vaguely discussed in Section 4.1 the occurrence nets that can be produced by a firing sequence and a Petri net, can either be "identical" or "unique". We have a few good reasons to be able to tell, whether an occurrence net is "equal" to another. Since these occurrence nets will be implemented as a type of graph (more on this later), each can take up a significant amount of space. Therefor, we do not want to store several identical occurrence net. There are also runtime savings that can be achieved, by not storing them all. This will also be discussed in more detail later. Lastly, we do not want to waste the user's time, by making

them look through a bunch of solutions, that are identical. With this motivation in place, we can define what an *isomorphism* is.

Isomorphism has it origin from group theory. As everything in group theory, it can be applied to a lot of settings. In general, if two objects are isomorphic, they are the equal to one another. Here we have to distinguish between if they are *representationally equal* or isomorphic. If two object a and b are representationally equal denoted $a \overset{r}{=} b$, they have the same structure, but are not the same mathematical object. If they are isomorphic, denoted with, $a \cong b$ then they are the same mathematical object [1].

An example could be the mathematical object or group of all real numbers $\mathbb{R}$. Here we can denote the same mathematical object $\frac{1}{2}$ in many ways such as, $\frac{2}{4}$, $\frac{3}{6}$ and $\frac{21}{42}$. Each of these objects are the isomorphic to each other, $\frac{1}{2} \cong \frac{2}{4}$ but they are not representationally equal $\frac{1}{2} \overset{r}{\neq} \frac{2}{4}$. One could then try to figure out, fractions are isomorphic to one another, by finding the greatest common divisor.

In the graphs, this question is more complicated. Here we can also distinguish between the representation of a graph, and the mathematical object. There are several ways a graph can be represented, such as adjacency list or adjacency matrix [5]. Here, the nodes are usally given ID's in the form of integers from 0 to n. However, these are **not** a part of the mathematical object. The ID's are simply attached to the nodes, so that we can reference the distinct nodes. You could take the exact same graph and label it completely differently, and it would still be the same graph. The example in Figure 14 is taken from [1], and exemplifies this difference. In this figure, all the graphs are isomorphic to one another. It is only their representation that differ. This is an important thing to note. The labels themselves are not a factor, when we try to compare 2 graphs for isomorphism. This form of isomorphism, is called *graph isomorphism without labels*[1].

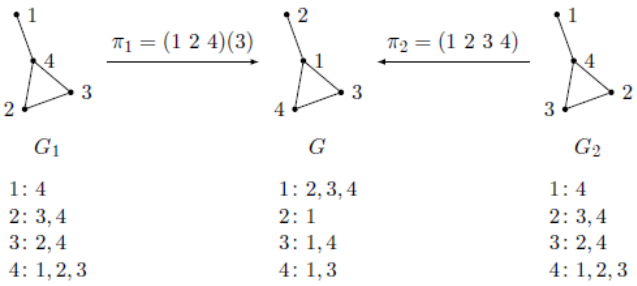


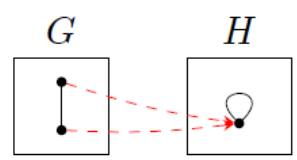
Figure 14: This example is taken from [1]. The graphs are represented as adjacency lists, and $G_1 \overset{r}{=} G_2$ while G is not representationally to the other two. All these graphs are isomorphic to each other.

To move forward, we need a formal definition of what it means for two graph to share the same structure. We can use the following definitions [1]:

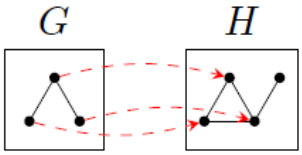
Definition 6 (Graph Morphisms). A graph (homo)morphism $m{:}G \to H$ is a structure-preserving mapping of vertices and edges. That is if $e = (u, v) \in E_G$ then $m(e) = (m(u), m(v)) \in E_H$.

- m is a *monomorphism* if it is injective: $\forall u, v \neq\!\!\to m(u) \neq m(v)$. When a monomorphism exists, we may simply write it as $G \subseteq H$ or in the reverse order $H \subseteq G$.
- m is a *subgraph isomorphism* if m is monomorphism and $(u, b) \in E_Q \Leftrightarrow (m(u), m(v)) \in E_H$.
- m is an *isomorphism* if it is a subgraph isomorphism, and it is a bijection of the vertices. When an isomorphism exists, we say G and H are *isomorphic* and write it as $G \cong H$.
- m is an *automorphism* if G and H refers to the same graph, and m is an isomorphism. We say that m is the *trivial* automorphism when the identity morphism id_G.

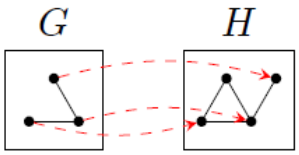
To help explain these definitions, the small graphical examples of [1] will be borrowed in Figure 15. A morphism is just a mapping for all the nodes in one graph H to all the other edges in the graph G. This is also valid, if two nodes from H are mapped to the same node, as we can see in Figure 15a. The only constraint is, that if there were an edge between two mapped nodes in H, there also need to be an edge between the nodes they are mapped to in G. A monomorphism, is the same as a morphism with just one more constraint. It needs to be injective, so we cannot map several nodes in H to one node in G. Note, that since a monomorphism does not require a mapping to be surjective. We do not need to map for all edges and nodes. This is why, the mapping in Figure 15b is a valid monomorphism. The restriction that the mapping of the edges must also be surjective comes with a subgraph monomorphism. If there is an edge between 2 nodes in H, there must also be an edge between those nodes in G. Lastly, an isomorphism is a subgraph isomorphism, with the added constraint that the mapping of the nodes must also be a bijection.



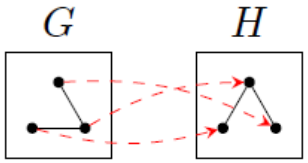
(a) A morpishm



(b) A monomorphism



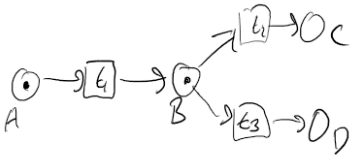
(c) A subgraph isomorphism



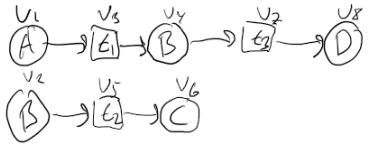
(d) An isomorphism

Figure 15: Example from [1] of the different morphism in definition 6. The red dotted lines, indicates the mappings, that makes the morphism, monomorphism, subgraph isomorphism and isomorphism valid.

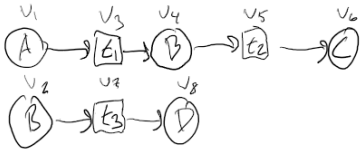
This definition fits the case of an *unlabeled graph*. However, the graphs we will work with are not unlabeled. Let us take a look at an example, to see why the definition 6 is not enough on its own.



(a)



(b)



(c)

Figure 16: An example, that the definition of isomorphism from 6 cannot cover the isomorphism question of occurrence net in a desirable way. The graphs in (b) and (c) are the 2 possible occurrence nets that can be created from (a) and the firing sequence $[t_1, t_2, t_3]$. The tags A, B, C, t_1 and t_2 inside the nodes of (b) and (c) is the auxiliary data of the nodes (The places that the O-places was build from, and the transition that was fired) and the tags v_n above the nodes of the occurrence nets of (b) and (c) are the labels. (b) $\cong$ (c) according to definition 6, but the two occurrence net are not equal.

In Figure 16 we can see a Petri net, and two occurrence net build from the firing sequence $[t_1, t_2, t_3]$. If we follow the definition of isomorphism that we saw from definition 6, then the occurrence net from Figure 16b and 16c would be isomorphic. However, we do not want these two occurrence net to be categorized as isomorphic. The reason is, that the casualty of the firing of the transitions are not the same for the two occurrence net. In the occurrence net of Figure 16b, the firing of t_3 depends on the firing of t_1, and t_2 can be fired independently off the other two transitions. In Figure 16c t_2 depends on the firing of t_1 and t_3 is independent of the other 2. Since the point of the individualized token is to capture these dependencies, we need further restrictions on our definition of isomorphism by including *auxiliary data* on the nodes.

Auxiliary data, is some extra data that will be attached to each node. Note, that this is different from the graph's representation. The auxiliary data is part of the mathematical object of the graph, while its representation is not. In the figures 16b and 16c the difference between the auxiliary data and labels are displayed. The auxiliary data are the tags A, B, C, t_1, t_2 inside the nodes, while the labels are the v_n tags above them. We need the auxiliary data to be taken into account, when comparing the two occurrence nets to capture the dependencies of the firings. However, we could change the v_n labels to anything, and it would not change the occurrence net itself, only the representation of it. In our case, the auxiliary data can simply be a number that represents what place the O-place was build from or the transitions ID from the Petri net. Given this, we can create a new definition that takes inspiration from [10]:

Definition 7 m is a *labeled isomorphism* if it is an isomorphism, and $\forall v \in V$, $l(v) = l(m(v))$.

Here I have chosen to follow convention, which is sadly to call a graph with auxiliary data a *labeled graph* and an isomorphism with labeled graphs a labeled isomorphism [10]. Definition 7 simply states, that a labeled isomorphism is an isomorphism, where the labels from H to G for all vertices also match. Under this definition, the graphs of 16b and 16c are not isomorphic. An example of an occurrence net, with a labeled graph that is labeled isomorphic to the occurrence net in Figure 16b can be seen in Figure 17. This occurrence net was also created from the Petri net in 16a, but from the firing sequence $[t_2, t_1, t_3]$.

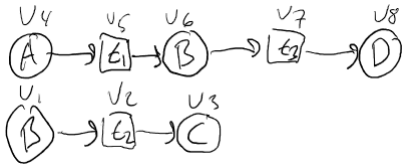


Figure 17: The only valid occurrence net that can be created from the Petri net, and the firing sequence $[t_2, t_1, t_3]$. The graph of this occurrence net, is labeled isomorphic to the graph of the occurrence net of Figure 16b.

Going forward, we will never use isomorphism on graphs that are not labeled again. Definition 6 was only introduced, as a building block to labeled isomorphism. Since this is the case, I will always refer to labeled isomorphism as isomorphism from now on. If 2 occurrence nets H and G are compared for labeled isomorphism, then I will similarly use $H \cong G$ to indicate that they are labeled isomorphic.

In principle, we are ready to start to build our solution. However, as we have already discussed shortly, we will need a method for telling if 2 occurence nets are isomorphic. Before we start implementing Petri nets and occurrence nets, let us take a short detour to discuss how this can be done.

4.4 Finding isomorphism with VF2

5 Overall design choices.

Before we start discussing how the different elements have been implemented, we have some design choices to discuss. First, the programming language that was used.

5.1 Why c++ was used for this project

The programming language C++ was chosen for a couple of reasons. Since MØD is written mainly in C++, and the hope is to use the library written for this thesis with MØD in the future, choosing C++ would make this process easier. More importantly, C++ have some distinct advantages when it comes to the requirements of this project. Since there does not exist (to my knowledge) any library for any of the bigger programming languages that has implemented Petri nets and Occurrence nets that is still supported, we would have to implement this from scratch. The most convenient way to implement both Petri nets and Occurrence nets, are as graphs with auxiliary data on the nodes. Furthermore, we will need to create at least some of the reachability graph, to find all the paths from the initial marking to the end marking (more on this later), and we will need to check if the generated occurrence nets are isomorphic with the `vf2` algorithm. Here, C++ has access to the excellent boost library, that has a very generic and fast implementation of graphs, and it has a well-made implementation of the `vf2` algorithm [3]. This fact alone, makes C++ a good choice to use for the implementing of our solution. Another factor is speed. Given that the reachability problem for even a 1-safe Petri net is NP-hard (our Petri nets will not have this restriction in general, and will thus be harder), we cannot afford to have the programming language itself slowing us down from finding a solution[6]. Choosing C++ for the performance benefits in memory and speed over most other languages, can be beneficial for us [7].

5.2 Coding style

Do I even need this section?
The other design choice that was made, was with regard to the coding style. In this project, we will be implementing concepts that either has not been done before to my knowledge, such as occurrence nets, or has not been documented in details, such as the algorithm we will use to find all the reachable markings in a depth first search like manner (more on this later). As we do, we will encounter problems where it is not initial clear what method or data structure that will be best for the job. One way to distinguish the different solutions from each other, is to benchmark them against each other. But this can potentially pose a problem. Let us for instance pretend, that we choose to implement the data structure that will hold the number of tokens on each place as an `unordred_map`, where the key is a place and the value it maps to are the number of tokens in that place. The map would be a member of some class, that would encapsulate a Petri net. Then we later come up with the idea, to use a `vector` instead for

this job. As long as we can get the index where the tokens for a place are stored in constant time, we can also look up the number of tokens that a place has in this `vector` in constant time. These to data structures have very different performance characteristics. Which is better? That will depend on how the data structure is used. But we cannot always know all the requirements we will need to adhere to beforehand. Sometimes, a new need that we did not predict could emerge. When this happens, or when we get a good idea to how we can make a certain part of our code better, we want to make the change to our code as noninvasive as possible. We do not want to have to copy our current Petri net class, name it something slightly different, and replace the `vector` with an `unoredered_map`. If we do this, we will create a massive amount of almost identical code, that we still have to maintain. And if we make a change in one of them, we have to remember to make the same change in the other. Otherwise, we claim to make a fair comparison between them. What we need to do, is to write our code generically and deploy the correct design patterns.

In C++, one way to write code generically is to use `templates`, which will work well in our case. Instead of changing the entire class, we can introduce a template parameter, which we can call a `Token_container`. The job of the `Token_container`, would be to store the tokens for each place. We can then build an interface, that any token container will use. If we later want to add a new `Token_container`, all that is needed is to implement it, so that it follows the interface of a `Token_container`. Then we can simply use it as the template parameter in our imaginary Petri net class, and we would not have to change any other code. This means, that trying new design and ideas is simple and fast, and we do not create redundant code that needs to be maintained. This approach will be used though out the implementation. Not just when implementing the data structures, but also the algorithms. We will try to make it possible, to switch out different part of the code in a noninvasive way, when there are several reasonable choices that could be made.

6 Implementing Petri nets

Now we are finally building our solution. The first thing we will need is an implementation of a Petri net. We will follow the theoretical setup from Definition 1, and implement a class `Net` that will hold the structure of the Petri net, and a class `Marking` that will hold the state of it the Petri net. We do this, to keep a clean separation of the two task. This is advantageous for two reason. First, if someone wanted to dive into the code, the theory they would have read from Section 3 and what would be in the code would follow each other closely, making it easier to understand. Second, there are several cases where we need to pass a `Net` around, but do not need a `Marking` to come with it. We also want the copying of a Petri net to be relatively cheap. Since the structure net N from Definition 1 can and will be implemented as a graph, copying it is a slow and memory intensive task. This can be solved, by having the `Marking` have a pointer to the `Net`, which can be created without a marking. We can then also create many different Petri nets, that all point to the same `Net`. Now that we have argued for this separation, we will first explain how the `Net` was implemented. Since the places and transitions are the core building blocks of a Petri net, we will shortly discuss how they were implemented.

6.1 Implementation of the `Places` and `Transitions`

Let us start with how the class `Place` was implemented. A `Place` is a very simple class. It consists of 2 numbers, an `id` and `token_index`. It has operators to compare for equality, less than and hash a `Place`, and methods to retrive the `id` and `token_index`. The `id` is used to find the node of the place in the `NetGraph` and is thus unique for all places and transitions. The `token_index` is used in the `Marking` class, to look up how many token there is in that place. Therefor, every `token_index` is unique among all places. As we can see, a `Place` knows very little about itself. It does not contain the token count, nor does it know what transitions it is connected to. That information has to be looked up in the `Marking` and `Net`. The reason we do not store this information in the `Place` class, is that we want to be able to copy the information of the placement and number of tokens in each place fast. This will be needed to explore the reachability graph. We will not go into detail with this problem now. But just to give the intuition, when the reachability graph is generated, it is desirable to know what markings that have already been seen, to avoid exploring them again. If this information was stored at the `Place` class, we would have to store them in some data-structure, and then copy that data-structure for each unique marking. Since a `Place` will no longer be a tiny class, this would make the copying of this information slow. What we will do instead, is copying the data structure that holds the information of how many tokens that are store in each place. One way to implement this, would be a `vector` of integers, where the index corresponds to the `Place token_index`, and the integer store at that index is how many tokens are at that place. This will be all we need to compare, to tell if the marking we currently have, is equal to any

of the other markings we have stored. Since comparing two `vector`'s of integers are fast, these comparisons will be fast. We also suspected, that we might need to copy the places themselves. Making them as small as possible, would have been beneficial in this case. However, we never needed to copy the places in the end.

The `Transition` class is similar to the `Place` class. It only has one member `id` that is a number. Like the `Place` class `id`, it tells us what node in the `NetGraph` the transitions corresponds to, and is therefore also unique among all places and transitions. Like the `Place` class, it also has operators to compare `Transition`'s with each other and an operator to hash them. This means, like the `Place` a `Transition` knows almost nothing about itself. What places a `Transition` is connected to is stored in the `Net` class, and whether it is enabled or not is stored in the `Marking` class. The reason is similair to why we made `Place` a small class. We will need to copy a `Transition` often. It happens in 4 instances. To keep it simple for now, we will only explore 2 of them, since the last 2 relates to an algorithm that has not been explained yet. These last examples, will be explained in Section 7 and Section 8. The first example relates to when we need to store all the firing sequences, or paths, that we find in the reachability graph from an initial marking to an end marking. For each path or firing sequences, we will need a copy of each `Transition` that is a part of that firing sequences or path. The second example is in the construction of the occurrence nets. Here we will also use the `Transition` class, when we build the O-transitions. And for each transition fired in the σ that was used to create an occurrence net, we will need 1 copy of a `Transition`. If there are many of them, this can take up a lot of memory and be time-consuming. Thus, we want a `Transition` to be as small as possible.

The details, of how an `id` from a `Place` or `Transition` is used will be explained in Section 6.2 and 6.4, while the details of how `token_index` will be used will be explored in Section 6.4. We will start with how the structure of the Petri net is implemented in the class `Net`.

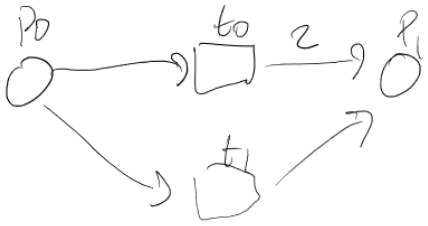
6.2 Implementation of the `Net`

The `Net` class has been implemented where the class has a private member, a BOOST graph called `NetGraph`, and nodes and edges have properties[3]. The edge's property `EProp` is the weight of the edge, and the nodes of the graph have the property `VProp` that holds what `Kind` of node it is, and the node's `id`. The `Kind` is simply either a 0 or a 1. It is a 0 if that node is a `Place` and a 1 if the node is a `Transition`. The `id` is same, as the `id` of the `Place`'s and `Transition`'s and is assigned when the node is created. The `NetGraph` can be accessed, with the method `get_graph()` that returns a `const` reference to the graph. Since we want to be able to look up information about both `Place's` and `Transition`'s without looking through the entire `NetGraph`, if we already have a `Place` or `Transition` (remember that these 2 do not know anything about themselves), we need some data structures to hold this information. We have a `vector<Place>` called `places`, and an `unordered_map<size_t,TransInfo>`

called `transitions` that contain the information of the transitions in the `Net`. Both of these data structures are private, and can be accessed though read methods. Before we explain what the `TransInfo` class consists of, lets us briefly cover `places`, since it is so simple. This is simply added, so that the user can iterate though all places of the `Net`, without exploring the `NetGraph`. This is a bit wasteful, but since we do not need to copy a `Net` at any point, we can sacrifice a little memory for quick access to all of the places, for the user's convenience.

The `TransInfo` class is where all the relevant information about a `Transition` is stored, except for if it is enabled or not. It has the `Transition` that it store information about, and two `vector<Transition>` called `can_enable` and `can_disable`. The optimizations that are used for these two list will be covered in section **??**. A quick summary of that section: For every transition, there is a number of transitions, that transition can either be enabled or disable if it is fired. Let us store those, to get quick access to them. Remember from Section 3, that a net from a Petri net, cannot be said to be either enabled or disabled, since it contains no tokens. This information will therefore be stored in the `Marking` class. We could also have added the places that this transition is connected to in `TransInfo`. However, storing this information here will not help us later, so this is omitted.

Now we need to look at how a `Net` is build. Instead of mentioning the methods of the class one by one and explaining them, we will make a small example of a Petri net, and show how one would create the net N of it using the `Net` class.



(a)

```
Net net = Net();
Transition t0 = net.add_transition();
Transition t1 = net.add_transition();

Place p0 = net.add_place();
Place p1 = net.add_place();

net.add_arc(p0,t0);
net.add_arc(p0,t1,2);

net.add_arc(t0,p1);
net.add_arc(t1,p1);
```

(b)

Figure 18

Figure 18 depicts an example of how to create `Net`, and the result of the code. First, we create a Net. This will initially be empty. Then we call `add_transition()` twice. When this method is called, the first thing that happens is that it finds the `id` for the transition. Here, the `id` is simply set to be the same as what the boost graph would, the current size of the graph, before the node is added [3]. The transition `t_0` thus gets an `id` of 0, and `t_1` gets and `id` of 1. Then, a new node is added to the `NetGraph` where the node's `Kind` (with

the value 1) is set to a transition and the `id` of the `VProp` is set to the same as the transition's `id`. Lastly, create a new `TransInfo` from the new transition, and put it into `transitions`. Then the newly added transition is returned.

The method `add_place()` works in a very similar way. The `id` of the new `Place` is added in the exact same way, and a node is added to the `NetGraph` with the only difference being that the `Kind` is a place (with the value 0) instead. The new `Place` is added to the `vector Places`, and the newly added `Place` is returned.

The methods `add_arc` are heavily inspired by how edges are added in the BOOST library. It either returns a `True` or a `False` depending on if a new edge was added or not. We do not allow for multiple edges in any pair of transition and places, nor do we allow an edge to be added with a weight less than 0. If the new requested edge meets these requirements, then it is simply added to the `NetGraph` with the given weight. Then the method `initialize_enable_disable_vectors()` is called. This method will make sure, that the `can_enable` and `can_disable vector`s of all transitions are correct, given that this new edge could change that fact. This is a bit wasteful, and why this choice was made will be discussed in Section 6.3. And we are now only left with explaining the optimization of Section 6.3, before we can move on to the implementation of the `Marking` class.

6.3 Checking enabledness relation between transitions

In this section, we will explain the optimization that LoLA that they call *Checking Enabledness*, but in a bit more detail [13]. The observation that is the inspiration for this optimization is quite simple. When a transition t is fired in a Petri net, this can affect the enabledness of the other transitions. The reason for this is, that t will remove tokens in its preset, and put tokens postset. Other transition, may be connected to the places that have these tokens too. So they may be enabled or disabled. Thus, we have to check the enabledness of the

6.4 Implementation of a `Marking`

6.5 Getting the enabled transition stack

As we have seen the `explore_state_space()` algorithm needs to retrieve a stack of all enabled transition, every time a new marking is reached. How this stack retrieved could greatly impact performance. The naive approach is to iterate though all transitions and put them on a stack, if they are enabled. The problem with this approach is, that the number of enabled transition will often be fewer, sometimes allot fewer than the number of transitions in the Petri net. A better approach is to keep some sort of data structure updated that contains the enabled transitions, and use that to return the stack. Note that no matter what data structure we choose for this job, `explore_state_space()` can never use references to that data structure, to save on the stack. The inner stacks on the transition stack can never change, unless they are popped. But if they are

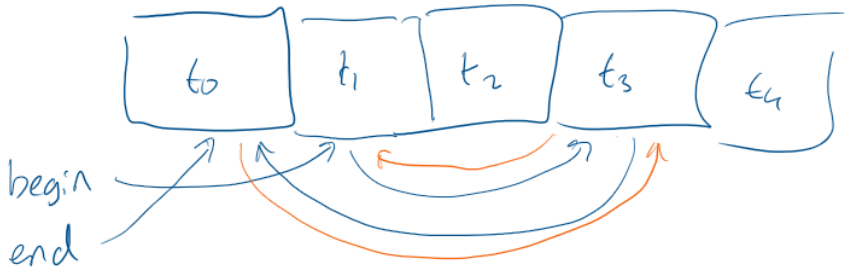
references, they could change every time a `fire()` or a `backfire()` is called when exploring the state space. Therefor, it has to be either a copy of the data structure we chose (if it itself is a stack of the enabled transitions) or it has to be able to construct the stack and return it `explore_state_space()`.

The designers of LoLA also encountered this problem, and they solved it with some kind of linked list. I have not followed their example fully, since they seem to maintain an in-order of the transitions in their linked list, since they have implemented the ability to delete and insert transitions to any position in their linked list. I have not been able to figure why this could be helpful, so that detail is not present in my solution [13]. The data structure that is inspired by LoLA's solution is called `Linked_list_alike`.

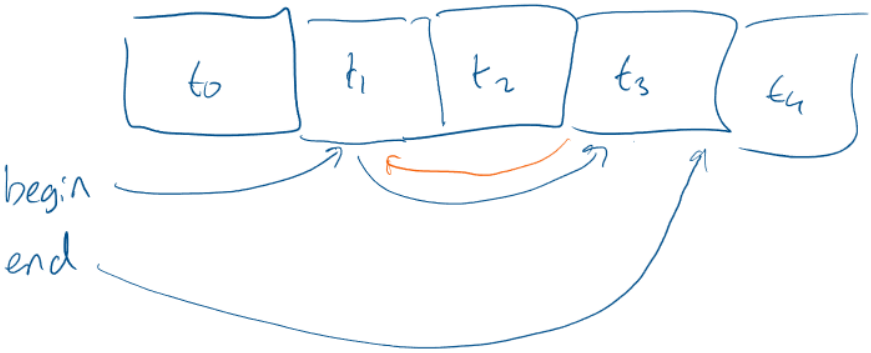
`Linked_list_alike` is an array with an internal double linked list inside of it. It contains all transitions in nodes, that also have a `forward` and a `backward` index, which are both set to -1 if the transition is not enabled. These functions as the pointers of the nodes. Lastly, `Linked_list_alike` has a `begin` and `end` index. When the linked list is empty (no transition is enabled) then `begin` and `end` are both set to -1. The position of the node's in the array, is determined by the transitions id's. Since they go from 0 to n (number of transitions) we can use this to jump directly to a transition's node when we want to update it, and not do a linear search as is typically done with linked lists [5].

A transition is added to the linked list by passing it to the method `insert()` when it is enabled. When a new element is added, it is always put in the back of the linked list. It is thus inserted by following the end index setting the new nodes pointers. The forward pointer will be set to -1 and the backward pointer is set to the old end of the linked list. Then the old end is reset, by adjusting the node's forward index to point to the new node. `insert()` running time is thus a constant function. Note that the backward pointers of the nodes, are not used at all in this method. However, we still need them, when we want to delete a node.

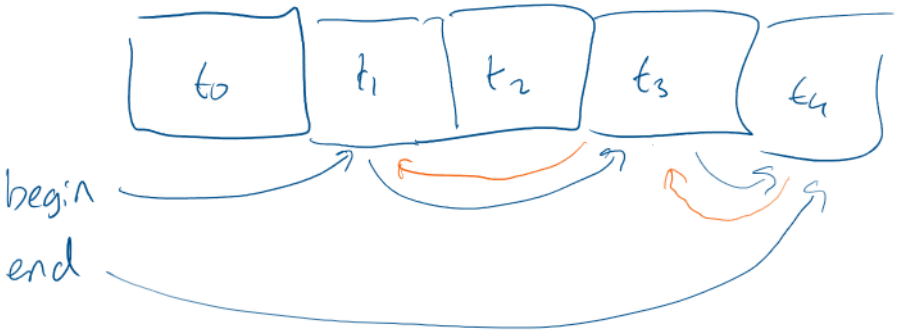
Removing a node, with the method `erase()` when the transition is no longer enabled, has a slightly different behavior than the typical linked list. Instead of searching from the `begin` index, we can simply jump directly to where the node is stored. The forward and backward index of the node is simply set to -1, and the predecessor and successor of that node has their forward and backward index adjust in the typical manner [5]. Note that we need the backward pointers of the nodes in this case. If we want to adjust the predecessor of a node's forward pointer, we need to do know what node is the predecessor. If the nodes do not have a backward pointer, then the only way to find it, is to do a linear search from `begin`. With backward pointers, this update can be done in constant time. Thus, `erase()` has a constant running time as well.



(a) Initial state.



(b) Remove t_0 from the enabled transition.



(c) Add t_4 to the enabled transitions.

Figure 19: A small example of a `linked_list_alike`. It has 5 transitions where t_0, t_1 and t_3 are enabled. The blue lines are the "pointers" or indexes that goes forward, while the orange are the "pointers" or indexes that go backwards for a given node.

The third method of `Linked_list_alike` called `get_enabled_transitions_stack`, is to create the stack of enabled transitions. It does this by following the point-

ers of the internal linked list. The process is started from the `begin` index, and then the forward indexes are followed until the current node's forward pointer is equal to -1. The method returns a `std::vector<Transition>` which size is fitted to the number of enabled transitions. This is important to do, since there will be a lot of these `vector`'s in memory. Adding this information to the `Linked_list_alike` and keeping it updated with each `insert()` and `erase()`, means that the `std::vector<Transition>` can be initialized to the correct size, and there is no waited time with memory allocations either, making the creating of `vector` as fast and memory efficient as possible.

One last observation, is the order of the transitions is not preserved. Normally, that is not a concern. However, when developing and debugging the `explore_state_space()`, it is helpful to be able to enforce a certain order of firing the transitions as the reachability graph is explored. It is also useful, when writing unit tests. For this purpose, another data structure has also been implemented called `Use_set`. It can be given either a `std::set<Transition>` or a `std::unordered_set<Transition>` as a template argument. Then it will use these containers, to keep track of what transition is enabled, with their `insert()` and `erase()` method, while its iterators is used to create the same `std::vector<Transition>`. `Use_set` uses the same interface as `Linked_list_alike`, so they can be used interchangeably in the `Marking` class. These will also later be used for benchmarking `Linked_list_alike` against two other reasonable solutions, to see how effective it is.

6.6 Token_containers

6.7 Checking if a transition is enabled after a fire and backfire

Some of the functions that will be called often are `fire()` and `backfire()`. Profiling has revealed, that up to 80% of the time could be used on these two functions while exploring the entire reachability graph, and when a straight forward implementation without any optimizations was used. It is thus important that any unnecessary work is removed from them.

The same profiling results showed that both `fire()` and `backfire()` used around 60% of their time calling and executing the function `is_transition_enabled()`. So if this function could be optimized or called less often, it could lead to a speedup. As the name indicates, the job `is_transition_enabled()` is to check whether a transition is enabled or not. It is called in the instances, when a transition is fired or backfired, and other transitions that shares places with it could be affected. Figure 20 exemplifies this.

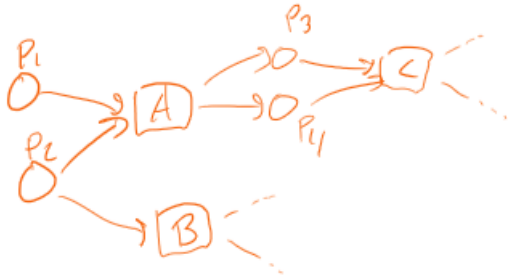


Figure 20: A is the transition we fire, and that can affect B and C's enabledness.

If A is fired, then it will take tokens from P_1 and P_2 and place some tokens in P_3 and P_4. Let us pretend that the transition B was enabled before A was fired. Depending on the number of tokens in P_2 and the weight of the edge from P_2 too B, this could disable B. Likewise, if C was disabled before A was fired, the firing of A could enable C. Thus, we need `is_transition_enabled()` to check if the states of B and C when A is fired. `is_transition_enabled()` needs to check for all places in B and C's pre-set, the weight of the edge going from the place to the transition, is higher than the number of tokens in that place. If this is true for any place/edge combination, then the transition needs to be disabled. If it is false for all of them, the transition should be enabled. However, there are cases where we can omit these checks. If B was already disabled before A was fired, we do not need to check if it should be enabled or not. The reason for this is, that firing A removes tokens from B's pre-set. If it is already is disabled, then removing tokens from the pre-set of B cannot enable it. The reverse is true for C. If C was already enabled before A was fired, then it will also be enabled after A put tokens in the pre-set of C. Thus, we do not need to check the pre-set of C either in this case. The same logic can be applied for backfire but in reverse. C is now getting tokens removed from the places of the pre-set, so if C was disabled before A was fired, then the call to `is_transition_enabled()` can be skipped. The same logic can be used for B.

There is one last call to `is_transition_enabled()` that we need to consider, and that is to the transition that is given to `fire()` and `backfire()` themselves. As we know, after a transition is fired, it might get disabled. So we need a call to `is_transition_enabled()`, or check it as we remove tokens from the transition's pre-set. The amount of work done is the same, just at two different times. So there is anything to gain here. The same cannot be said for `backfire()`. We do not need to check, if the transition backfired is enabled after it has been backfired, since we can deduce that it must be enabled at that point. A transition we backfire in `explore_state_space()` must have been enabled when it was first fired, since it could not have been fired otherwise. Thus, it must become enabled again after the call to `backfire()`. Thus, we never have to call `is_transition_enabled()` to check the backfired transition.

7 Exploring the state space

Skriv om at Transition skal være lille, da vi har vores stack af transitions.

8 Getting all paths

8.1 Answering the reachability problem

The simplest and most naive way to answer a reachability problem, is to produce the reachability graph from the petri net, and then check if the target marking is in the reachability graph.

8.2 Explore state space

Algorithm 1: Explore state space

Input : `marking`, The initial marking
`end_marking`, The target marking

```
1  stack = [null,(enabled(Marking))]
2  while stack is not empty do
3  |  (t_from,enabled) = stack.pop()
4  |  if enabled is not empty then
5  |  |  transition = enabled.pop()
6  |  |  stack.push((t_from,enabled))
7  |  |  marking.fire(transition)
8  |  |  stack.push(transition,enabled(marking))
9  |  else
10 |  |  if t_from is equal to null then
11 |  |  |  continue
12 |  |  marking.backfire(t_from)
13 |  |  path.pop()
```

8.3 Is reachable

8.4 Find a path

8.5 Find all paths v1

Once `find path` is defined, it is trivial to create the algorithm to find all firing sequences from the initial marking to the target marking. This algorithm will be referred to as `Get_all_paths`. A trivial solution is to use the same pseudocode from `get path` and make two small changes. The first is to add some container that will hold all of the paths. The second is to save a copy of the *path* in that

container every time the target marking is found, and not stop the exploration of the state space before the entire state space has been explored.

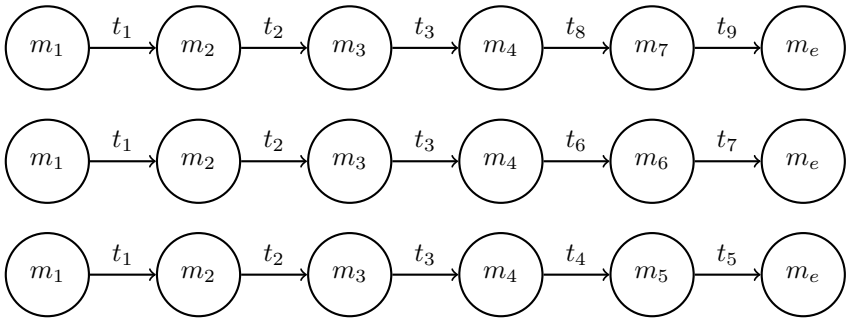
Algorithm 2: Get all paths v1

Input : `marking`, The initial marking
`end_marking`, The target marking
Output: `all_paths`, a container with all the paths

```
1  Initialize the path and the all_paths
2  if marking is equal to end_marking then
3  |  return All_paths
4  stack = [null,(enabled(Marking))]
5  while stack is not empty do
6  |  (t_from,enabled) = stack.pop()
7  |  if enabled is not empty then
8  |  |  transition = enabled.pop()
9  |  |  stack.push((t_from,enabled))
10 |  |  marking.fire(transition)
11 |  |  path.push(transition)
12 |  |  if marking is equal to end_marking then
13 |  |  |  all_paths.push_back(path)
14 |  |  |  marking.backfire(transition)
15 |  |  |  continue
16 |  |  stack.push(transition,enabled(marking))
17 |  else
18 |  |  if t_from is equal to null then
19 |  |  |  continue
20 |  |  marking.backfire(t_from)
21 |  |  path.pop()
22 return all_paths
```

Algorithm 2 will return all firing sequences, but it has some major flaws. The first one is that any paths that goes through the same n markings, will have firing sequences that have the same $n-1$ transitions in the same order, and thus wasting space in this scenario. In the worst case, all paths to the target marking will be the same until the last two transitions. The memory consumption will thus be $\Theta(n \times m)$ where n is the number of paths, and m is the length of them measured in the number of arcs in them. In this case $\Theta(n \times (m-2))$ of the arcs are the same for all paths. This scenario is depicted in Figure 21 for a small state space graph. No matter what is stored, paths or firing sequences, we will need to store many copies of `Transition`'s, which is the last reason, we made the `Transition` class as small as possible, as discussed in Section 6.1.

Figure 21: All paths memory consumption worst case



The second flaw is more troublesome, since it is not case dependent and will in many cases prevent us from computing a solution in a reasonable amount of time. If we only collect firing sequences and no paths while exploring the state space, we cannot make use of any `Collection`. The concept of a `Collection` will be fully explained in Section 8.8. For now, it is enough for the reader to know that a `Collection` is some sort of container that stores all of the seen markings. When a new marking is found while exploring the reachability graph, it is saved in the `Collection`. It can then be used for look ups after a transition is fired. If we have seen the marking before, then we do not have to explore it again and we can simply backtrack. This gives a major speedup of the cost of memory consumption. This idea comes from this LoLA article [13]. As tests will later reveal, exploring the state space without a `Collection` is in most cases meaningless since the algorithm will slow down to an unacceptable degree. Thus, using a `Collection` is required.

However, if a `Collection` is used then allot of desirable guarantees will be lost about the paths that can be recreated from the firing sequences, without any post processing. Such as if all paths are represented by a firing sequence, or if the shortest path or the longest path is present in the returned firing sequences. All that can be guaranteed is that at least one path will be returned if there is a path from the initial marking to the target marking. The figures 22 and 23 highlights the problem. In both figures the green node is the initial marking, and the target marking is the yellow one, while the blue path is the first that is discovered. If we remember the markings 2,3 and 4 with a `Collection` we will have two choices when we discover that our current path leads us to a marking that have already been explored that is not the target marking. It can either be stored as is or it can be forgotten.

If we don't store it, then we don't store any firing sequences that goes though the same markings. This means that only the blue path in Figure 22 and 23 will be stored, and not all paths will be returned. The algorithm is therefore not correct.

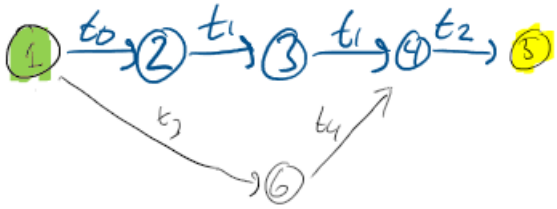


Figure 22: Shortest path will not be saved by *All paths* with `Collection`

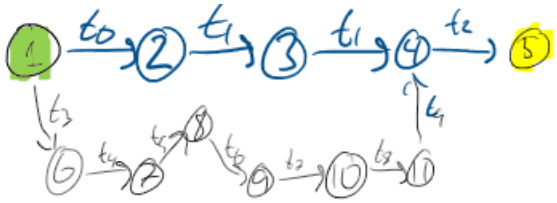


Figure 23: Longest path will not be saved by *All paths* with `Collection`

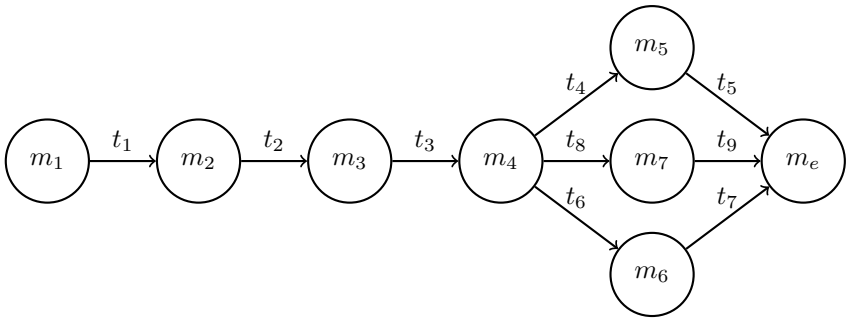
If we do store the firing sequences, when a marking that has been visited before is encountered then the firing sequences themselves will loose many desirable properties. Except for the first, it cannot be guaranteed that the firing sequences leads to the target marking. So they will not be useful, before they have been post processed. The natural thing to do is to create the state space graph from the firing sequences and retrieve the paths from it. These paths can later be used to produce all firing sequences. This is not a very elegant or efficient solution.

The other option is to build the state space graph during the state space exploration. The downside of this strategy is, that the graph itself can become quite memory intensive. However, this is also true if we only stored the firing sequences. Which one is the best memory wise, depends on the state space graph. If it has a lot of paths that share the same edges, then it will be optimal to create the graph during the state space exploration. If the state space graph consist of many isolated paths, then it will be optimal to store it as a list of firing sequences. And this strategy has the advantage of only having one post process step after the state space exploration is complete, namely getting the firing sequences from the graph. For this reason, the later strategy has been chosen in this project.

In the next few sections, we will arrive at what will be the template for the `state_space_exploration` algorithm. We will add the optimization of a `Collection` to it, and make the algorithm more generic. So far, we have seen three different versions of basically the same algorithm. `is_reachable`, `get_path` and `get_all_paths` all explore the reachability graph in the exact same way. The only thing that differs between them, is what information is

picked up along the way and when the algorithms terminates. This introduces duplicate code, which is bad for many reasons, which will be explored later. To avoid this, we can use the strategy of `visitor patterns`, that is also famously used for algorithms such as `depth first search` [5]. This and `Collections` will be explored in the next two sections

Figure 24: All paths memory consumption using a state space graph



I have already discussed some of the problem with `is_reachable`, `get_path` and `get_all_paths` in Section 8.5. If we knew beforehand that these three algorithms would never change, and that it made no sense to try different strategies to build the state space graph, then it may be possible to argue that this is okay. However, this is not the case. We want to try different strategies of building the state space graph, and test them against each other. As an example, we might want to only add paths without cycles in them. Right now, a entirely new and mostly identical algorithm would have to be added. Just like with `is_reachable`, `get_path` and `get_all_paths` the only difference between the mentioned algorithm and the others is the information gathered while exploring the state space graph. The cost of innovating and trying new things is thus quite high, since for each new algorithm we would add allot of duplicate code that has to be maintained. This is also bad for benchmarking purposes. The reason for this is, that we want these algorithms to be directly comparable when benchmarking them against each other. We want everything to be the same, execpt for the information gathered along the way. However, the chance of this not being the case, is higher with many mostly identical algorithms. What if one of the algorithms did not get an optimization that the others did, simply because it was overlook or wrongly implemented?

Thus, the state space exploration part of the algorithm and the information collecting about the reachability graph needs to be separated. If an optimization to the state space exploration part of the algorithm is implemented, we want it to make that optimization in one place, so that it is applied to all algorithms. And we want to be able to try a new algorithm, without having to rewrite much or ideally any code. Thankfully, others have already encountered these problems before and have created design patterns to address them. One suitable for this case is as the title of the section suggest *visitor patterns*.

The idea of visitor patterns is to make adding operations to one or more types easy [8]. In our case, the type would be the reachability graph we explore, and the operations are the different functions that pick up information about the graph as it is explored.

8.6 Visitor patterns

What visitor pattern does, is to isolate one aspect of a function or class that changes and isolates it. In our case, we want to isolate the gathering of information, and when the algorithm should stop, from how the reachability graph is explored.

8.7 State space exploration with visitor patterns

8.8 Collections

As shortly discussed in Section 2 Section, `Collection` is a container that stores markings. However, we have not yet covered them in detail and why we can simply backtrack, if we encounter an already explored marking. This is done in this section. First a definition:

Definition 8 *A collection is a container that holds markings. It has a single operation,* `try_insert()` *that takes a marking and tries to insert it into the collection. If the marking is not in the collection, it is inserted into it, and is given a unique number as an ID. This number is returned along with the boolean value false. If the marking is already in the container, the boolean value true is returned along with that marking's unique ID number.*

As mentioned, the idea of a Collection comes from In LoLA. As they describe it, a collection has two operations, `search()` and `insert()`[13]. These operations are used to update the collection when a new marking is encountered, and to check, after each firing of a transition, if the marking that has just been reached is one that has already been explored earlier. If this is the case, then they claim that it is not necessary to explore that marking again. We can simply backfire and go to the next entry on the stack. [13]. Our definition follows their description, except that the two operations have been comprised into `try_insert()`, and that we return a number that identifies that marking. These two choices will be explained later. For now, there is a more pressing concern, is their claim true? Can we backtrack when we encounter a marking that has been seen before?

To demonstrate that this will go well, we can look at the figures below. In Figure 25 the marking that is found again is the end marking. The question is now, why we can omit to search beyond the end marking for more paths. Since we want all paths, including the ones that go through the end marking, we need to establish that the markings that are past the end marking have already been explored, or will be explored later. We can thus divide this up into those two cases.

The first is easy. If the marking we find already has been fully explore, there is no reason to explore it again. We can then just backtrack and go to the next iteration in the while loop.

The other case is not quite as trivial. When a marking that has been encountered before is found, that has not been explored yet. This implies, that the enabled transitions from this marking already is somewhere on the stack. If we backtrack now, eventually that marking will be reached again, and then all reachable markings from it will be searched by firing those enabled transitions. Thus, there is no reason to explore that marking now, and it can be omitted.

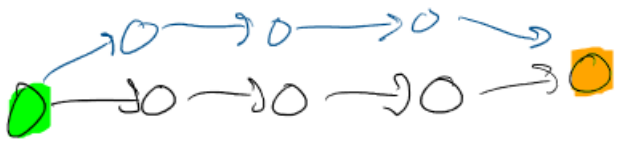


Figure 25: New path to add,

There is another very important observation related to Collections that the authors of LoLA makes. The markings stored in the collection is only used for checking if the current marking has already been seen. They are never altered in any way, and not used for anything else. We can thus compress the markings stored in the collection to save memory [13]. The only requirement for this compression is that it is injective. So for every marking m, the compression $c(m)$ should be unique, and the inverse function $c^{-1}(m)$ does not need to exist. The compression function is allowed to be bijective. It is just not required. This fact gives the option to compress a given marking in a way, where it will not be usable as a Marking-container after it is compressed. Compressing a Marking-container in such a way, will create a new entity, we will call a *collection-container*. The definition can be seen below:

Definition 9 *A collection-container is created from a Marking-container. For every Token-container, a unique Collection-container exists. It is not required but allowed, that the inverse function exists.*

Note that this definition allows to use the identity function (do nothing) on a Marking-container to "create" a collection-container. This might be beneficial to do, if memory is not an issue for the specific instance. The reason is, no matter what function (except for the identity function) is used, it will cost some time to do. Since a `search` in the Collection is needed for every time a transition is fired, and an `insert` for each new marking found, this could save a substantial number of computations. This tradeoff will be tested later.

It is clear from the definition, that a collection works slightly differently than the corresponding construct from LoLA. Our collection has taken the two functions `search()` and `search()` and combined it into one function, `try_insert()`. The reason for this, is that it is more efficient to combine them. After each fire,

we need to insert the marking if it is not there. If it is, then we can immediately return true along with that marking's ID, instead of searching for it just after. The same is true, if the marking is not there. There is no reason to search for it just after, since we just had the correct values. However, LoLA probably does something similar, even though they don't write it [13].

The ID's that the markings are given are not necessary if we were only trying to implement `is_reachable()` and `get_path()`. But they are extremely useful when building the state space graph. How they are used, will be covered in 9.

8.9 Collection-containers

9 Building the state space graph

9.1 Building the state space graph with cycles

Call this APC

9.2 Building the state space graph as a DAG

We will use the same strategy, of using `build_first_path()` and `attach_path()` for creating the graph. No changes are needed for `build_first_path()`. It has to be called for the exact same cases, since creating the first path can never create a cycle. The function `attach_path()` does not need to change either, but when it is called does since this function can create a cycle. We need to either know beforehand if a cycle is created and not add the path, or add it and check if a cycle has been created and then remove it. Let us explore the former option.

10 Creating the occurrence nets

11 Implementation

11.1 Finding all the path to an end marking

11.2 Building the state space graph

11.3 Using type traits, to choose the correct Transition stack

11.4 Optimize `attach_path()`

12 benchmarking

In this section, we will benchmark the code that has been implemented. There will be 3 overall sections. The first section, will benchmark the implementation of Petri net, and state space exploration. As we have discussed, there were a lot of different choices that could be made throughout. Which token containers do we use, how do we retrieve the transition stack, what collection was used and so on. The goal of this first section, is to find the best combination of those choices in the general case.

The next section, will be a comparison with LoLA. Our software can solve problems that LoLA can not, and vice versa. But how do we compare when it comes to the reachability problem, and retrieving one firing sequence from an initial marking to a target marking? Most likely, our solution will be a lot slower than LoLA in most cases. However, we do not know in which cases that we have a comparable performance to LoLA and where we do not. Finding those cases, could reveal where our software has weakness, and what optimizations could be added in future works to improve it.

The last section, will benchmark the generation of occurrence nets. We will try to figure out, what type of Petri nets are expensive to produce occurrence nets for and which are not. We will also run some benchmark for instances from MØD.

12.1 Finding the most efficient setup for state space exploration

This section's goal, is to find the best setup for state space exploration. We have 2 variables we would like to optimize for, running time and memory consumption. To do this, we will go through many of the different options we have, when setting up the `Marking` and `Collection` class. Not all options will be explored, since some of are not relevant. This will be discussed in detail when appropriate in later sections. In this section, we will discuss how the benchmarking test were set up, and where the Petri nets for them came from. Except for the Petri net used in Section 12.1.1, most Petri nets came from the *Model Checking Contest 2022* [**model˙check˙contest**]. The rest were Petri nets that MØD produced. These will first appear in Section 12.1.4 where we compare my tool vs LoLA.

12.1.1 The fastest firing of transition

The first thing we would like to know, is how we get the fastest `fire()` and `back_fire()`. When exploring the state space, these two methods are the ones that are called the most often. This has been revealed, by using KCachegrind on a few examples [**kcachegrind**]. These can be found in the `profiling_results` folder, inside of the `petri_v1` project. These files will reveal, that `fire()` and `back_fire()` account for 40−60% of the time used, when using the `Graph_visitor`. Choosing the setup that makes these two methods as fast as possible, is thus a high priority.

The only choice we have, that could affect how fast a `fire()` or a `back_fire()` is what `Token_container` is used. The `Token_container` could affect the two firing methods, by limiting how fast it retrieves and updates the token count of the places. Here we have 2 possible choices. We could either use a `Vector_marking` or we could use a `Map_marking` as the `Token_container`, that is given to the `Marking`. A quick reminder, `Vector_marking` uses a `vector` to store the number of tokens for each place, and `Map_marking` uses an `unordered_map`. From what we know about the asymptotically running time of these two data structures, the `unordered_map` has an average constant time lookup when given a place, while the vector can have a constant time lookup, if it knows which index to into. Since the every `Place` has a `token_index` as a member, it indeed can make lookup these the tokens of a place in constant time[**cormen har ikke et afsnit om arrays**]. Given this, the `Vector_marking` should be the faster of the 2.

To try to isolate the firing speed as much as possible, a Petri net that has a reachability graph in the form of a DAG has been created. This allows us to use the `Dory_collection`. The reason we want to use it for this benchmark, is that we then do not spend time on storing or looking up markings. The other variable we will lock, is how we retrieve the enabled transitions. Here a `set` will be used, to make the sequence of firings the same as the state space is explored. This will enable us to produce some nice plots, since every marking is always discovered after firing a specific amount of transitions. If we used an `unordered_set` or the `Linked_list_alike` then this would not be true. For this purpose, a `set` will be used to retrieve the enabled transitions for all benchmark until Section 12.1.3, where we will test how much implementing the `Linked_list_alike` us in terms of performance.

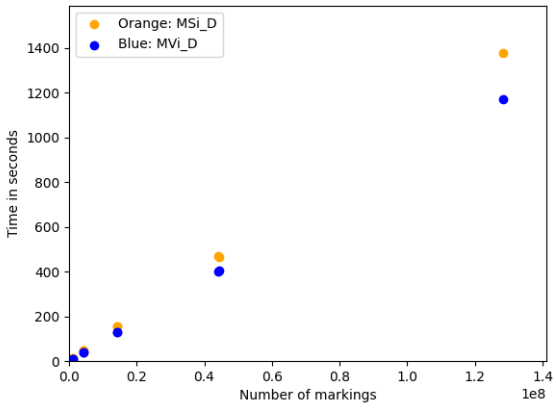


Figure 26: Benchmark of `Vector_marking` vs `Map_marking` in terms of speed. The orange dots are the results of reachability queries to different markings when a `Map_marking` was used as the `Token_container` in the `Marking` class. The blue dots are the corresponding for the `Vector_marking`. The x-axis is the number of markings that were visited. Because the `Dory_collection` were used in this benchmark, it is not the number of unique markings visited.

As we can see in Figure 26 the `Vector_marking` is indeed the faster of the 2 `Token_containers`. There are also results for memory usages, but those are not interested in this case for 2 reasons. First, the memory usage was around 0 MB for this benchmark. Second, it was the same for both. The reason the memory consumption is so low, is that there were not many unique markings that were visited in this benchmark. The `Dory_collection` is to blame for this. Even though, that the reachability graph only contained 440 different markings, it still took `expore_state_space` a really long time to visit them all, since it never stored the markings it had already seen. Thus, it had to revisit the same markings over and over again, as we also discussed in Section 7. With just some sort of collection, even a really simple one, will make this same example run in milliseconds. A collection is thus a requirement, and we will look at which is the best in the next section.

12.1.2 Benchmarking Collections

As we know, the collections have 2 jobs. Store the different markings as they are seen for the first time, and allow for lookup of all previous seen markings. Given this fact, both running time and space used is interesting from a benchmarking perspective. What we would like to learn, is which collection is the fastest, which is the most memory efficient and if there is a tradeoff. We will not be examining the `Dory_collection` in benchmarks of this section, since that collection is not useful in most cases, as we have seen. This leaves us with the `Collection_map` as our collection to benchmark. As a reminder, it stores the given marking as in

either a `map` or `unordered_map`. It has 2 template parameters, `Input_container` and `Stored_container`. The two types can either be the same, which means that no compression of the `Token_containers` that is given as input will be taking place. If the `Input_container` is the same as the `Token_container`, then we can choose to compress it to a `Sparse_Vector`. This gives us a some options to test. The goal of these benchmarks is to discover what the best combination is in general.

The first benchmark, will be comparing storing the `Input_container` in a `map` vs `unordered_map`. Here we want to learn, which of the two is faster at storing and retrieving the stored `Token_containers`, since there is some disagreement depending on the specific setting. However, we do expect that the `unordered_map` is the fastest in general, since it does have the better running time asymptotically running time on average, and no implementation optimization should be able to make `map` faster in practice [5]. We do not expect there to be any difference in how much memory they each use. Thus, since are testing the speed of the two maps, we will only use `Vector_marking` as the `Input_container`, and we will not compress in this test, so the `Stored_container` will be the same. Furthermore, we are only interested in the performance of the chosen map in this benchmark. So we want to isolate it as our testing variable as much as possible. To do this, the `Is_reachable_visitor` was chosen, without getting the path itself, to not influence the memory or time results. A `set` was again used to store the enabled transitions, so that the firings always happened in the same order. This makes us able to produce some nice plots.

And as we can see in Figure 27, reality our expectations in this case. The `unordred_map` was significantly faster than the `map`. If the instances were small, then it made no real difference what map was used. However, if it was a bigger instance as we can see in Figure 27a, then the `unordered_map` was a lot faster.

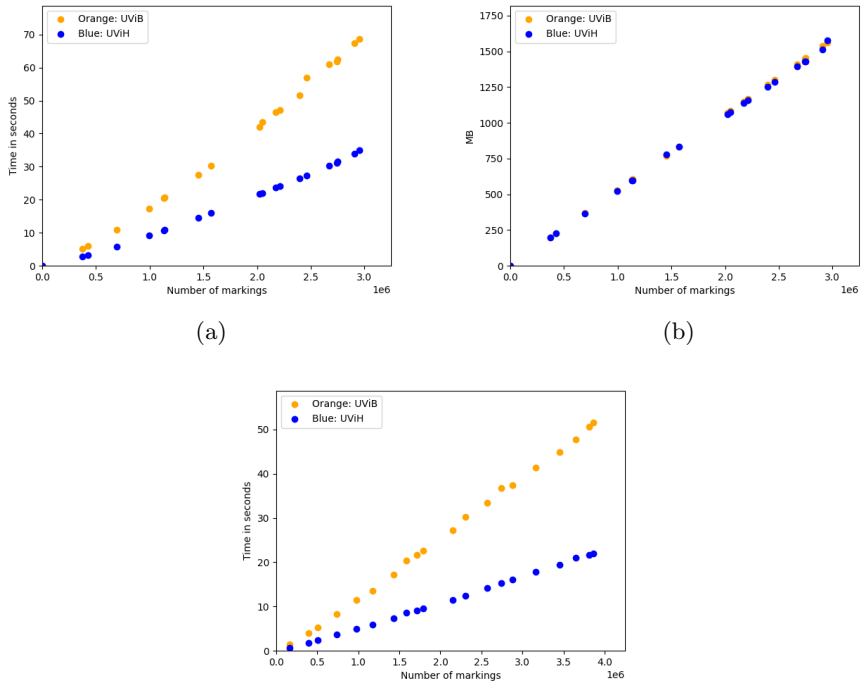


Figure 27

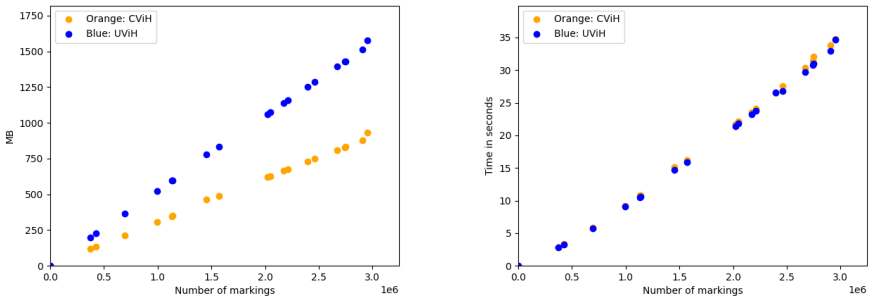
Looking at the memory consumption between the two maps, reveals that there were no big gaps between the 2. Even when using a Petri net that would use up toward 1.75 GB of memory, they both used approximately the same. Given these facts, there is no reason to choose `map` over `unordered_map` going forward. The `unordered_map` is much faster, and just as memory efficient as `map`.

Now that we have settled on a map to use, the next question we would like to answer, is about the behavior of `Collection_map` when the `Input_container` is a `Vector_marking` and the `Stored_container` is a `Sparse_vector`. As we saw in Section 6.6, compressing a `Vector_marking` to a `Sparse_vector` only saves memory, if the `Vector_marking` is sparse. If this is not the case, the `Sparse_vector` may use more memory than the equivalent `Vector_marking`. This signifies that it will be case dependent, whether using the `Sparse_vector` as the `Stored_container` will lead to memory savings. We need to test for these cases.

To accomplish this, when all reachable markings were calculated for a Petri net with the `Seen_markings_visitor` it checks how many of the markings in the reachability graph are sparse. It then saves either a `True` or a `False` in the second to last line. If the line contains `True`, then more than half of the

marking were sparse. Otherwise, less than half were sparse. The next line has the percentage of markings that were sparse. This information is not a perfect indicator to how well the `Sparse_vector` will be able to compress the average marking in that reachability graph. There will be a big difference when the average marking only has 1 place with tokens in it, vs $\frac{1}{2}|P|-1$ of them. In the first case, the compression will have a big impact on the memory used, while there will be almost no difference in the latter, compared to if the `Vector_marking` was not compressed to a `Sparse_marking`. However, the markings in the reachability graph tends to be either extremely sparse or dense in most cases. So in practice, if the analysis shows that more than half of the markings are sparse in a reachability graph, then the compression to `Sparse_vector` works quite well, as we will see. Regardless, this is still a factor that we should keep in mind.

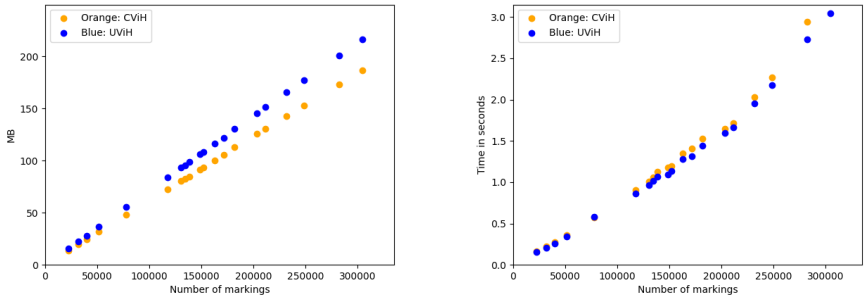
Just as the last benchmark, the visitor `Is_reachable_visitor` was used for the same reason as the last benchmark. The same goes with using a `Set` to store the enabled transitions. Lets us start with an example, where the reachability graphs markings are all sparse. Figure 28 is from the AutoFlight Petri net from the Model Checking Contest 2022 [**model˙check˙contest**], specifically the file they call `afcs_04_a`. As we can see in Figure 28b, there is no gap in how much time they each use to reach the end markings. However, there is a big gap when it comes to memory consumption. And as expected, it increases as the number of markings that were explored grows. In this specific benchmark, when the `Vector_markings` were compressed, the state space exploration used a maximum of 934.37 MB, while if they were not compressed it used 1.57 GB. The uncompressed version used almost 69% more memory than the compressed one, and it did not cost us anything when it comes to speed.



(a) (b)

Figure 28: AutoFlight Petri net from the *Model Checking Contest 2022* [**model˙check˙contest**], specifically the file call `afcs_04_a`. The blue dots are for the instance, where the `Vector_markings` are compressed to `Sparse_markings` before they are stored in the `Collection_map`. The orange dots are the results, where the `Vector_markings` were not compressed before they were stored in the `Collection_map`. The visitor `Is_reachable` was used, and the enabled transitions were stored in a `set`.

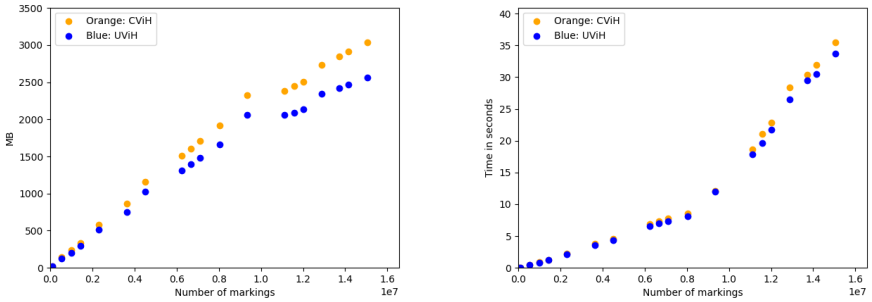
Another interesting result is depicted in Figure 29. Here, the markings of the reachability graph are also always sparse. However, it does not result in any significant memory saving. The reason for this is that the markings in the reachability graph are just barely sparse. The good news is, that we do not lose any performance, compared to when we do not compress the `Vector_markings` to `Sparse_markings`.



(a) (b)

Figure 29: AirplaneLD Petri net from the *Model Checking Contest 2022* [**model˙check˙contest**], specifically the file call `AirplaneLD-pt-0020`. The blue dots are for the instance, where the `Vector_markings` are compressed to `Sparse_markings` before they are stored in the `Collection_map`. The orange dots are the results, where the `Vector_markings` were not compressed before they were stored in the `Collection_map`. The visitor `Is_reachable` was used, and the enabled transitions were stored in a `set`.

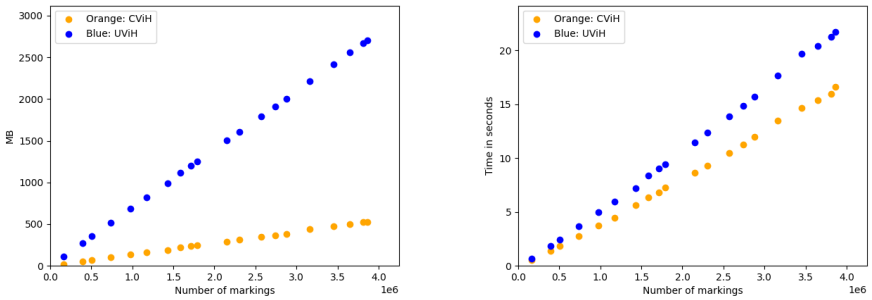
We have also included a benchmark case, where the compressed version ended up using more memory than the uncompressed version. Here, the compressed version ended up using 3.03 GB at its maximum, while the uncompressed version used 2.56 GB. Crucially, the compressed version did not use any more time completing the reachability queries, than the uncompressed one. This was the worst case I could find, that did not run out of memory before it had found all reachable markings for that Petri net. It is possible to create a worse case for this compression technique. Simply create a Petri net, where every place starts out with n tokens, have all the weights of the arcs be 1, and do $n - 1$ firings. Then all places, will always have at least one token, for all the markings that is created. The result of this is, that the all off `Sparse_vector`'s will use twice as much memory as the `Vector_marking`'s they were created from. Such a Petri net is not included, for the same reason the absolute best case is not included either. They do not give a good picture, of how these 2 options do in real world examples. Before we make our conclusion of these results we will take a look at one last case.



(a) (b)

Figure 30: SatelliteMemory Petri net from the *Model Checking Contest 2022* [**model˙check˙contest**], specifically the file call `X01500Y0046.pnml`. The blue dots are for the instance, where the `Vector_markings` are compressed to `Sparse_markings` before they are stored in the `Collection_map`. The orange dots are the results, where the `Vector_markings` were not compressed before they were stored in the `Collection_map`. The visitor `Is_reachable` was used, and the enabled transitions were stored in a `set`.

In Figure 31 we see the best case for this compression technique that I could find. In this case, the `Map_collection` that compressed the `Vector_marking` to `Sparse_markings` saved a significant amount of memory, and it was also faster than the uncompressed version. The is due to, that the `vector`'s that has to be hashed in order to store them in the `Collection_map` were often significantly smaller, than their corresponding uncompressed `vector`'s. Even though the `Collection_map` has to make a pass over the `Vector_marking` that was given as input, in order to compress it to a `Sparse_vector`, it was still faster than hashing it immediately. The result is, that the compressed version were around 30% faster, and a little over 5 times as memory efficient, when comparing the two reachability quires that had to visit the most markings.



(a) (b)

Figure 31: AutonomousCar Petri net from the *Model Checking Contest 2022* [**model˙check˙contest**], specifically the file call `autocar02_b`. The blue dots are for the instance, where the `Vector_markings` are compressed to `Sparse_markings` before they are stored in the `Collection_map`. The orange dots are the results, where the `Vector_markings` were not compressed before they were stored in the `Collection_map`. The visitor `Is_reachable` was used, and the enabled transitions were stored in a `set`.

To summarize the results we have found. In most cases, it does not matter if the `Vector_marking` that is given to the `Collection_map` is compressed to a `Sparse_vector` or not, if we look at the time used. There are outliers as we just saw, but from the testing that have been done beyond these examples, it does not matter if the `Vector_markings` were compressed or not, if we only optimize for time. When we try to optimize for memory, the choice to compress the `Vector_marking` or not can have a big impact. As we expected, it all depends on the markings of the reachability graph. If they are sparse, then we can save both memory and time. If they are not, then it can cost us a bit of both. Since we expect to be working with mostly sparse reachability graphs, the compression from a `Vector_marking` to a `Sparse_vector` will be chosen as the default.

12.1.3 Benchmarking the retrieval of the enabled transition stack

12.1.4 LoLA vs ??

13 Discussion

14 Conclusion

References

[1] Jakob Lykke Andersen. "Analysis of Generative Chemistries". In: (2016). URL: https://ul.qucosa.de/api/qucosa%3A14700/attachment/ATT-0/.

[2] Jakob Lykke Andersen. *MedØlDatschgerl (MØD)*. https://jakobandersen.github.io/mod/. Last accessed 2023-12-30. 2023.

[3] Boost. *Boost C++ Libraries*. http://www.boost.org/. Last accessed 2023-12-30. 2023.

[4] Roberto Bruni and Hernán Melgratti. "Non-sequential Behaviour of Dynamic Nets". In: *Petri Nets and Other Models of Concurrency - ICATPN 2006*. Ed. by Susanna Donatelli and P. S. Thiagarajan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 105–124. ISBN: 978-3-540-34700-2.

[5] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[6] Javier Esparza. "Decidability and complexity of Petri net problems - An introduction". In: *LNCS* 1491 (Sept. 2000). DOI: 10.1007/3-540-65306-6_20.

[7] Gillings MR. Fourment M. "A comparison of common programming languages used in bioinformatics." In: *BMC Bioinformatics* (Feb. 2008). DOI: doi:10.1186/1471-2105-9-82. URL: https://pubmed.ncbi.nlm.nih.gov/18251993/.

[8] Erich Gamma, Richard Helm, and Ralph E. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. 1st ed., Reprint. Addison-Wesley Longman, Amsterdam, 1994. ISBN: 0201633612.

[9] Robert Jan van Glabbeek. "The Individual and Collective Token Interpretations of Petri Nets". In: *CONCUR 2005 – Concurrency Theory*. Ed. by Martín Abadi and Luca de Alfaro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 323–337. ISBN: 978-3-540-31934-4.

[10] Shu-Ming Hsieh, Chiun-Chieh Hsu, and Li-Fu Hsu. "Efficient Method to Perform Isomorphism Testing of Labeled Graphs". In: *Proceedings of the 2006 International Conference on Computational Science and Its Applications - Volume Part V*. ICCSA'06. Glasgow, UK: Springer-Verlag, 2006, pp. 422–431. ISBN: 3540340793. DOI: 10.1007/11751649_46. URL: https://doi.org/10.1007/11751649_46.

[11] T. Murata. "Petri nets: Properties, analysis and applications". In: *Proceedings of the IEEE* 77.4 (1989), pp. 541–580. DOI: 10.1109/5.24143.

[12] Karsten Schmidt. "LoLA: A low level analyser". In: vol. 1825. June 2000, pp. 465–474. ISBN: 978-3-540-67693-5. DOI: 10.1007/3-540-44988-4_27.

[13] Karsten Wolf. "Generating Petri Net State Spaces". In: *Petri Nets and Other Models of Concurrency – ICATPN 2007*. Ed. by Jetty Kleijn and Alex Yakovlev. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 29–42. ISBN: 978-3-540-73094-1.