Planning via Colored Petri Nets

Henrik Ginnerup and Sebastian Lassen

Aalborg University
Department of Computer Science, Aalborg, Denmark
{hginne19, slasse19}@student.aau.dk

Abstract. Both lifted planning tasks and colored Petri nets have the benefit of being much more compact and usable by humans than their grounded and P/T net counterparts. We present a novel method for translating a large subset of lifted planning tasks to colored Petri nets. The correctness of this method is argued, and an implementation of it is made. This implementation is compared to a previous translation from a planning task, grounded using the planning system Fast Downward, to a P/T net. Both translations are run in the Petri net verification tool TAPAAL with the same search strategy. Additionally, both of these are compared to the lama-first configuration of Fast Downward. The Fast Downward configuration outperforms both of the Petri net approaches in terms of how fast it finds a non-optimal plan, however, there are some interesting differences in performance between the two Petri net approaches. While their performance is largely similar, in specific domains, the grounded translation contains much fewer transitions than the unfolded colored Petri net of the lifted translation, yielding a much faster verification time for the grounded transition. This hints at possible optimizations to the unfolding of colored Petri nets, as the grounding process of Fast Downward was able to drastically decrease the size of the task, compared to the unfolding of TAPAAL.

1 Introduction

Classical planning is an area of study concerned with finding a sequence of actions which lead from some initial configuration to a desired configuration [15]. Other types of planning differ, but in classical planning, the environment is fully observable and deterministic, meaning the consequences of actions are known and consistent. As this paper only covers classical planning, it will be referred to as just planning. The other area contended with in this paper, is that of Petri nets [14], also referred to as P/T nets. The typical application of P/T nets is using them to model distributed systems, which can then be analyzed with the many techniques developed for this purpose [13]. Even though the typical use-cases are different, there are some immediately apparent similarities in the way planning tasks are represented and the structure of P/T nets, as well as the semantics of firing a transition in a P/T net and applying an action in planning. Exploring these similarities, and inversely also the differences, is the initial motivation for this paper.

This relationship has been previously noticed and used to solve planning tasks with P/T nets, as Hickmott et al. [10] created a method for translating a grounded planning task to a 1-safe P/T net. In their paper, this translation is used by a heuristic-directed unfolder in order to solve the planning problem. However, other tools for Petri net modelling and verification which also support higher-level representations such as colored Petri nets (CPN) exist and are continually being improved. One such is CPN Tools[16], and more recently TAPAAL [2] has also emerged with support for CPNs [12]. An advantage of CPNs is their high-level modelling capabilities, allowing them to remain comprehensible for much larger, and more complicated representations than P/T nets. The same is true for lifted representations of planning tasks when compared to their grounded counterparts.

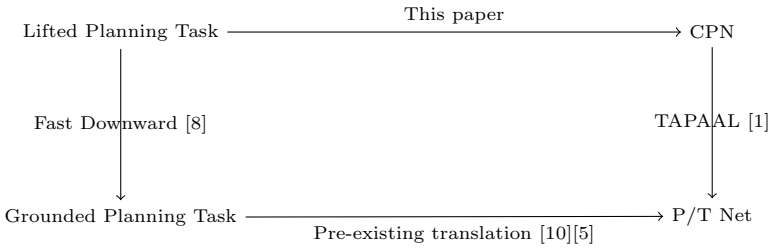


Fig. 1: An overview of the difference between the translation proposed in this paper, and the pre-existing method for translating planning tasks to P/T Nets.

The main contribution of this paper is a method for translating a planning problem in lifted form to a CPN. The difference in approach compared to the previous work is showcased in Figure 1. The pre-existing translation that we use for comparison is an implementation of the method proposed in Hickmott et al. [10], implemented by Daniel Gnad as part of his PhD thesis [5], and later modified by Peter Gjøl and us. This implementation uses the grounding process of the planning tool Fast Downward [8] on the lifted task before translating it. Conversely, our implementation translates the lifted planning task directly to a CPN which is then unfolded and verified using TAPAAL [1].

In Section 2, the syntax and semantics of grounded planning, lifted planning, and CPNs are defined along with examples. Section 3 presents the translation from lifted planning task to CPN and argues for its correctness. In Section 4, the implementation of the translation and other code-based contributions are detailed. In Section 5, the results of experiments which have been run on several planning domains with a planner, the pre-existing P/T net translation, and an implementation of the proposed translation are presented, compared, and discussed. Section 6 presents an extension to the translation which fully supports

type inheritance, and the final Section 7 concludes the paper and proposes future work to improve on the translation and its implementation.

2 Preliminaries

This section defines the syntax and semantics of grounded planning tasks, lifted planning tasks, and colored Petri nets. These are presented along with examples of each formalism and its semantics.

2.1 Grounded planning

The objective of a planning task is finding a sequence of actions which transform the environment from a given initial state to a so-called goal state. An example of a planning task could be how to move containers onto a freight ship one at a time such that they end up stacked in a particular way. Here, a definition for grounded planning is given along with the semantics of how actions are applied to states and what constitutes a valid plan. The definition chosen for the grounded planning task is based on STRIPS [3]. In this definition, states are comprised of propositions typically referred to as facts or atoms, denoting whether a fact is true in that state.

Definition 1. *A grounded planning task is a 4-tuple* $\Pi = \langle F, \mathcal{A}, s_0, G \rangle$ *where:*

1. F *is a finite set of facts,*
2. $\mathcal{A} \subseteq 2^F \times 2^F \times 2^F$ *is a set of actions,*
3. $s_0 \subseteq F$ *is a set of initial facts, and*
4. $G \subseteq F$ *is a set of required goal facts.*

The semantics of a grounded planning task is defined as a transition system that is traversed by applying actions. A state s is a subset of facts $s \subseteq F$. The elements of an action 3-tuple are called preconditions, delete effects, and add effects, respectively. We write $s \xrightarrow{a} s'$ if $\exists a = (pre, del, add) \in \mathcal{A}$ s.t. $pre \subseteq s$ and $s' = (s \setminus del) \cup add$. By this, a sequence of action applications are defined as $s \xrightarrow{a\omega} s'$ if $s \xrightarrow{a} s''$ and $s'' \xrightarrow{\omega} s'$ where $\omega \in \mathcal{A}^*, a \in \mathcal{A}$, and $s \xrightarrow{\epsilon} s$.

By these semantics, a valid plan π for a planning task is a sequence of actions $\omega \in \mathcal{A}^*$ s.t. $s_0 \xrightarrow{\omega} s$ and $G \subseteq s$.

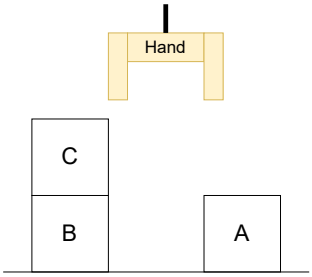


Fig. 2: The initial state.

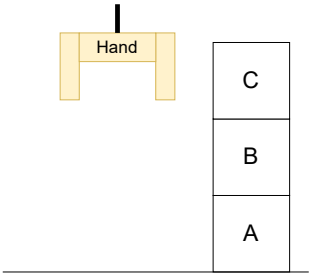


Fig. 3: The goal state.

Grounded planning example - Blocks World A planning task which is reminiscent of the real life example of stacking shipping containers is that of Blocks World [6] [18], which is depicted in Figure 2 and 3. All of the facts and what they represent is described in Figure 4a. Similarly, all the actions are showcased in Figure 4b and the initial state as well as the goal facts are in Figure 4c.
The optimal solution to this planning task is the action sequence:

$$\pi = \texttt{Unstack C B, PutDown C, PickUp B, Stack B A, PickUp C, Stack C B}$$

This plan traverses the following states:

$$\begin{aligned}
&\texttt{\{Clear A, Clear C, On C B, OnTable B, OnTable A, HandEmpty\}} \xrightarrow{\texttt{Unstack C B}} \\
&\texttt{\{Clear A, OnTable B, OnTable A, Holding C, Clear B\}} \xrightarrow{\texttt{PutDown C}} \\
&\texttt{\{Clear A, OnTable B, OnTable A, Clear B, Clear C, OnTable C, HandEmpty\}} \xrightarrow{\texttt{PickUp B}} \\
&\texttt{\{Clear A, OnTable A, Clear C, OnTable C, Holding B\}} \xrightarrow{\texttt{Stack B A}} \\
&\texttt{\{OnTable A, Clear C, OnTable C, Clear B, HandEmpty, On B A\}} \xrightarrow{\texttt{PickUp C}} \\
&\texttt{\{OnTable A, Clear B, On B A, Holding C\}} \xrightarrow{\texttt{Stack C B}} \\
&\texttt{\{OnTable A, On B A, Clear C, HandEmpty, On C B\}} \supseteq \texttt{\{On C B, On B A\}}
\end{aligned}$$

Facts $F =$	**Size**	**Description**
$\{\texttt{Clear}\ x \mid x \in \{A, B, C\}\} \cup$	3	Is the top of block x clear?
$\{\texttt{On}\ x\ y \mid x, y \in \{A, B, C\}\} \cup$	9	Is block x on block y?
$\{\texttt{OnTable}\ x \mid x \in \{A, B, C\}\} \cup$	3	Is block x on the table?
$\{\texttt{HandEmpty}\} \cup$	1	Is the hand empty?
$\{\texttt{Holding}\ x \mid x \in \{A, B, C\}\}$	3	Is the hand holding block x?

(a) The facts in the grounded Blocks World planning task

	Actions $\mathcal{A} =$	**Size**	**Description**
PickUp x**:**	$\{(pre, del, add) \mid x \in \{A, B, C\}\}$ where $pre = \{\texttt{Clear}\ x, \texttt{OnTable}\ x, \texttt{HandEmpty}\}$, $del = \{\texttt{Clear}\ x, \texttt{OnTable}\ x, \texttt{HandEmpty}\}$, $add = \{\texttt{Holding x}\}$	3	Pick block x up from the table.
PutDown x**:**	$\{(pre, del, add) \mid x \in \{A, B, C\}\}$ where $pre = \{\texttt{Holding}\ x\}$, $del = \{\texttt{Holding}\ x\}$, $add = \{\texttt{Clear}\ x, \texttt{OnTable}\ x, \texttt{HandEmpty}\}$	3	Put block x on the table.
Stack $x\ y$**:**	$\{(pre, del, add) \mid x, y \in \{A, B, C\}\}$ where $pre = \{\texttt{Holding}\ x, \texttt{Clear}\ y\}$, $del = \{\texttt{Holding}\ x, \texttt{Clear}\ y\}$, $add = \{\texttt{Clear}\ x, \texttt{HandEmpty}, \texttt{On}\ x\ y\}$	9	Stack block x on block y.
Unstack $x\ y$**:**	$\{(pre, del, add) \mid x, y \in \{A, B, C\}\}$ where $pre = \{\texttt{On}\ x\ y, \texttt{Clear}\ x, \texttt{HandEmpty}\}$, $del = \{\texttt{On}\ x\ y, \texttt{Clear}\ x, \texttt{HandEmpty}\}$, $add = \{\texttt{Holding}\ x, \texttt{Clear}\ y\}$	9	Pick up block x from block y.

(b) The actions in the grounded Blocks World planning task. Notice that the actions $\mathcal{A}$ is a set of triples and every element of the triples is a subset of the facts F.

Initial facts $s_0 =$	**Size**	**Description**
{Clear A, Clear C, On C B, OnTable B, OnTable A, HandEmpty}	6	See Figure 2
Goal facts $G =$	**Size**	**Description**
{On C B, On B A}	2	See Figure 3

(c) The start facts and the goal facts in the grounded Blocks World planning task.

Fig. 4: The full grounded Blocks World planning task $\Pi = \langle F, \mathcal{A}, s_0, G \rangle$ displayed across three tables. The "Size" column denotes how many elements the set in that row has.

2.2 Lifted planning

Since the amount of facts and actions needed to describe a planning task in a grounded representation increases rapidly wrt. the arity of these and the number of objects, it is useful to have another format for describing planning tasks, which is both more readable and writable. For the lifted planning definition, we have split the concept of a planning task into a domain and a problem part. The domain describes the general part of a task in terms of predicates and action schema. The problem then specifies the instance of the domain that is to be solved by stating what the initial state, the goal facts, and the concrete objects are.

First, we define the syntax of a lifted planning domain, promptly followed by a description of how a grounded planning task can be constructed by binding the predicates and action schema to specific objects, thereby creating facts and actions. This is done because the semantics needed to solve a planning task are given in grounded form as defined in Section 2.1.

Definition 2. *A lifted planning domain is a 6-tuple* $\Pi = \langle (\mathcal{T}, \leq), \mathcal{O}, X, \mathcal{P}, \mathcal{P}_\mathcal{X}, \mathcal{A}_s \rangle$ *where:*

1. $(\mathcal{T}, \leq)$ *is a finite preordered set defining a type hierarchy,*
2. $\mathcal{O}$ *is a finite set of objects, each with a type indicated by* $typeOf : \mathcal{O} \to \mathcal{T}$*,*
3. X *is a finite set of parameters, each with a type indicated by* $typeOf : X \to \mathcal{T}$*,*
4. $\mathcal{P}$ *is a finite set of first-order predicates, each with an n-tuple of types indicated by the signature* $sig : \mathcal{P} \to \mathcal{T}^*$*,*
5. $\mathcal{P}_\mathcal{X}$ *is the set of all parameterized predicates* $p(x_1, ..., x_n)$ *s.t.* $p \in \mathcal{P}$ *and* $\forall i \in (1...n).typeOf(x_i) \leq \tau_i$ *where* $sig(p) = (\tau_1, ..., \tau_n)$*, and*
6. $\mathcal{A}_s \subseteq 2^{\mathcal{P}_\mathcal{X}} \times 2^{\mathcal{P}_\mathcal{X}} \times 2^{\mathcal{P}_\mathcal{X}}$ *is a finite set of action schema.*

Now we construct the grounded planning 4-tuple $\Pi = \langle F, \mathcal{A}, s_0, G \rangle$ from the lifted planning domain. Since the initial state s_0 and the goal facts G are always given in grounded form, all we need to do is ground the predicates and action schema to produce the set of facts F and actions $\mathcal{A}$. Here, grounding means replacing each parameter by a concrete object. Let $\mathcal{B}$ be a set of bindings from parameters to objects, written as $b : X \to \mathcal{O}$. A binding is valid if the type of each parameter is compatible with the type of the object being bound to it. The set of valid bindings is $\mathcal{B}_{Val} \stackrel{\text{def}}{=} \{b \in \mathcal{B} \mid typeOf(o) \leq typeOf(x)\}$. Furthermore, we define the notation $p[b]$ to denote that the parameters of a predicate p are bound $p[b] \stackrel{\text{def}}{=} p(b(x_1), ..., b(x_n))$. The set of facts can be constructed by replacing each parameter of every predicate by valid objects $F = \{p[b] \mid p \in \mathcal{P}_\mathcal{X}, b \in \mathcal{B}_{Val}\}$. Similarly, the set of actions can be constructed by replacing each parameter of every predicate in each of its elements by a valid object $\mathcal{A} = \{(pre[b], del[b], add[b]) \mid (pre, del, add) \in \mathcal{A}_s, b \in \mathcal{B}_{Val}\}$.

Lifted planning example 1 - Blocks World To showcase the difference, the same planning task as in the grounded example, depicted in Figure 2, is written as a lifted planning task here. To represent the domain, we start by constructing the 6-tuple $\Pi = \langle(\mathcal{T}, \leq), \mathcal{O}, X, \mathcal{P}, \mathcal{P}_\mathcal{X}, \mathcal{A}_s\rangle$. There is only one type in the Blocks World domain, which we will call "block". This means that every parameter and object will always have the same type. The set of objects are the three blocks $\mathcal{O} = \{\texttt{A, B, C}\}$. The set of parameters has 2 elements as none of the predicates have an arity higher than this and only one type exists $X = \{x, y\}$. There are five predicates $\mathcal{P} = \{\texttt{Clear, OnTable, Holding, On, HandEmpty}\}$ with the first 3 having the signatures $sig(\texttt{Clear}), sig(\texttt{OnTable}), sig(\texttt{Holding}) = block$, $sig(\texttt{On}) = (block, block)$, and $sig(\texttt{HandEmpty}) = \epsilon$. The parameterized predicates $\mathcal{P}_\mathcal{X}$ are shown in Figure 5a and the action schema $\mathcal{A}_s$ are likewise shown in Figure 5b.

$\mathcal{P}_\mathcal{X} =$	**Description**
$\{\texttt{Clear } x\}\cup$	Is the top of block x clear?
$\{\texttt{On } x\ y\}\cup$	Is block x on block y?
$\{\texttt{OnTable } x\}\cup$	Is block x on the table?
$\{\texttt{HandEmpty}\}\cup$	Is the hand empty?
$\{\texttt{Holding } x\}$	Is the hand holding block x?

(a) The parameterized predicates in the lifted Blocks World planning domain

$\mathcal{A}_s =$	**Description**
PickUp x: $\{(pre, del, add)\}$ where $pre = \{\texttt{Clear } x, \texttt{OnTable } x, \texttt{HandEmpty}\}$, $del = \{\texttt{Clear } x, \texttt{OnTable } x, \texttt{HandEmpty}\}$, $add = \{\texttt{Holding x}\}$	Pick block x up from the table.
PutDown x: $\{(pre, del, add)\}$ where $pre = \{\texttt{Holding } x\}$, $del = \{\texttt{Holding } x\}$, $add = \{\texttt{Clear } x, \texttt{OnTable } x, \texttt{HandEmpty}\}$	Put block x on the table.
Stack x y: $\{(pre, del, add)\}$ where $pre = \{\texttt{Holding } x, \texttt{Clear } y\}$, $del = \{\texttt{Holding } x, \texttt{Clear } y\}$, $add = \{\texttt{Clear } x, \texttt{HandEmpty}, \texttt{On } x\ y\}$	Stack block x on block y.
Unstack x y: $\{(pre, del, add)\}$ where $pre = \{\texttt{On } x\ y, \texttt{Clear } x, \texttt{HandEmpty}\}$, $del = \{\texttt{On } x\ y, \texttt{Clear } x, \texttt{HandEmpty}\}$, $add = \{\texttt{Holding } x, \texttt{Clear } y\}$	Pick up block x from block y.

(b) The action schema in the lifted Blocks World domain.

Fig. 5: The parameterized predicates$\mathcal{P}_\mathcal{X}$ and $\mathcal{A}_s$ of the lifted Blocks World domain displayed across two tables.

The process of binding each parameter of every $p(x_1, ..., x_n) \in \mathcal{P}_{\mathcal{X}}$ and each element of every $a_s \in \mathcal{A}_s$ would be exactly akin to the set-building seen in Figure 4a and 4b, where each parameter is replaced by elements from the set of objects. Since the initial facts and goal facts are always given in grounded form, these are identical to those in Figure 4c.

Lifted planning example 2 - Shapes World Since the Blocks World example is uninteresting in terms of types, these will be demonstrated in more detail with this example. In this planning domain, which we call Shapes World, the general goal, predicates, and actions available remain the same as in Blocks World. However, there are now 3 types of shapes instead of only blocks. The initial state of this Shapes World task is shown in Figure 6 and the goal state in Figure 7. The preordered set of types $(\mathcal{T}, \leq)$ is illustrated in Figure 8. Here, $square \leq rectangle \leq shape$ and $triangle \leq shape$. As can be seen in the aforementioned figures, the types of each object are $typeOf(A) = square$, $typeOf(B) = triangle$, and $typeOf(C) = rectangle$.

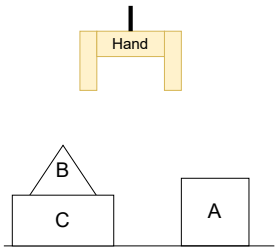


Fig. 6: The initial state.

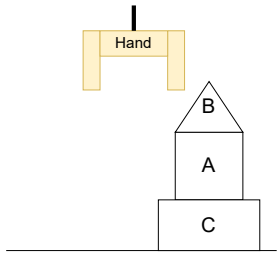


Fig. 7: The goal state.

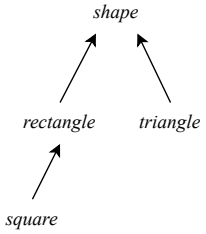


Fig. 8: Type hierarchy.

An example of how the types are used, is that the predicate `On` in Shapes World has the signature $sig\ (\texttt{On}) = (shape, rectangle)$, as we can not stack anything on top of a triangle. By the type conditions presented in Definition 2 for the set of parameterized predicates, how these are used in the action schema, and later grounding, this results in only objects with the types *rectangle* and *square* being able to have another shape on top of them. Since the first parameter has the *shape* type, it is still possible to stack triangles on non-triangle shapes so the goal state remains reachable.

2.3 Colored Petri Net

This section defines colored Petri nets with set-based semantics, as opposed to other definitions as for example in "Coloured Petri Nets and the Invariant-Method" by Kurt Jensen[11] or "Improvements in Unfolding of Colored Petri Nets" by Bilgram et al.[1], where the configuration, or marking, is based on multisets of colors. The reason for this set-based adaptation is to more directly represent the statespace of a planning problem. For instance in planning, then if fact f is in the state s and an action is applied which adds the same f to the state, then f is still just in the state and not doubly in the state. If some action then removes f from the state, then it is immediately removed no matter how many times f was previously set to be in the state.

Color Let $\mathbb{C}_{atom}$ and $\mathbb{C}_{T_{atom}}$ be the disjoint sets of all *atomic colors* and *atomic color types* respectively. Atomic colors and atomic color types are associated many-to-one where each atomic color is associated to a distinct type.
The function $typeOf : \mathbb{C}_{atom} \rightarrow \mathbb{C}_{T_{atom}}$ returns the type of a value, and inversely $typeValues : \mathbb{C}_{T_{atom}} \rightarrow 2^{\mathbb{C}_{atom}}$ returns all values of a given type, s.t. given $c \in \mathbb{C}_{atom}$ and $t \in \mathbb{C}_{T_{atom}}$ then $typeOf(c) = t$ iff $c \in typeValues(t)$.

Let $\mathbb{C}$ and $\mathbb{C}_T$ be the disjoint sets of all colors and color types respectively, where a color is a non-empty tuple s.t. $\mathbb{C} = {\mathbb{C}_{atom}}^+$ and likewise for types where $\mathbb{C}_T = {\mathbb{C}_{T_{atom}}}^+$. The atomic colors and types can be considered 1-tuples which implies $\mathbb{C}_{atom} \subseteq \mathbb{C}$ and $\mathbb{C}_{T_{atom}} \subseteq \mathbb{C}_T$. Similar to the atomic version, colors are each associated to a distinct color type. To get the type of a given color $(c_1, ..., c_i) \in \mathbb{C}$, the function $typeOf : \mathbb{C} \rightarrow \mathbb{C}_T$ is defined as:

$$typeOf((c_1, ..., c_i)) = (typeOf(c_1), ..., typeOf(c_i))$$

To then get all possible values of a color type $(ct_1, ..., ct_i) \in \mathbb{C}_T$ the function $typeValues : \mathbb{C}_T \rightarrow 2^{\mathbb{C}}$ is defined as:

$$typeValues((ct_1, ..., ct_i)) = typeValues(ct_1) \times ... \times typeValues(ct_i)$$

Variables Given a finite set of atomic color types $\mathbb{C}_{T_{atom}}$, let $\mathbb{V}$ be a finite set of *variables*, where each variable is associated to a type through the function $typeOf : \mathbb{V} \rightarrow \mathbb{C}_{T_{atom}}$.

Atomic Value Atomic values are either a single variable or an atomic color of a given type τ, and is represented through the following syntax:

$$atom^\tau ::= var^\tau \mid color^\tau$$

where var is a variable, $color$ is an atomic color and τ is the type of the expression s.t. $typeOf(var) = \tau$ or $typeOf(color) = \tau$.

Arc Expression Let $\mathbb{AE}^{\tau}$ be a set of *arc expressions* of color type τ. An arc expression ae^{τ} of type τ is represented with the following syntax:

$$ae^{\tau} ::= (atom^{\tau_1}, ..., atom^{\tau_k}) \mid ae_1^{\tau} + ae_2^{\tau} \mid \epsilon$$

where $\tau = \tau_1 \times ... \times \tau_k$ and ϵ is the empty string.

Guard Expression Let $\mathbb{GE}$ be a set of *guard expressions* for expressing boolean logic on atomic values. Guard expressions are represented through the syntax:

$$ge ::= atom_1^{\tau} = atom_2^{\tau} \mid true \mid false \mid \neg ge \mid ge_1 \wedge ge_2 \mid ge_1 \vee ge_2$$

In guard expressions only atom pairs, such as in $atom_1^{\tau} = atom_2^{\tau}$ must match in type, where a different pair may match on a different type, unlike arc expressions where all subexpressions must to be of a single type.

Colored Petri Net

Definition 3. *A colored Petri net with set semantics, or CPN, is a 5-tuple* $(\mathbb{P}, \mathbb{T}, \mathbb{V}, Arcs, Guards)$ *where*

1. $\mathbb{P}$ *is a finite set of places, where each place is associated to a distinct color type through the function* $typeOf : \mathbb{P} \to \mathbb{C}_T$,
2. $\mathbb{T}$ *is finite set of transitions, s.t.* $\mathbb{P} \cap \mathbb{T} = \emptyset$,
3. $\mathbb{V}$ *is a finite set of variables,*
4. $Arcs : (\mathbb{P} \times \mathbb{T}) \cup (\mathbb{T} \times \mathbb{P}) \to \mathbb{AE}$ *connects any place and transition with an arc expression. Given* $p \in \mathbb{P}$ *and* $t \in \mathbb{T}$*, then Arcs is type-restricted s.t.* $Arcs(p,t) \in \mathbb{AE}^{typeOf(p)}$ *and* $Arcs(t,p) \in \mathbb{AE}^{typeOf(p)}$*. The arc expression of any disconnected pair is* ϵ.
5. $Guards : \mathbb{T} \to \mathbb{GE}$ *is a function to get the guard expression of a transition.*

Marking The configuration of a Petri net is called a *marking*. For colored Petri nets with set based semantics, let a marking be any function $M : \mathbb{P} \to 2^{\mathbb{C}}$ that for each place returns a set of colors of the same color type as the place, s.t. given a place $p \in \mathbb{P}$ then $M(p) \subseteq typeValues(typeOf(p))$.

Colored Petri net example - Blocks World As an early introduction to the general translation in Section 3, this example models the same problem as the planning example for Blocks World. For this, the colors are defined in Figure 1.

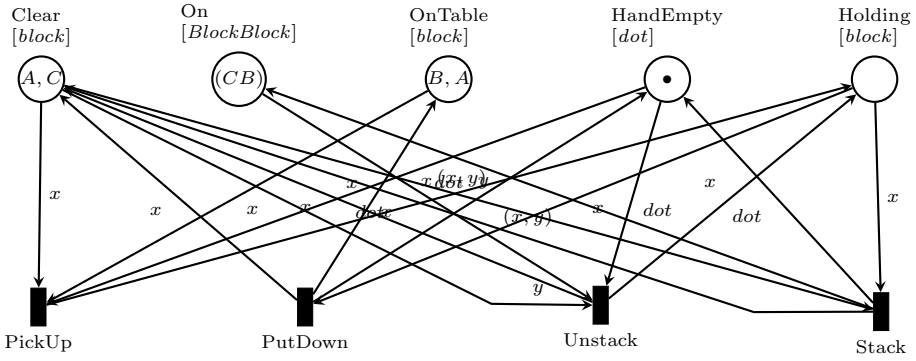
$\tau \in \mathbb{C}_T$	$typeValues(\tau)$	Purpose
dot	$\{token\}$	For predicates with no parameters, such as $HandEmpty$
$block$	$\{A, B, C\}$	Represents blocks.
$(block, block)$	$typeValues(block)^2$	For predicates with pairs of blocks, such as On.

Table 1: The colors used in the Blocksworld CPN.

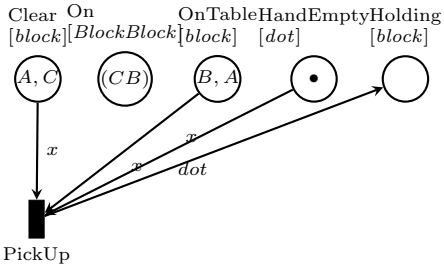
And then the CPN $BlocksCpn = (\mathbb{P}, \mathbb{T}, \mathbb{V}, Arcs, Guards)$ is first defined below, and then visually represented in Figure 9.

1. $\mathbb{P} = \{On_{(block,block)}, Clear_{block}, HandEmpty_{dot}, OnTable_{block}\}$,
   where $place_\tau$ denotes $typeOf(place) = \tau$,
2. $\mathbb{T} = \{PickUp, PutDown, Stack, Unstack\}$,
3. $\mathbb{V} = \{x_{block}, y_{block}\}$,
4. $Arcs$ is written here for only $PickUp$:
   $Arcs(Clear, PickUp) = \text{`}x\text{`}$
   $Arcs(OnTable, PickUp) = \text{`}x\text{`}$
   $Arcs(HandEmpty, PickUp) = \text{`}token\text{`}$
   $Arcs(PickUp, Holding) = \text{`}x\text{`}$

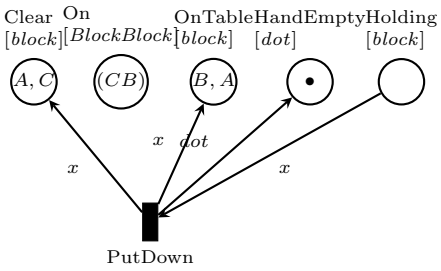
   The rest of the arcs can be seen in the visual representation in Figure 9.
5. In this example all guards return $true$.



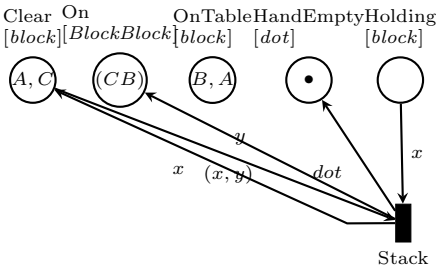
(a) The entire visual representation of *BlocksCpn*.



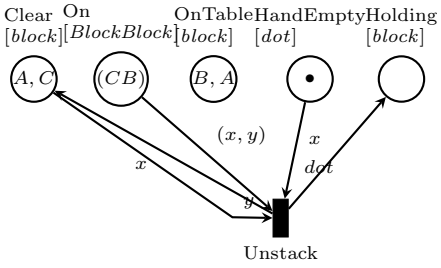
(b) Shows only arcs for *PickUp*.



(c) Shows only arcs for *PutDown*.



(d) Shows only arcs for *Stack*.



(e) Shows only arcs for *UnStack*.

Fig. 9: Figure 9a shows the entire visual representation of *BlocksCpn*, where circles are places, rectangles are transitions and arrows are arcs, where for instance $Arcs(Clear, PickUp) =$ 'x' is shown by the arrow from *Clear* to *PickUp* with x written next to it. Admittedly it is difficult to read the arcs in Figure 9a, and therefore each transition t here has its own subfigure that shows only the arcs connected to or from t. Lastly, the marking is represented by the dots on each place, with the exact colors written under the name of the place, for instance $M(Clear) = \{A, C\}$.

2.4 Petri Net Semantics

This section goes through how a colored Petri net with set semantics transitions from one marking to another. Let $\mathcal{N} = (\mathbb{P}, \mathbb{T}, \mathbb{V}, \mathit{Arcs}, \mathit{Guards})$ be a constant CPN for the rest of this section.

Bindings Let a binding be any function $b : \mathbb{V} \to \mathbb{C}_{atom}$ that takes a variable and returns a concrete, atomic color of the same type, s.t. given a variable $v \in \mathbb{V}$ then $\mathit{typeOf}(b(v)) = \mathit{typeOf}(v)$.

For notation when given a binding b and either variable v or color c, then $v[b] = b(v)$ or $c[b] = c$, which then extends atomic values, as they consist of either a variable or an atomic color.

Arc Expression Evaluation Given a binding b and an arc expression ae, let $ae[b]$ be the resulting set of colors from evaluating ae with b according to the semantics in Figure 10.

$$[COLOR] \quad \frac{}{b \vdash (atom_1, ..., atom_k) \rightarrow_{ae} (c_1, ..., c_k)} \quad \text{where } c_1 = atom_1[b],\ ...,\ c_k = atom_k[b]$$

$$[UNION] \quad \frac{b \vdash ae_1 \rightarrow_{ae} c_1 \qquad b \vdash ae_2 \rightarrow_{ae} c_2}{b \vdash ae_1 + ae_2 \rightarrow_{ae} c_1 \cup c_2}$$

$$[EMPTY] \quad \frac{}{b \vdash \epsilon \rightarrow_{ae} \emptyset}$$

Fig. 10: Semantic rules for arc expressions.

Guard Expression Evaluations Given a binding b and a guard expression ge, let $ge[b]$ be the resulting boolean value $true$ or $false$ from evaluating ge with b according to the semantics in Figure 11.

$$[EQUAL] \quad \frac{}{b \vdash atom_1 \ = \ atom_2 \rightarrow_{ge} c_1 \ = \ c_2} \quad \text{where } c_1 = atom_1[b],\ c_2 = atom_2[b]$$

$$[TRUE] \quad \frac{}{b \vdash true \rightarrow_{ge} true}$$

$$[FALSE] \quad \frac{}{b \vdash false \rightarrow_{ge} false}$$

$$[NOT] \quad \frac{b \vdash ge \rightarrow_{ge} bool}{b \vdash \ \neg ge \rightarrow_{ge} \neg bool}$$

$$[AND] \quad \frac{b \vdash ge_1 \rightarrow_{ge} bool_1 \qquad b \vdash ge_2 \rightarrow_{ge} bool_2}{b \vdash ge_1 \ \wedge \ ge_2 \rightarrow_{ge} bool_1 \wedge bool_2}$$

$$[OR] \quad \frac{b \vdash ge_1 \rightarrow_{ge} bool_1 \qquad b \vdash ge_2 \rightarrow_{ge} bool_2}{b \vdash ge_1 \ \vee \ ge_2 \rightarrow_{ge} bool_1 \vee bool_2}$$

Fig. 11: Semantic rules for boolean guard expressions.

Transition Firing A transition t under binding b is enabled for marking M, if

1. $Guards(t)[b] = true$ and
2. $Arcs(p,t)[b] \subseteq M(p)$ for all $p \in \mathbb{P}$.

An enabled transition can be fired, written $M \xrightarrow{t,b} M'$, producing a marking M' defined as $M'(p) = (M(p) \setminus Arcs(p,t)[b]) \cup Arcs(t,p)[b]$ for all $p \in \mathbb{P}$.

Additionally, a sequence of transition firings is called a trace, which is defined as $M \xrightarrow{(t,b)w} M'$ if $M \xrightarrow{(t,b)} M''$ and $M'' \xrightarrow{w} M'$, where $w \in (\mathbb{T} \times \mathbb{B})^*$, $(t,b) \in w$, and $M \xrightarrow{\epsilon} M$.

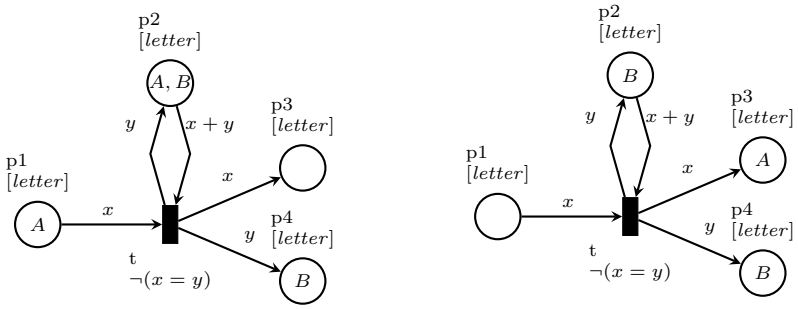
Transition Firing - Example For a small new example CPN, let $\mathbb{C}_{T_{atom}} = \{letter\}$ and $\mathbb{C}_{atom} = typeValues(letter) = \{A, B\}$. The CPN can be seen in Figure 12, along with the effect of firing its transition t under binding $b = \{x \Rightarrow A, y \Rightarrow B\}$, where the arc expressions get evaluated:

$$\begin{aligned} \text{`}x\text{`}[b] &= \{A\}, \\ \text{`}y\text{`}[b] &= \{B\} \text{ and} \\ \text{`}x + y\text{`}[b] &= \{A\} \cup \{B\} = \{A, B\} \end{aligned}$$

Firing t then updates the marking of each place:

$$\begin{aligned} M'(p1) &= (M(p1) \setminus Arcs(p1,t)[b]) \cup Arcs(t,p1)[b] = (\ \{A\}\ \setminus\ \{A\}\ ) \cup\ \emptyset \\ M'(p2) &= (M(p2) \setminus Arcs(p2,t)[b]) \cup Arcs(t,p2)[b] = (\ \{A,B\} \setminus \{A,B\}\ ) \cup \{B\} \\ M'(p3) &= (M(p3) \setminus Arcs(p3,t)[b]) \cup Arcs(t,p3)[b] = (\ \emptyset\ \setminus\ \emptyset\ ) \cup \{A\} \\ M'(p4) &= (M(p4) \setminus Arcs(p3,t)[b]) \cup Arcs(t,p3)[b] = (\ \{B\}\ \setminus\ \emptyset\ ) \cup \{B\} \end{aligned}$$

The only valid binding in this example is $b = \{x \Rightarrow A, y \Rightarrow B\}$, as both $\{x \Rightarrow A, y \Rightarrow A\}$ and $\{x \Rightarrow B, y \Rightarrow B\}$ would be blocked by the guard expression $\neg(x = y)$ and $b = \{x \Rightarrow B, y \Rightarrow A\}$ would mean $Arcs(p1,t)[b] = \{B\}$, which is not a subset of $M(p1)$.



(a) Shows M before firing t.

(b) Shows M' from $M \xrightarrow{t,b} M'$.

Fig. 12: Fired t with binding $b = \{x \Rightarrow a, y \Rightarrow b\}$

3 Translation

This section presents the translation from a subset of lifted planning tasks to a colored Petri net as it is defined in Section 2.3. This subset is lifted planning tasks as defined in 2.2, but without type inheritance and action schema which contain a specific relationship detailed in Section 3.1. This is the translation which has been implemented and tested. An extension which supports type inheritance is presented in Section 6. This section includes the syntactic translation, the translation of a given planning state to a CPN marking, and an argument of correctness in the form of a proof sketch.

3.1 Planning Domain to Colored Petri Net

We define a constant lifted planning domain for the translation:

$$\mathit{\Pi} = \langle (\mathcal{T}, \leq), \mathcal{O}, X, \mathcal{P}, \mathcal{P}_{\mathcal{X}}, \mathcal{A}_s \rangle$$

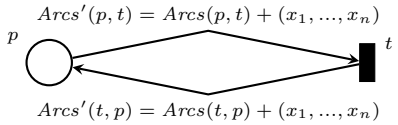
Let $\mathbb{C}_{T_{atom}} = \mathcal{T}$ and $\mathbb{C}_{atom} = \mathcal{O}$, meaning $\mathit{typeOf} : \mathbb{C}_{atom} \to \mathbb{C}_{T_{atom}}$ always returns the same as $\mathit{typeOf} : \mathcal{O} \to \mathcal{T}$.

The translated CPN $\mathcal{N} = (\mathbb{P}, \mathbb{T}, \mathbb{V}, \mathit{Arcs}, \mathit{Guards})$ is then constructed as follows:

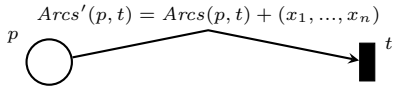
1. $\mathbb{P} = \mathcal{P}$, and for any place, $\mathit{typeOf} : \mathbb{P} \to \mathbb{C}_T$ returns the same as $\mathit{sig} : \mathcal{P} \to \mathcal{T}^*$,
2. $\mathbb{T} = \mathcal{A}_s$,
3. $\mathbb{V} = X$, where $\mathit{typeOf} : \mathbb{V} \to \mathbb{C}_{T_{atom}}$ is already defined via $\mathit{typeOf} : X \to \mathcal{T}$,
4. Arcs are constructed according to the procedure below, and
5. $\mathit{Guards}(t) =$ 'true' for every $t \in \mathbb{T}$.

Arc Expressions The function Arcs is initiated to ϵ for each $p \in \mathbb{P}$ and $t \in \mathbb{T}$, s.t. $\mathit{Arcs}(p,t) = \epsilon$ and $\mathit{Arcs}(t,p) = \epsilon$. Arcs is then updated as described by the following procedure. For each $a_s = (\mathit{pre}, \mathit{del}, \mathit{add}) \in \mathcal{A}_s$, every $p_x = p(x_1, ..., x_n) \in \mathit{pre} \cup \mathit{del} \cup \mathit{add}$ belongs to one of the following cases:

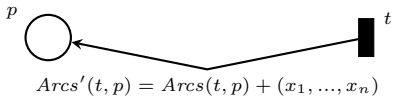
1. The require-relationship. If $(p_x \in \mathit{pre}) \wedge ((p_x \notin \mathit{del}) \vee (p_x \in \mathit{del} \cap \mathit{add}))$, then update Arcs to be the Arcs' shown in:



2. The consume-relationship. If $(p_x \in \mathit{pre}) \wedge (p_x \in \mathit{del}) \wedge (p_x \notin \mathit{add})$, then update Arcs to be the Arcs' shown in:



3. The add-relationship. If $(p_x \notin pre) \wedge (p_x \notin del) \wedge (p_x \in add)$, then update $Arcs$ to be the $Arcs'$ shown in:



4. The delete-relationship occurs if $(p_x \notin pre) \wedge (p_x \in del) \wedge (p_x \notin add)$. This relationship is not currently supported by the translation.

3.2 Planning State to CPN Marking

Let $s \subseteq F$ be a state of the planning task. Each individual fact $f \in s$ consists of a predicate and objects, $f = pred(o_1, ...o_n)$. As $\mathbb{P} = \mathcal{P}$ and $\mathbb{C}_{atom} = \mathcal{O}$, then a fact can also be represented as a place and atomic colors. This leads to the function $translate : 2^F \rightarrow (\mathbb{P} \rightarrow 2^{\mathbb{C}})$.

$$translate(s)(p) = \{(c_1, ..., c_n) \mid p(c_1, ..., c_n) \in s\}$$

3.3 Finding a Plan with CPNs

The function $translate : (\mathbb{T} \times (\mathbb{V} \times \mathbb{C}_{atom}))^* \rightarrow (\mathcal{A}_s \times (X \times \mathcal{O}))^*$ takes a CPN trace and returns a plan. It does this by taking each transition in the trace and mapping it to the corresponding action schema, and each CPN binding (v, c) is mapped to the corresponding planning binding (x, o). By the CPN translation, $\mathbb{T} = \mathcal{A}_s$, $\mathbb{V} = X$ and $\mathbb{C}_{atom} = \mathcal{O}$, so the translation from a trace to a plan is the identity function.

Theorem 1. *Given a lifted planning task Π, initial state s_0, and goal condition G, then there exists a sequence of actions $s_0 \xrightarrow{w} s' \supseteq G$ iff for the translated CPN there exists a sequence of transition firings $translate(s_0) \xrightarrow{w} M' \supseteq translate(G)$.*

3.4 Proof Sketch

To prove Theorem 1, it must be individually shown that a plan implies a trace, and then that a trace implies a plan. How to prove the implication of a trace is sketched below, and proving the implication of a plan should be possible using the same method.

If there is a plan, there is a corresponding trace Proof by contradiction. Assume that there exists a plan with no corresponding trace, then there is some lifted planning task Π with state s, where the action can be applied, but the corresponding transition cannot fire, or visa versa. Alternative the state can transition to a next state, where different actions or transitions become available for the state or marking.

Recall from Section 2.2 that an action is equivalent to an action schema in which every parameterized predicate of each element is bound, written as $a_s[b] = (pre[b], del[b], add[b])$.

Given an action schema a_s, binding b, and $M = translate(s)$, then one of the following conditions must hold:

1. $s \xrightarrow{a_s[b]} s'$ is valid, but $M \xrightarrow{a_s,b} M'$ is invalid,
2. $s \xrightarrow{a_s[b]} s'$ is invalid, but $M \xrightarrow{a_s,b} M'$ is valid,
3. $s \xrightarrow{a_s[b]} s'$ and $M \xrightarrow{a_s,b} M'$, but $translate(s') \neq M'$.

For the first condition to hold, $s \xrightarrow{a_s[b]} s'$ must be valid, meaning $pre(a_s[b]) \subseteq s$, and then there are two ways for $translate(s) \xrightarrow{a_s,b} M'$ to be invalid:

1. $Guards(a_s)[b] = false$, which will never be the case, because every guard expression in the translation is $true$,
2. $\exists p \in \mathbb{P}.(Arcs(p, a_s)[b] \not\subseteq M(p))$, which will never hold, because the preconditions of a_s are translated s.t. if the preconditions of a_s are a subset of s, then $Arcs(p, a_s)[b] \subseteq translate(s)(p)$ for every predicate p in the precondition of a_s.

For the second condition to hold, $s \xrightarrow{a_s[b]} s'$ must be invalid, meaning $pre(a_s[b]) \not\subseteq s$, and then there are two conditions for $translate(s) \xrightarrow{a_s,b} M'$ to be valid:

1. $Guards(a_s)[b] = true$, which trivially holds by the translation,
2. $\forall p \in \mathbb{P}.(Arcs(p, a_s)[b] \subseteq M(p))$,
   which will never hold, because the preconditions of a_s are translated s.t. if the preconditions of a_s are not a subset of s, then $Arcs(p, a_s)[b] \not\subseteq translate(s)(p)$ for every predicate p in the precondition of a_s.

The third condition can be shown to not hold by deriving the general case of what happens when $a_s[b]$ is applied to state s, and then what happens to the same state s if it had instead first been translated to a marking before the transition corresponding to a_s was fired. For no contradiction to occur, it should be possible for one of these to result in a state s' for which $translate(s') \neq M'$.

As a draft of the general case when applying $a_s[b]$:
$s \xrightarrow{a_s[b]} s'$ where:

$$s' = (s \setminus del[b]) \cup add[b]$$

$$(pre[b], del[b], add[b]) = a_s[b]$$

Then every $p_X \in pre[b] \cup del[b] \cup add[b]$ belongs to one of the following cases:

1. The require-relationship. If $(p_x[b] \in pre[b]) \wedge ((p_x[b] \notin del[b]) \vee (p_x[b] \in del[b] \cap add[b]))$,
   then $p_X[b] \in s'$.
2. The consume-relationship. If $(p_x[b] \in pre[b]) \wedge (p_x[b] \in del[b]) \wedge (p_x[b] \notin add[b])$,
   then $p_X[b] \notin s'$.
3. The add-relationship. If $(p_x[b] \notin pre[b]) \wedge (p_x[b] \notin del[b]) \wedge (p_x[b] \in add[b])$,
   then $p_X[b] \in s'$.
4. The delete-relationship. If $(p_x[b] \notin pre[b]) \wedge (p_x[b] \in del[b]) \wedge (p_x[b] \notin add[b])$,
   then this theorem doesn't apply, as this case is explicitly out of scope for the translation.

It should then be shown algebraically that first translating state s, and then firing $a_s[b]$ as a transition will yield exactly the same marking, as the marking achieved when $a_s[b]$ was applied *before* translating.

None of the three conditions hold, so we reach a contradiction. □

4 Implementation

This section highlights the significant implementation details. These concern mostly the colored translation, which is the focus of this paper, but also some meaningful modifications to the VerifyPN engine used in TAPAAL, and some modifications on the format of the output of the pre-existing grounded translation.

4.1 Colored Translation

The translation from Section 3 is implemented in Python 3.9.2 and can be found on Github[1]. It takes a planning domain and a task written in the Planning Domain Definition Language (PDDL) and returns a CPN and query in Petri Net Markup Language (PNML). The output of the translation has been tested in TAPAAL for various planning tasks of multiple domains. Furthermore, we have also modified the engine of TAPAAL, which is called VerifyPN. We have made three meaningful changes to VerifyPN:

1. The set-based semantics of the CPN markings is implemented through a single if-statement preventing the multi-set marking from exceeding a weight of 1,[2] which goes to show how closely the semantics used in this paper resemble typical CPNs.
2. The experiments are conducted on a development version of VerifyPN, which did not yet provide the bindings used when a transition is fired, which is needed to recover the colored trace from the trace generated by the unfolded CPN. This has been implemented by printing every binding during the unfolding process in the modified VerifyPN[3].
3. Variables used in different contexts can have the same name in a PDDL planning task, but this is not the case in TAPAAL where their scope is global. Therefore, the name of each variable had to be modified in a way such that it is unique and can be reverted to its original name.

4.2 Grounded Translation

The grounded translation is Daniel Gnad's implementation of the method proposed by Hickmott et al. [10] which is part of his PhD thesis [5]. This implementation uses the grounding process of the planning tool Fast Downward [8] on the lifted planning problem before the grounded representation is then translated to a P/T net. This implementation has been modified by Peter Gjøl to output the P/T net in PNML format, which is understood by TAPAAL. We have further modified it to add a unique identifier to the name of each transition, to prevent an issue which caused multiple actions to have the same name after the grounding process. This was an issue as it is not allowed to have multiple transitions of the same name in VerifyPN.

[1] https://github.com/petrinet-planning/colored-lifted-planning
[2] https://github.com/petrinet-planning/verifypn/commit/870405098fbe4471f4a23705d260789dd99d7bc1
[3] https://github.com/petrinet-planning/verifypn/commit/9f4fe30a3275e9e2af9fc85025831ebea83ec73e

4.3 Test Suite

The experiments are managed through a testing suite[4] we have written in Python 3.9.2. It uses a config file with startup parameters for each of the three tools, the set of benchmarks to run them on, and some sample size *samplecount*. It then generates run-scripts for the SLURM cluster management system. These run-scripts then translate each benchmark for each individual tool and runs each tool on that benchmark *samplecount* times. Everything is timed through the Unix `time` command, which records the actual user and system time spent by the process. After the experiments have been run, the suite can then analyse the results and store them in a compressed format used for plotting.

5 Experiments

In this section, three methods of non-optimal planning are tested and compared. The translation proposed in Section 3, which will be referred to as Colored. The existing translation from a grounded planning problem to a P/T net which will be referred to as Grounded, and lastly Fast Downward which is an existing planning system[7] that will be referred to as simply Downward. Since the unfolded Colored and Grounded translations have their Petri net structure in common, these are also compared on metrics other than pure planning performance such as the number of places and transitions generated for a task.

Autoscale Benchmarks The Autoscale benchmark suite from Torralba et al.[19] includes a number of different domains, each containing 30 planning tasks, P01 through P30. These are generally constructed in a way, such that they get incrementally harder from P01 to P30. Not every domain has been used in these tests. The exact ones used being evident from Figure 13, where the names refer to the agile/satisficing versions of the domains, and to list of the reasons some of the benchmarks have been skipped:

- 8 benchmarks are included,
- 7 cases where the parser we used, Unified Planning[5], fails to parse the benchmark,
- 10 benchmarks use type inheritance,
- 6 benchmarks use the delete-relationship from Section 3,
- 5 benchmarks have multiple domain files, which isn't a problem on its own, but we did not account for it in the test suite,
- 5 cases where the translation fails in PNML generation, because it was originally written for multi-sets was unclear how to handle the weight of unioning a weighted color on itself. This is solvable for the set-based semantics as $A \cup A = A$, with no ambiguous weight, but this problem was investigated too late in the process to run tests on.
- 1 case where the "hiking" benchmark causes an obscure assertion error from Unified Planning with no message, when prompted for the name of the object *couple*0 while the initial marking is generated.

[4] https://github.com/petrinet-planning/benchmarking
[5] https://github.com/aiplan4eu/unified-planning

Configurations Colored and Grounded are both run through VerifyPN, where Colored specifically uses the VerifyPN we have modified in Section 4.1. The modified VerifyPN first unfolds the CPN with "aggressive reduction" of the colors, turning the CPN into a P/T net. From here the two approaches are configured the same, aside from whether they run on the base version of VerifyPN or the modified one. VerifyPN is run with structural reductions set to "aggressive reduction", letting it attempt to reduce the size of the P/T net. After unfolding and reductions, the to search for a trace is done using Random Potency-First Search (RPFS) [9], which is chosen as the search strategy for both Petri net approaches as it showed very promising initial results upon testing various search strategies.
For the configuration of Downward, `lama-first`[17] was chosen as it is a powerful non-optimal planning strategy as showcased in 2023 International Planning Competition [4] where it was used as a baseline for comparing the entries and ended up getting an overall better result than all of them in the agile track.

5.1 Results

Each configuration has been run on each task in the benchmark 10 times and the results have been plotted. Figure 13 shows an overview of how much time it has taken each configuration to find a plan. There seems to be four types of results here:

1. In blocksworld, ged, miconic and scanalyzer, Colored and Grounded perform similar to one another, and is significantly outperformed by Downward,
2. in rovers, Downward outperforms Grounded, which outperforms Colored,
3. in pegsol Grounded and Downward perform similar to one another, but Colored is orders of magnitude slower,
4. in childsnack and visitall, Colored and Grounded solve only a few tasks, and it's difficult to compare their performance based on this plot, though Grounded *does* solve more tasks in visitall than Colored does.

Case 1 is somewhat expected. Colored and Grounded use the same search strategy in VerifyPN, so it makes sense for them to perform similarly to one another. Furthermore, the used Downward configuration is proven to be strong, even when compared to other state-of-the-art planners which are all developed with these well-known benchmarks in mind. For case 2 and 3, there are two key factors to investigate concerning the difference between Colored and Grounded. First, there is the unfolding. How much time is spent on unfolding? Secondly, the quality of the final P/T net, where quality refers to how quickly VerifyPN can verify properties of it. To investigate this, we will look at the verification time reported by TAPAAL, as well as the number of places and transitions in the final, structurally reduced P/T net. These three measures are independent of the unfolding time of the CPN, helping us narrow down what may cause the roughly 3 orders of magnitude difference in pegsol.

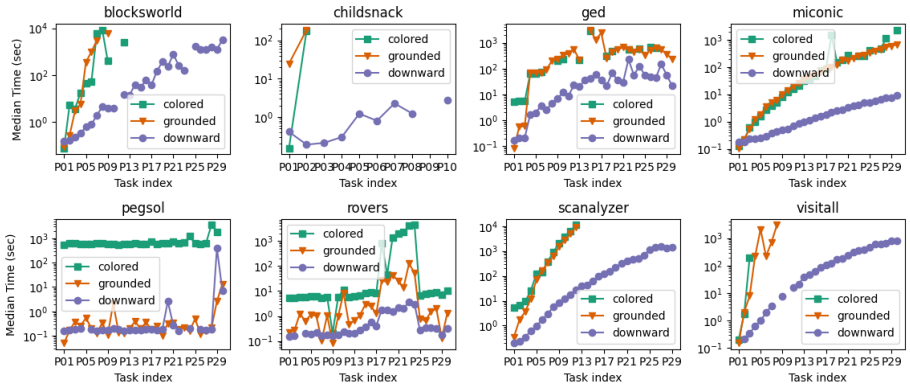


Fig. 13: Median time to find a plan over 10 samples.
Measured as user time + system time from the unix `time` command. For Colored it is the time of calling VerifyPN on the CPN, for Grounded it is the time of calling VerifyPN on a P/T net, and for Downward it is the time of calling fast-downward.py on a SAS file.

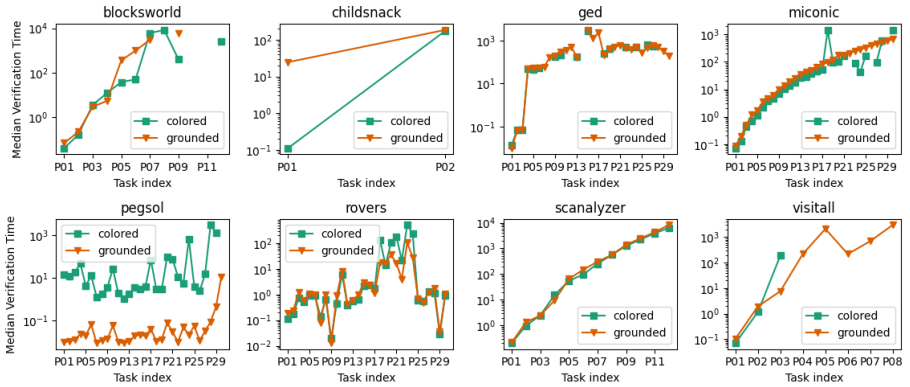


Fig. 14: Median verification time reported by VerifyPN over 10 samples. Verification time excludes the time spent unfolding the CPN, as well as structural reductions.

In Figure 14, we see the verification time reported by VerifyPN. The most interesting difference to observe from the previous figure is how Colored and Grounded perform more closely to one another in pegsol and rovers. In the general time measurement, Colored was roughly 3 orders of magnitude slower than Grounded on pegsol, but here it is "only" about 2 orders. In rovers there is still a range where Colored is slower than grounded, but they generally have similar verification time. This hints towards the difference in rovers mostly being in unfolding. For pegsol, there is still a large difference after accounting for unfolding, so this hints towards there also being a difference in quality of the generated P/T nets. To look into this, the number of places and transitions in the reduced P/T nets has been plotted in Figures 15 and 16.

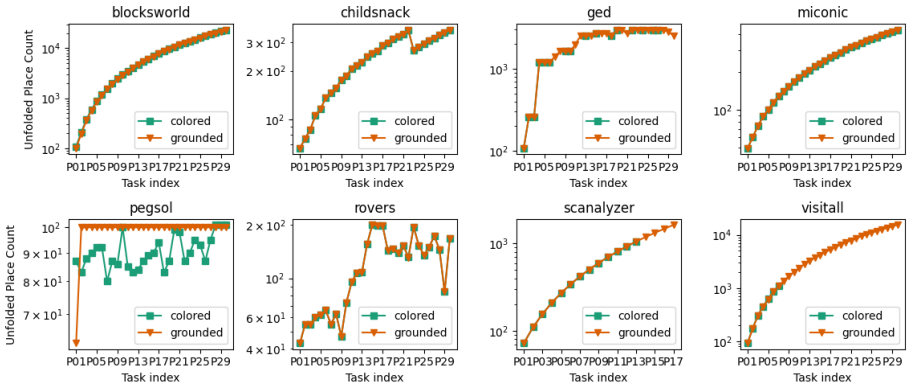


Fig. 15: Number of places in the P/T net after reductions by VerifyPN. These plots include more data points than the time measurement figures, as counting the number of places in the P/T net only requires the generation of the P/T net, without the necessity of finding a trace.

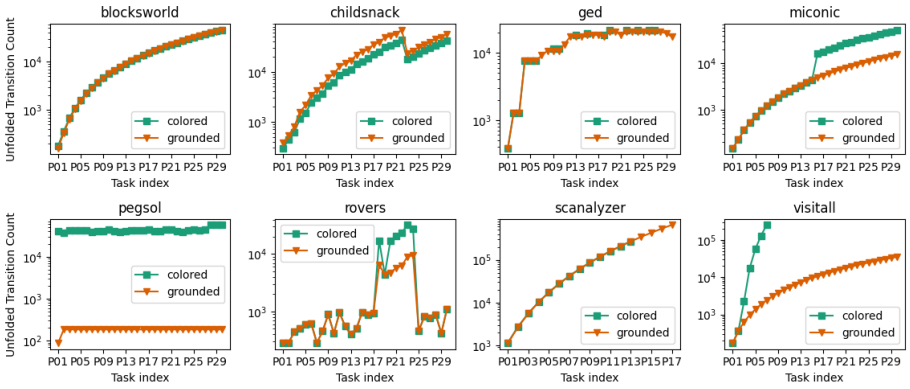


Fig. 16: Number of transitions in the P/T net after reductions by VerifyPN.

There are two surprising things in Figure 15. Firstly, Colored has fewer places than Grounded in pegsol, which would seem like a good thing, but Colored spends longer time on verification that Grounded does. Secondly, the translation used in Grounded is made for a multiset-based P/T net, so extra places have been added to make it 1-safe, but it generally has a very similar number of places. Additionally, there is a big difference in visitall, and to a lesser extent in scanalyzer. VerifyPN has managed to unfold and structurally reduce P01-P13 in scanalyzer, and P01-P07 for visitall, while Grounded has managed P01-P17 and P01-P30.

The number of transitions shown in Figure 16 seems to correspond well with the differences observed in verification time. Colored has roughly 2 orders of magnitude mode transitions in pegsol than grounded, which matches the difference in verification time. In rovers the same range of tasks where Colored spent a longer time, Colored has more transitions than Grounded. The rapidly increasing number of transitions in visitall also seems to explain why VerifyPN would fail to unfold beyond P07, the same doesn't seem to hold for scanalyzer though, where they have the same number of transitions. Surprisingly miconic also has a range of tasks with an increased number of transitions for Colored compared to Grounded, but this does not appear in the verification time.

As Colored and Grounded are both tested through VerifyPN with the same runtime arguments, the difference in verification time is dependent on the structure of the final P/T net of each method. Recall Figure 17 from the introduction.

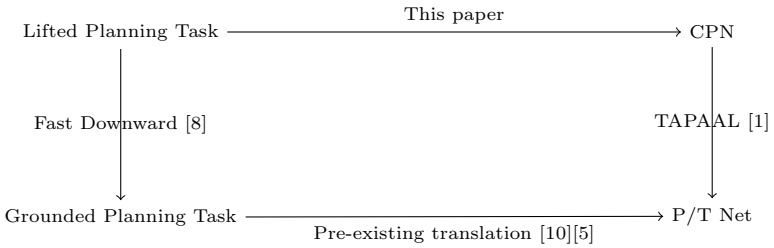


Fig. 17: Copy of Figure 1. Note: Tapaal uses VerifyPN.

A key difference between the P/T nets of Colored and Grounded is that the CPN of Colored is unfolded to a P/T net by VerifyPN, while for Grounded, the lifted task is first grounded with Fast Downward. This lends towards these experiments being a comparison between the capabilities of Fast Downward to reduce the amount of actions and predicates during grounding, and the capabilities of VerifyPN to unfold to a P/T net while applying its own reductions.

It seems reasonable that the difference in time it takes to find a plan/trace for Downward compared to Grounded comes down to how each of them searches for the plan/trace. Likewise, the difference between Grounded and Colored would mostly be the difference between how Downward and VerifyPN grounds or unfolds the lifted or colored representation. This difference in performance between the two engines opens up for possible future work of investigating whether techniques from Fast Downward can be applied to Petri net tools to reduce the number of transitions generated when unfolding a colored Petri net, or search strategies for speeding up the time it takes to find a trace.

6 Extending the Translation

This section defines an extension to the translation presented in Section 3 to support the type hierarchies with inheritance that exist in many lifted planning domains.

We formally define the syntactic translation from a lifted planning domain to a colored Petri net with set-based semantics. For this purpose, we define a constant lifted planning domain for the translation:

$$\mathit{\Pi} = \langle (\mathcal{T}, \leq), \mathcal{O}, X, \mathcal{P}, \mathcal{P}_{\mathcal{X}}, \mathcal{A}_s \rangle$$

As a CPN does not contain the explicit notion of a type hierarchy as exists in a planning domain, this extended translation truncates all types into a single color type called *object* s.t.

$$\mathbb{C}_{T_{atom}} = \{object\} \text{ and } \mathbb{C}_{atom} = typeValues(object) = \mathcal{O}$$

The type hierarchy is then enforced through guard expressions by use of this. The translated CPN $\mathcal{N} = (\mathbb{P}, \mathbb{T}, \mathbb{V}, Arcs, Guards)$ is constructed as follows:

1. $\mathbb{P} = \mathcal{P}$, where $typeOf(p) = (\underbrace{object, \ldots, object}_{arity(sig(p)) \text{ times}})$ for every $p \in \mathbb{P}$,
2. $\mathbb{T} = \mathcal{A}_s$,
3. $\mathbb{V} = X$, and $typeOf(v) = object$ for $v \in \mathbb{V}$,
4. *Arcs* are translated in accordance with Section 3.1,
5. *Guards* are constructed according to the procedure below.

Guard Expressions As described in Section 2.2, there are type restrictions regarding which objects a parameter can be bound to s.t. $typeOf(o) \leq typeOf(x)$. The valid atomic colors of a variable are the same as the valid objects of the corresponding parameter as defined by the type hierarchy $(\mathcal{T}, \leq)$. Thereby, the function $validColors : X \to 2^{\mathbb{C}_{atom}}$ returns the colors corresponding to the objects that are valid for the parameter, using the fact that $\mathbb{C}_{atom} = \mathcal{O}$:

$$validColors(x) = \{o \mid o \in \mathcal{O}, typeOf(o) \leq typeOf(x)\}$$

Let $variablesIn : \mathbb{AE} \to 2^{\mathbb{V}}$ return all variables that occur inside an arc expression, and $allAE : \mathbb{T} \to 2^{\mathbb{AE}}$ is defined as:

$$allAE(t) = \bigcup_{p \in \mathbb{P}} \{Arcs(p,t)\} \cup \{Arcs(t,p)\}$$

For every $t \in \mathbb{T}$, construct the following guard:

$$\bigwedge_{ae \in allAE(t)} \bigwedge_{v \in variablesIn(ae)} \bigvee_{c \in validColors(v)} x = c$$

6.1 Extension - Example

For this example, let $(\mathcal{T}, \leq)$ be the type hierarchy in Figure 18, $\mathcal{O} = \{rectangle1, rectangle2, square1, triangle1\}$, and the 2 parameterized predicates in the action schema corresponding to transition t be $p_1(x_1^1, x_2^1)$ and $p_2(x_2^2)$. The types of their parameters are $typeOf(x_1^1) = rectangle$, $typeOf(x_2^1) = triangle$, and $typeOf(x_1^2) = square$. The following guard expression is constructed:

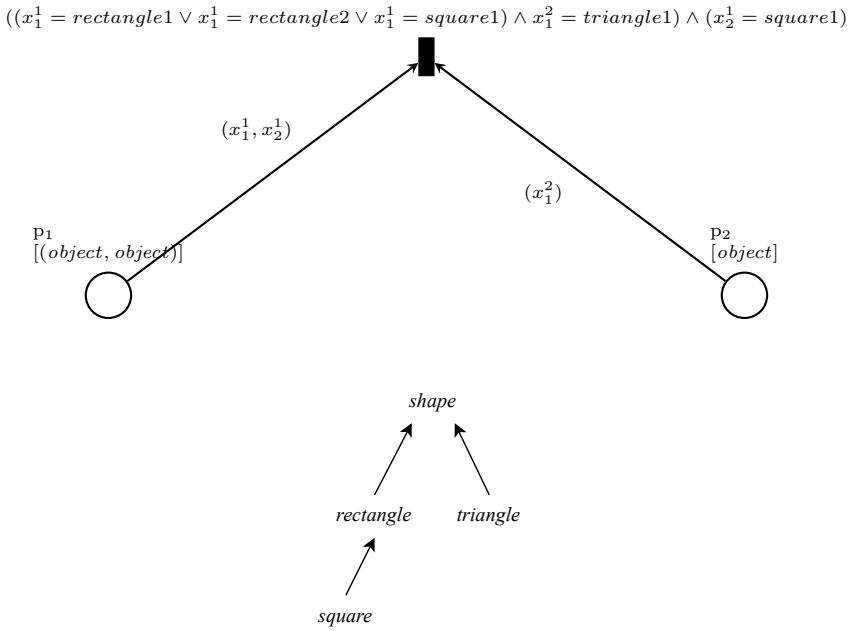


Fig. 18: Type hierarchy.

7 Conclusion and Future Work

7.1 Conclusion

We have defined lifted planning tasks and colored Petri nets with set-based semantics (CPN) and provided a method for translating a large subset of lifted planning tasks to CPNs, as well as a method for using Petri net tools such as TAPAAL to provide a plan for a lifted planning task. This translation has been used to compare a planning system and a Petri net tool, which have shown differences in the performance of how they each ground or unfold their planning task or CPN respectively. In the experiments, the planning system Fast Downward has generally outperformed both the pre-existing grounded translation and the translation proposed in this paper. The grounded and colored translations have very similar results in terms of time it takes to find a plan for the Autoscale benchmarks. However, the grounded translation has better results in particular domains concerning time and the number of transitions in the final P/T net representation of the planning task. The difference in performance between the proposed colored translation and the grounded one in particular benchmarks, shows potential for improvements to the CPN unfolding process. Moreover, the large gap between both the Petri net approaches and the used Fast Downward configuration show that Petri net solvers may be improved with techniques used in the field of planning.

7.2 Future Work

Support for delete-relationships In Section 3.1, the delete-relationship is defined as occurring if for some $a_s = (pre, del, add) \in \mathcal{A}_s$, there exists $p_x \in pre \cup del \cup add$ s.t. $(p_x \notin pre) \wedge (p_x \in del) \wedge (p_x \notin add)$. This is more difficult to translate than the other relationships as the effect of applying the action constructed by binding the predicate depends on whether or not it is in that state. If $\exists p_x[b] \in s$ then $s \xrightarrow{a} s'$ where $p_x[b] \notin s'$. However, the action must still be applicable even if $p_x[b] \notin s$ as $p_x \notin pre$, in which case $s \xrightarrow{a} s$. This is not a problem with the semantics of planning, however, the semantics of firing a transition require $Arcs(p, a_s)[b] \subseteq translate(s)(p)$ or the transition will not be enabled. A way to solve this dilemma would be to create multiple copies of the transition corresponding to the action schema, with one consuming the color, and one only being enabled if $Arcs(p, a_s)[b] \nsubseteq translate(s)(p)$. This would be enforced via an inhibitor arc. A sketch of this solution is presented in Figure 19.

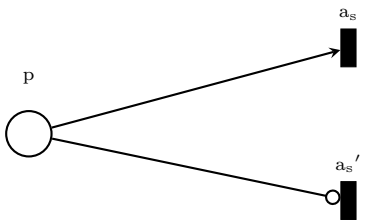


Fig. 19: Sketch of how the delete-relationship could be translated using inhibitor arcs and multiple transitions.

Implementing Type Inheritance Implementing Section 6 would make it possible to experiment with more than double the number of domains presented in this paper. The results from these could support the trends already discovered, or hint at new things to investigate. Either way, it would be interesting to double the amount of experimental data, which would also be helpful in pointing out the areas which have the most room for improvement.

Translating CPNs to Lifted Planning Tasks With the lama-first configuration of Fast Downward outperforming both the Grounded and Colored translations, it would be interesting to investigate how well this and other planning approaches perform on benchmarks for testing verification on CPNs. Since the translation proposed in this paper already establishes many correspondences between lifted planning and CPNs, much of the groundwork for a translation in the opposite direction has already been made.

Acknowledgements. We would like to thank Álvaro Torralba and Jiří Srba for great guidance throughout the project, and particularly Álvaro for providing his expertise in planning and likewise Jiří for his expertise in Petri nets.

References

1. Bilgram, A., Jensen, P.G., Pedersen, T., Srba, J., Taankvist, P.H.: Improvements in unfolding of colored petri nets. In: Bell, P.C., Totzke, P., Potapov, I. (eds.) Reachability Problems. pp. 69–84. Springer International Publishing, Cham (2021)
2. David, A., Jacobsen, L., Jacobsen, M., Jørgensen, K., Møller, M., Srba, J.: TAPAAL 2.0: integrated development environment for timed-arc Petri nets. In: Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12). LNCS, vol. 7214, p. 492–497. Springer-Verlag (2012)
3. Fikes, R.E., Nilsson, N.J.: Strips: A new approach to the application of theorem proving to problem solving. Artificial Intelligence **2**(3), 189–208 (1971). https://doi.org/https://doi.org/10.1016/0004-3702(71)90010-5, https://www.sciencedirect.com/science/article/pii/0004370271900105
4. Fišer, D., Pommerening, F.: International planning competition 2023 classical tracks. Presentation of results (2023), https://ipc2023-classical.github.io/results/presentation.pdf
5. Gnad, D.: Star-Topology Decoupled State-Space Search in AI Planning and Model Checking. Ph.D. thesis, Saarland University (2021)
6. Gupta, N., Nau, D.S.: On the complexity of blocks-world planning. Artificial Intelligence **56**(2), 223–254 (1992). https://doi.org/https://doi.org/10.1016/0004-3702(92)90028-V, https://www.sciencedirect.com/science/article/pii/000437029290028V
7. Helmert, M.: The fast downward planning system. J. Artif. Intell. Res. **26**, 191–246 (2006), https://api.semanticscholar.org/CorpusID:17305
8. Helmert, M.: Concise finite-domain representations for pddl planning tasks. Artificial Intelligence **173**(5), 503–535 (2009). https://doi.org/https://doi.org/10.1016/j.artint.2008.10.013, https://www.sciencedirect.com/science/article/pii/S0004370208001926, advances in Automated Plan Generation
9. Henriksen, E.G., Khorsid, A.M., Nielsen, E., Risager, T., Srba, J., Stück, A.M., Sørensen, A.S.: Potency-based heuristic search with randomness for explicit model checking. In: Caltais, G., Schilling, C. (eds.) Model Checking Software. pp. 180–187. Springer Nature Switzerland, Cham (2023)
10. Hickmott, S.L., Rintanen, J., Thiébaux, S., White, L.B.: Planning via petri net unfolding. In: International Joint Conference on Artificial Intelligence (2007), https://api.semanticscholar.org/CorpusID:203755
11. Jensen, K.: Coloured petri nets and the invariant-method. Theoretical Computer Science **14**(3), 317–336 (1981). https://doi.org/https://doi.org/10.1016/0304-3975(81)90049-9, https://www.sciencedirect.com/science/article/pii/0304397581900499
12. Jensen, K.: Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Vol 1, Basic Concepts. Monographs in theoretical computer science: an EATCS series, Springer, Netherlands, 2. ed., 2. corr. printing edn. (1997)
13. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4), 541–580 (1989). https://doi.org/10.1109/5.24143
14. Petri, C.A.: Communication with automata (1966), https://api.semanticscholar.org/CorpusID:8420535
15. Poole, D.L., Mackworth, A.K.: Artificial intelligence, pp. 239–240. Cambridge University Press, Cambridge, England, 2 edn. (Sep 2017)

16. Ratzer, A.V., Wells, L., Lassen, H.M., Laursen, M., Qvortrup, J.F., Stissing, M.S., Westergaard, M., Christensen, S., Jensen, K.: Cpn tools for editing, simulating, and analysing coloured petri nets. In: van der Aalst, W.M.P., Best, E. (eds.) Applications and Theory of Petri Nets 2003. pp. 450–462. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
17. Richter, S., Westphal, M.: The LAMA planner: Guiding cost-based anytime planning with landmarks. CoRR **abs/1401.3839** (2014), http://arxiv.org/abs/1401.3839
18. Slaney, J., Thiébaux, S.: Blocks world revisited. Artificial Intelligence **125**(1), 119–153 (2001). https://doi.org/https://doi.org/10.1016/S0004-3702(00)00079-5, https://www.sciencedirect.com/science/article/pii/S0004370200000795
19. Torralba, Á., Seipp, J., Sievers, S.: Automatic instance generation for classical planning. In: Goldman, R.P., Biundo, S., Katz, M. (eds.) Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021). pp. 376–384. AAAI Press (2021)