Translating Colored Petri Nets to Lifted Numeric Planning Tasks

& Lifted Planning Tasks to Colored Petri Nets

Henrik Ginnerup and Sebastian Lassen

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

Thesis Summary

This thesis contains two main contributions. The first builds on our previous translation from lifted classical planning to colored Petri nets (CPN) with set-based semantics by extending the range of lifted planning tasks that can be translated. This was done through reworking the color types, and the order colors were added from the planning task. Thus allowing type inheritance to be modeled using guard expressions. The experiments show that it can solve substantially more planning tasks than before, however, it is still slower when compared to state-of-the-art planners and a competing translation from a grounded planning problem.

The second contribution is a novel translation from a general CPN to lifted numeric planning. In this translation, places are represented as numeric variables, the partial order of colors is maintained through classical predicates and numeric variables. An issue was found for the translation of transitions to action schemas, but a concrete solution is proposed as future work. A theorem is presented along with an argument for the correctness of the method. The entire translation is implemented within the Java frontend of TAPAAL. The MCC2023 colored Petri net benchmarks are used to test the implementation. The benchmarks are translated to planning problems, after which the planning system ENHSP is used to detect satisfiability vs unsolvability of each query, which is then compared to the answer from TAPAAL. ENHSP with our translation managed to answer 246 queries correctly, though 9 of the resulting plans have been flagged as potentially affected by the discussed issue. ENHSP and TAPAAL disagreed *once*, but it is in a flagged case, so it is outside the restrictions of the theorem.

Translating Colored Petri Nets to Lifted Numeric Planning Tasks
& Lifted Planning Tasks to Colored Petri Nets

Henrik Ginnerup and Sebastian Lassen

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

Abstract. We present a novel method for translating general colored Petri nets to lifted numerical planning tasks, along with arguments for its correctness. Additionally, the paper also presents an improvement to a previous translation of classical lifted planning to a CPN. Both translations are implemented, have their output validated, and are run against other tools. The first translation is run with the numeric planner ENHSP and is compared to the Petri net verification tool TAPAAL. The second translation is run in TAPAAL and compared to a translation made from a grounded planning task and the powerful Fast Downward configuration, lama-first. The first translation is generally outperformed by TAPAAL, but more importantly, it correctly categorizes reachability queries within the limitation of the theorem as satisfiable or unsolvable, with a proposed solution for removing the limitation. The second translation is an improvement to the former, but is still substantially slower than the Fast Downward configuration and the grounded translation. However, its outputs are correct and it can now translate a much larger subset of lifted planning tasks.

Keywords: Lifted planning · Colored Petri nets · Lifted numerical planning · Translation · Model checking

1 Introduction

This paper explores the connection between *colored Petri nets* and *lifted planning tasks*. Colored Petri nets (CPN)[15] are an extension to the modeling formalism known as P/T nets[18], which are typically used to model distributed systems s.t. different properties of these can be verified. Translating a system to a Petri net is particularly useful as many powerful techniques are developed for checking their properties [17]. One such property is whether it is possible to go from one state in the modeled system to another, known as a reachability query. This concept has a lot in common with planning, where the objective is to find a sequence of actions which will lead from a starting state to a desired goal state [19]. We previously explored these connections in Ginnerup H., Lassen S. [7], culminating in an implementation of a translation from a reduced subset of classical lifted

planning to CPN with set-based semantics. The restrictions placed on this initial translation and the strong results showcased by the planning system Fast Downward (FD)[10] in the experiments were the initial motivation for this paper.

There is not much related work in the area of translating planning tasks to Petri nets and vice versa but Hickmott et al. [13] remains an important milestone, as they created a translation from a planning task grounded by FD[11] to a 1-safe P/T net. However, the translated net was not used with a general Petri net modelling and verification tool such as CPN Tools[20] or TAPAAL[1], but instead with a heuristic-directed unfolder. This implementation was later modified by Daniel Gnad as part of his PhD thesis [8], even more recently modified by Peter Gjøl and us [7] to output correct PNML s.t. it can be used in conjunction with modern tools like TAPAAL.

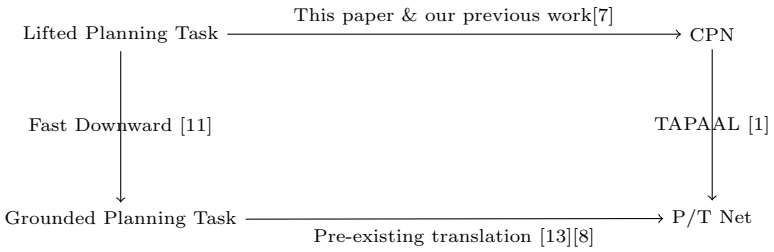


Fig. 1: An overview of the difference between the first translation proposed in this paper, and the pre-existing methods for translating planning tasks to Petri nets [7].

This paper has two main contributions. The first is shown in Figure 1 and is a less restricted translation from a classical lifted planning task to a CPN. The biggest restriction on the previous translation was the lack of support for type inheritance, which is the major new feature of the improved translation. However, the largest contribution of this paper is a translation from a general colored Petri net to a lifted numeric planning task. Additionally, this new translation has been implemented and is used to solve Model Checking Competition (MCC)[16], CPN benchmarks by searching for a plan with the numeric planner ENHSP[22]. This translation is shown in relation to other existing tools in Figure 2. An additional, smaller contribution stemming from this and the previous paper, is that we have contributed 2 new models to the MCC 2024 call-for-models. The first was made using the initial translation from lifted planning task to CPN, whereas the second was made using the enhanced translation from this paper.

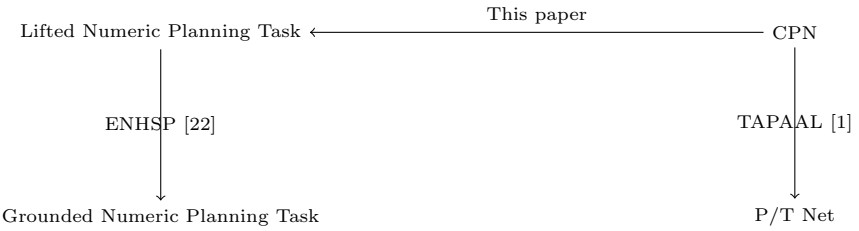


Fig. 2: An overview of the new translation proposed in this paper, and how it fits in with pre-existing tools.

Section 2 presents the syntax of a colored Petri net along with two sets of semantics, one for each translation. Section 3 presents the enhanced translation from classical lifted planning task to a CPN with set-based semantics, and Section 4.3 presents the novel translation from muliset-based CPN to a lifted numeric planning task. Each of the two former sections will also present the implementations and experimental results for their respective translation. Finally, Section 5 presents a joined conclusion and proposes future work.

1.1 Bibliographical Remarks

This paper builds on our previous work in Ginnerup H., Lassen S.[7]. In Section 2, most of the CPN syntax is reused, but the multiset-based semantics and several examples are new additions. Section 3 reuses both of the classical lifted planning definitions and their semantics, with some minor tweaks, but the translation has been changed substantially, and new, more relevant examples have been added. The definitions of grounded- and lifted numeric planning in Section 4.3 are new, but are defined as extensions to the existing definitions classical definitions. Additionally, the entirety of the translation from lifted numeric planning to CPN and the accompanying arguments of correctness are entirely new. All of the experiment setups, results, and discussions surrounding them are new.

Section 2.3 is new, but the rules and figures displaying their applications are borrowed from multiple sources[4] [2][3].

1.2 Preliminaries

Let $\mathbb{Z}$ denote all integers, and let $\mathbb{Z}^{\geq 0}$ denote all positive integers including 0. A multiset is an unordered collection where each element occurs finite many times. Given a set A, then a multiset over A is defined as A along with any function $Multiplicity : A \to \mathbb{Z}^{\geq 0}$. Let $2'a + 1'c$ denote a multiset where a has a multiplicity of 2, c has a multiplicity of 1, and anything else has a multiplicity of 0. Let the function $toSet : (\mathbb{U} \to \mathbb{Z}^{\geq 0}) \to \mathbb{U}$ map any multiset to the set of elements

with a multiplicity greater than 0, s.t. $toSet(2'a + 0'b + 1'c) = \{a, c\}$.

The operator $\uplus$ denotes the union of disjoint sets. Therefore, $\mathcal{A} \uplus \mathcal{B} \equiv \mathcal{A} \cup \mathcal{B}$ where $\mathcal{A} \cap \mathcal{B} = \emptyset$

Let $f = \{x_1 \Rightarrow y_1, ..., x_n \Rightarrow y_n\}$ denote a function f where $f(x_1) = y_1, ..., f(x_n) = y_n$.

2 Syntax and Semantics of Colored Petri Nets

This section defines the syntax, and then two different semantics for colored Petri nets. First, a set-based semantic, which will be used in the translation from a planning task to a CPN, and then a more common multiset-based semantic, which will be used for translating a multiset-based CPN to a lifted numeric planning task.

2.0.1 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 $typeOf : \mathbb{C}_{atom} \to \mathbb{C}_{T_{atom}}$ returns the type of a color, and inversely $typeValues : \mathbb{C}_{T_{atom}} \to 2^{\mathbb{C}_{atom}}$ returns all colors of a given type, s.t. given $c \in \mathbb{C}_{atom}$ and $t \in \mathbb{C}_{T_{atom}}$ then $typeOf(c) = \tau$ iff $c \in typeValues(\tau)$.

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 $typeOf : \mathbb{C} \to \mathbb{C}_T$ is defined as:

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

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

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

2.0.2 Color Predecessor and Successor Atomic colors are assumed to be totally ordered within their type, s.t. given a type $\tau \in \mathbb{C}_{T_{atom}}$ and $typeValues(\tau) = \{c_1, c_2, c_3, ..., c_n\}$, then $c_1 \prec c_2 \prec c_3 \prec ... \prec c_n$. The *predecessor* function $pred : \mathbb{C}_{atom} \to \mathbb{C}_{atom}$ returns largest smaller value, e.g. $pred(c_2) = c_1$, but is cyclic s.t. $pred(c_1) = c_n$. Similarly the *successor* function $succ : \mathbb{C}_{atom} \to \mathbb{C}_{atom}$ is the smallest larger color, e.g. $succ(c_2) = c_3$ and $succ(c_n) = c_1$.

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

2.0.4 Atoms An atom is either an atomic color or a variable, s.t. given finite sets of atomic colors $\mathbb{C}_{atom}$ and variables $\mathbb{V}$, then:

$$\mathbb{A} = \mathbb{C}_{atom} \cup \mathbb{V}$$

Atoms can be a reference to the prede- or successor, and is represented through:

$$atom^{\tau} ::= variable^{\tau} \mid atomicColor^{\tau} \mid pred(atom^{\tau}) \mid succ(atom^{\tau})$$

where τ is some atomic color type.

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

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

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

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

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

where $\bowtie \in \{<, \leq, =, >, \geq\}$

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.

2.1 Colored Petri Net

Definition 1. *A colored Petri net, 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 a 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.*

2.1.1 Set Based 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 $M(p) \subseteq \mathit{typeValues}(\mathit{typeOf}(p))$.

2.1.2 Multiset Based Marking In a multiset based Petri Net, each place contains finite many colors of a given type. A multiset based marking is defined as any function $M : \mathbb{P} \to (\mathbb{C} \to \mathbb{Z}^{\geq 0})$, with the limitation that each place can only contain colors of its own type, s.t. given place p, color c, and $\mathit{typeOf}(c) \neq \mathit{typeOf}(p)$ then $M(p)(c) = 0$.

2.2 Petri Net Semantics

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

2.2.1 Binding 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 given a binding b and an atom a, then

$$a[b] = \begin{cases} a & a \in \mathbb{C}_{atom} \\ b(a) & a \in \mathbb{V} \\ \mathit{pred}(a'[b]) & a \text{ is in the form } \mathit{pred}(a') \\ \mathit{succ}(a'[b]) & a \text{ is in the form } \mathit{succ}(a') \end{cases}$$

This further extends to sequences of atoms, s.t. $(a_1, a_2, ..., a_i)[b] = (a_1[b], a_2[b], ..., a_i[b])$.

2.2.2 Evaluating Arc Expressions Given an arc expression ae, let $ae[]$ denote the multiset where each value in ae is counted, and given a binding b then let $ae[b]$ denote the count after applying the binding.

Example: Given colors $\mathbb{C}_{atom} = \{A, B, C\}$ and variables $\mathbb{V} = \{x, y\}$, all of the same type, and $b = \{x \Rightarrow B, y \Rightarrow C\}$.
Let $ae_1 = A + A + C + x + \mathit{pred}(x) + y$, then
$ae_1[] = 2'A + 1'C + 1'x + 1'\mathit{pred}(x) + 1'y$
$ae_1[b] = 3'A + 1'B + 2'C$

Let $ae_2 = (A, B) + (A, \mathit{succ}(x)) + (C, \mathit{succ}(\mathit{succ}(y)))$, then
$ae_2[] = 1'(A, B) + 1'(A, \mathit{succ}(x)) + 1'(C, \mathit{succ}(\mathit{succ}(y)))$
$ae_2[b] = 2'(A, B) + 1'(C, A)$

2.2.3 Guard Expression Evaluation Guard expressions compare the order of colors through boolean expressions. Given a guard expression ge and binding b, let $ge[b]$ denote b being applied to every atom in ge, which is then evaluated to $true$ or $false$.

Example: Given the atomic types A and B with the colors $\{a_1, a_2, a_3\}$ and $\{b_1, b_2\}$ respectively, the variables $\{v_{A1}, v_{A2}, v_{B1}, v_{B2}\}$, $a_1 \prec a_2 \prec a_3$, $b_1 \prec b_2$ and the binding $b = \{v_{A1} \Rightarrow a_1, v_{A2} \Rightarrow a_2, v_{B1} \Rightarrow b_1, v_{B2} \Rightarrow b_2\}$, then:

The guard expression $ge = succ(v_{B1}) < v_{B2} \vee \neg(v_{A1} = v_{A2})$ is evaluated as

$$
\begin{aligned}
ge &= succ(v_{B1}) < v_{B2} \vee \neg(v_{A1} = v_{A2}) \\
ge[b] &= (succ(v_{B1}) [b]) < (v_{B2} [b]) \vee \neg((v_{A1} [b]) = (v_{A2} [b])) \\
&= (b_2) < (b_2) \vee \neg((a_1) = (a_2)) \\
&= b_2 < b_2 \vee \neg(a_1 = a_2) \\
&= true
\end{aligned}
$$

Additionally, the comparison is transitive, so $a_1 \prec a_2$ and $a_2 \prec a_3$, implies $a_1 < a_3\,[b] = true$.

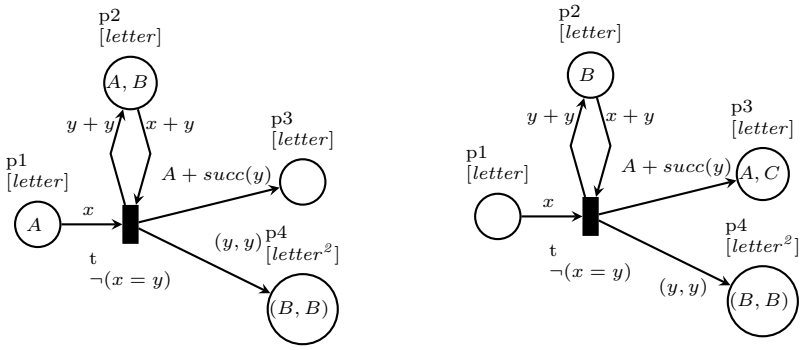
2.2.4 Set based Transition Firing A transition t under binding b is enabled for marking M, if

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

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

2.2.5 Example: Firing a Transition with Set Based Semantics Given $\mathbb{C}_T = \{letter, letter^2\}$ and $\mathbb{C}_{atom} = typeValues(letter) = \{A, B, C\}$. A CPN can be seen in Figure 37, along with the effect of firing its transition t under binding $b = \{x \Rightarrow A, y \Rightarrow B\}$.



(a) Shows M before firing t.

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

Fig. 3: Transition t fired under binding $b = \{x \Rightarrow A, y \Rightarrow B\}$ using set based semantics.

M' is computed as follows:

$M'(p)$	$= ($	$M(p)$	$\setminus toSet(Arcs(p,t)[b])$	$) \cup$	$toSet(Arcs(t,p,)[b])$	
$M'(p1)$	$= ($	$\{A\}$	$\setminus \{A\}$	$) \cup$	$\{\}$	$= \{\}$
$M'(p2)$	$= ($	$\{A,B\}$	$\setminus \{A,B\}$	$) \cup$	$\{B\}$	$= \{B\}$
$M'(p3)$	$= ($	$\{\}$	$\setminus \{\}$	$) \cup$	$\{A,C\}$	$= \{A,C\}$
$M'(p4)$	$= ($	$\{(B,B)\}$	$\setminus \{\}$	$) \cup$	$\{(B,B)\}$	$= \{(B,B)\}$

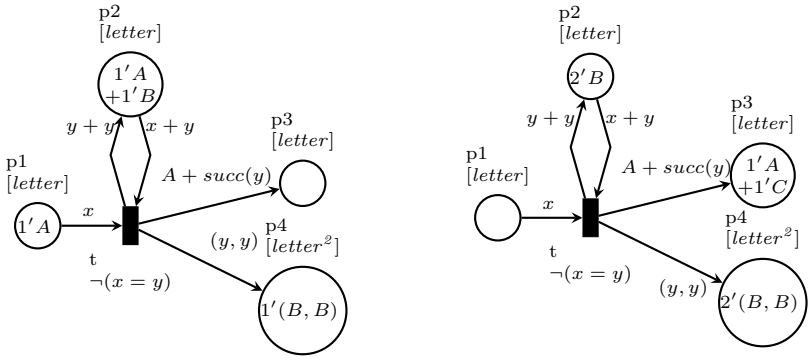
2.2.6 Multiset based Transition Firing A transition t under binding b is enabled for multiset marking M, if

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

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

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 $M \xrightarrow{\epsilon} M$.

2.2.7 Multiset Transition Firing - Example Reusing the example from the set based firing of a transition, Figure 4 shows the same CPN as Figure 37, but with a similar multiset based marking.



(a) Shows M before firing t.

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

Fig. 4: Transition t fired under binding $b = \{x \Rightarrow A, y \Rightarrow B\}$ using multiset based semantics.

M' is computed as follows:

$M'(p)$	$=$	$M(p)$	$-$	$Arcs(p,t)[b]$	$+$	$Arcs(t,p,)[b]$		
$M'(p1)$	$=$	$(1'A)$	$-$	$(1'A)$	$+$	$()$	$=$	$()$
$M'(p2)$	$=$	$(1'A + 1'B)$	$-$	$(1'A + 1'B)$	$+$	$(2'B)$	$=$	$2'B$
$M'(p3)$	$=$	$()$	$-$	$()$	$+$	$(1'A + 1'C)$	$=$	$1'A + 1'C$
$M'(p4)$	$=$	$(1'(B,B))$	$-$	$()$	$+$	$(1'(B,B))$	$=$	$2'(B,B)$

2.2.8 Set- vs Multiset based semantics - Example A set-trace is not necessarily a multiset-trace, and oppositely, a multiset-trace is not necessarily a set-trace. To demonstrate this, A CPN is given in Figure 5, after which some traces are shown.

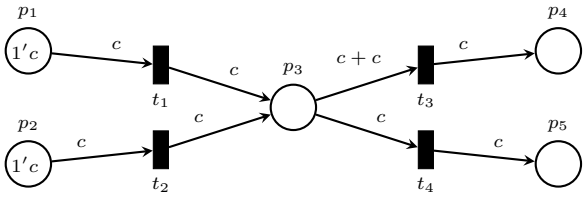


Fig. 5: A CPN with a single color c. It is shown with a multiset marking, but it can also be considered with the set based marking $\{p_1 \Rightarrow \{c\}, p_2 \Rightarrow \{c\}\}$.

Using set-semantics, it is possible to apply t_1 and then t_3. This is not possible using the multiset-semantics as $Arcs(p_3, t_3)[c]$ consumes too many tokens.

Valid set-trace	Invalid multiset-trace
$\{p_1 \Rightarrow \{c\}, p_2 \Rightarrow \{c\}\} \xrightarrow{t_1}$	$\{p_1 \Rightarrow 1'c, p_2 \Rightarrow 1'c\} \xrightarrow{t_1}$
$\{p_1 \Rightarrow \{c\}, p_3 \Rightarrow \{c\}\} \xrightarrow{t_3}$	$\{p_1 \Rightarrow 1'c, p_3 \Rightarrow 1'c\} \not\xrightarrow{t_3}$
$\{p_1 \Rightarrow \{c\}, p_4 \Rightarrow \{c\}\}$	

Using multiset semantics, it is possible to fire t_1 and t_2, storing two tokens in p_3, and then firing t_4 twice. This is not possible in the set semantics, as they cannot store multiple tokens of a single color in a single place.

Invalid set-trace	Valid multiset-trace
$\{p_1 \Rightarrow \{c\}, p_2 \Rightarrow \{c\}\} \xrightarrow{t_1}$	$\{p_1 \Rightarrow 1'c, p_2 \Rightarrow 1'c\} \xrightarrow{t_1}$
$\{p_1 \Rightarrow \{c\}, p_3 \Rightarrow \{c\}\} \xrightarrow{t_2}$	$\{p_1 \Rightarrow 1'c, p_3 \Rightarrow 1'c\} \xrightarrow{t_2}$
$\{p_3 \Rightarrow \{c\}\} \xrightarrow{t_4}$	$\{p_3 \Rightarrow 2'c\} \xrightarrow{t_4}$
$\{p_5 \Rightarrow \{c\}\} \not\xrightarrow{t_4}$	$\{p_3 \Rightarrow 1'c, p_5 \Rightarrow 1'c\} \xrightarrow{t_4}$
	$\{p_5 \Rightarrow 2'c\}$

2.3 Examining Petri Net Techniques

Let a *P/T net* be defined as a CPN with only a single color, and where all guards are considered as $true$. Any CPN can be *unfolded* to a bisimilar P/T net[1].

Since the definition of a colored Petri net with set-based semantics that is used in Section 3 is non-standard, the soundness of various Petri net unfolding techniques and reductions is not guaranteed. In this section, the soundness of techniques we utilize with our set-based CPNs will be examined and argued. Additionally, an example of why certain techniques are not safe when sets are used instead of multisets will be showcased. We with an illegal reduction, as it builds upon Figure 5 above.

2.3.1 Illegal Reduction: A Structural Reduction rule A, shown in Figure 8, is an example of an illegal reduction.

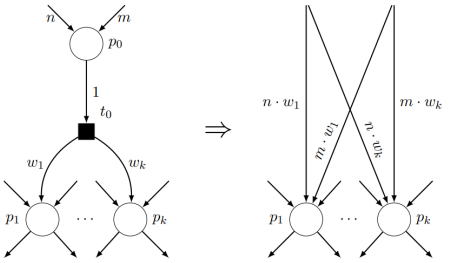


Fig. 8: Structural Reduction rule A.[2]

Intuitively, rule A says that that only t_0 can consume tokens from p_0, so the P/T net can be reduced by firing t_0 until p_0 has no tokens remaining, and then any arcs feeding into p_0 is instead connected to every place that came after t_0. Both p_0 and t_0 can then be removed.

Figure 5 from the previous Section is a counter example to rule A being safe. Rule A can immediately be applied to (p_1, t_1), moving the token from p_1 into p_3. Next it can be applied on (p_2, t_2), moving the token of p_2 into p_3. By applying structural reductions, we have now moved two tokens into the same place, and one of them disappeared. Before structural reductions, it would have been possible to reach a marking $\{p_3 \Rightarrow \{c\}, p_5 \Rightarrow \{c\}\}$, but it is no longer possible, because a token disappeared in the set semantics.

2.3.2 Unfolding techniques We have utilized the following techniques when unfolding CPNs with set-based semantics in TAPAAL:

Color fix point is safe and used, as the method itself is set-based. It uses set semantics to over approximate which places can ever contain each color.

We have not used not used partial order reductions, as they rely on the assumption that the order of firing two transitions does not matter. The order does however matter for set-based.

2.3.3 Structural reductions of colored Petri nets We have used the following structural colored reductions in conjunction with our CPNs with set-based semantics:

Parallel transitions, dead transitions, dead places, irrelevant places and transitions

All the colored reductions and figures in this section are from van Diepen et al. [4] where these are first introduced.

Parallel transitions This rule concerns removing parallel colored transitions. These are transitions where firing one is equivalent to firing the other k times. Since every arc in the nets produced by our translation always have a weight W of 1, this rule removes transitions that behave exactly the same as other existing transitions. Therefore, it is safe to use.

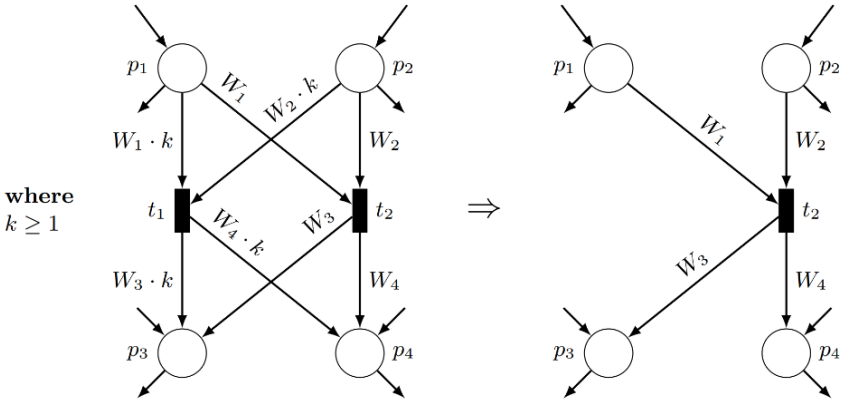


Fig. 9: An application of the parallel transition reduction[4].

The same reasoning of how the set-based semantics does not interfere with the rules as the arc expressions are also set-based applies to the other utilized colored structural reductions.

2.3.4 Structural reductions of P/T nets In conjunction with our CPNs with set-based semantics, we have utilized the following structural reduction rules:

C, D, E, G, I, L, M

Rules A to I and the figures showing them are from Bønneland et al. [2]. Rules L and M along with related figures are likewise from van Diepen et al. [3].

As an example of a safe reduction, see reduction C in Figure 10. For set semantics, the weights can be assumed to be 1. Then p_1 and p_2 will always have the same marking, and it is safe to remove one of them.

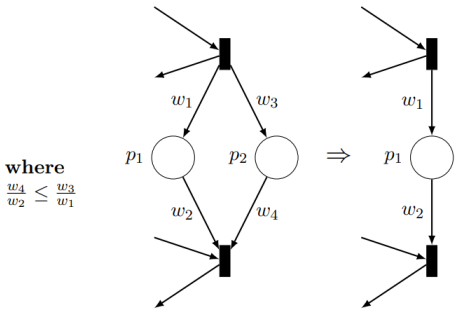


Fig. 10: Structural Reduction rule C.[2]

As another example of a safe reduction, see reduction E in Figure 11. It states that a single firing of t_0 requires more tokens than can ever be in p_0. There is no way for t_0 to ever be fired, so it can safely be removed.

Fig. 11: Structural Reduction rule E.[2]

3 Lifted Planning Tasks to Colored Petri Nets

The section contains an explanation of how a given planning state is translated to a CPN marking and vice versa, culminating in an argument of correctness in the form of a proof sketch.

3.1 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 [5]. In this definition, states are comprised of propositions typically referred to as facts or atoms, denoting whether a fact is true in that state.

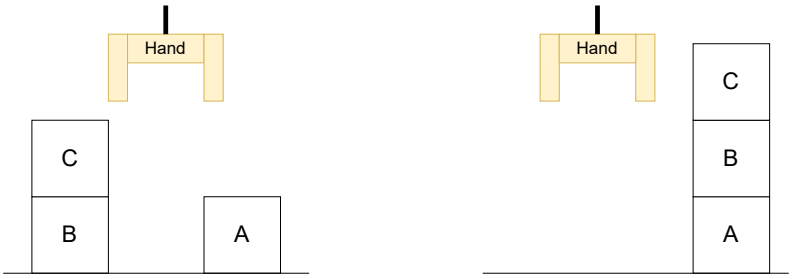


Fig. 12: A possible initial and goal state of a container stacking task.

Definition 2. *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 $G \subseteq s$.

3.2 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 task part. The domain describes the general part of a task in terms of constant objects, predicates, and action schemas. The task then specifies the instance of the domain that is to be solved by stating the initial state, the goal facts, and any additional objects.

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 schemas 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 3.1.

Definition 3. *A lifted planning domain is a 6-tuple* $\Pi = \langle (\mathcal{T}, \leq), \mathcal{O}, \mathcal{X}, \mathcal{P}, \mathcal{P}_\mathcal{X}, \mathcal{A}_s \rangle$ *where:*

1. $(\mathcal{T}, \leq)$ *is a finite, partially ordered 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. $\mathcal{X}$ *is a finite set of parameters, each with a type indicated by* $typeOf : \mathcal{X} \to \mathcal{T}$*,*
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.* $p \in \mathcal{P}$*,* $x_i \in \mathcal{X} \cup \mathcal{O}$*, 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 $b : \mathcal{X} \to \mathcal{O}$. 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 $\mathcal{B}_{Val} \stackrel{\text{def}}{=} \{b \mid b \in \mathcal{B}, x \in \mathcal{X}, typeOf(b(x)) \leq typeOf(x)\}$. 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 $\mathcal{A} = \{(pre[b], del[b], add[b]) \mid (pre, del, add) \in \mathcal{A}_s, b \in \mathcal{B}_{Val}\}$.

3.2.1 Lifted planning example - Shapes World Since the improved translation can handle type hierarchies, the main example in this section will be Shapes World, which is a version of the container stacking task with objects of different shapes. A task in this domain is illustrated in Figures 13 and 14.

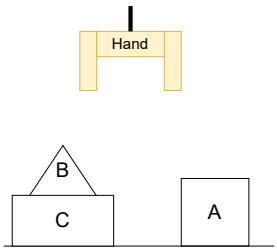


Fig. 13: The initial state.

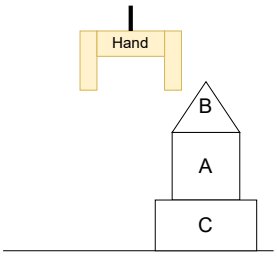


Fig. 14: The goal state.

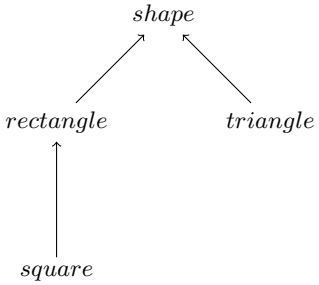


Fig. 15: Type hierarchy.

To represent the domain, we construct the 6-tuple $\Pi = \langle(\mathcal{T}, \leq), \mathcal{O}, \mathcal{X}, \mathcal{P}, \mathcal{P}_{\mathcal{X}}, \mathcal{A}_s\rangle$. The partially ordered set of types $(\mathcal{T}, \leq)$ is illustrated in Figure 15. Here, $square \leq rectangle \leq shape$ and $triangle \leq shape$. As can be seen in Figures 13 and 14, the types of each object in $\mathcal{O} = \{\texttt{A, B, C}\}$ are $typeOf(A) = square$, $typeOf(B) = triangle$, and $typeOf(C) = rectangle$. The set of parameters is $\mathcal{X} = \{x, y\}$ with types $typeOf(x) = shape$ and $typeOf(y) = rectangle$. Shapes World has five predicates $\mathcal{P} = \{\texttt{Clear, OnTable, Holding, On, HandEmpty}\}$ with the first 3 having the signatures $sig(\texttt{Clear}), sig(\texttt{OnTable}), sig(\texttt{Holding}) = shape$, $sig(\texttt{On}) = (shape, rectangle)$, and $sig(\texttt{HandEmpty}) = \epsilon$.

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

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

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

(b) The action schemas in the lifted Shapes World domain.

Fig. 16: The parameterized predicates $\mathcal{P}_\mathcal{X}$ and $\mathcal{A}_s$ of the lifted Shapes World domain displayed across two tables.

An example of how the types are used, is that the predicate `On` has the signature sig (`On`) = ($shape, rectangle$), as we can not stack anything on top of a triangle, but a triangle can be placed on top of rectangles and squares. Following the semantics described under Definition 3, the set of facts constructed by replacing each parameter in `On` x y is {`On A A`, `On A B`, `On B A`, `On B B`, `On C A`, `On C B`}. The set of actions created during grounding will also be impacted by the type restrictions as no actions for stacking something on top of, or picking up, a shape from a triangle would be created.

3.3 Translating Lifted Planning Tasks to Colored Petri Nets

This section presents the translation from a subset of lifted planning tasks to a colored Petri net with set-based semantics as defined in Section 2. The subset is lifted planning as defined in 3.2 but without action schemas containing a specific relationship defined in Section 3.3.2. This section also contains two examples,

each showcasing parts of the translation on an excerpt of the Shapes World domain from Section 3.2.1.

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:

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

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

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

This color type is an ordered set, and its elements are added according to the method showcased in Algorithm 1. The imposed order is later used in guard expressions to enforce the type hierarchy. A second, singleton color type referred to as $dot = \{dot\}$ is used for facts; predicates with $sig(p) = \epsilon$.

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. $\mathbb{V} = \mathcal{X}$, and $typeOf(v) = object$ for $v \in \mathbb{V}$,
4. *Arcs* is constructed as explained in Section 3.3.2,
5. and *Guards* is constructed as explained in Section 3.3.3.

Algorithm 1 : Creating the ordered set of colors

Input: $\langle (\mathcal{T}, \leq), \mathcal{O} \rangle$
Output: Ordered set of colors $\mathbb{C}_\tau$

procedure ConstructColors($(\mathcal{T}, \leq), \mathcal{O}, \tau$)
  $\mathbb{C}_\tau \leftarrow \emptyset$
  Append all $o \in \mathcal{O}$ where $typeOf(o) = \tau$ to $\mathbb{C}_\tau$ ▷ Add objects of type τ

  ▷ Recursively add objects of immediate subtypes
  for $\tau_i \in \mathcal{T}$ where $\tau_i \neq \tau, \tau_i \leq \tau$, and $\neg\exists \tau_j \in \mathcal{T}. \tau_i \leq \tau_j \leq \tau$ **do**
    Append ConstructColors($(\mathcal{T}, \leq), \mathcal{O}, \tau_i$) to $\mathbb{C}_\tau$
  end for

  return $\mathbb{C}_\tau$
end procedure

$\tau_{max} \leftarrow$ maximal element of $\mathcal{T}$
return ConstructColors($(\mathcal{T}, \leq), \mathcal{O}, \tau_{max}$)

3.3.1 Constructing the color type *object*, example - Shapes World

Let $(\mathcal{T}, \leq)$ be the type hierarchy in Figure 17 and $\mathcal{O} = \{shape1, triangle1, rec1, rec2, square1\}$. The ordered set of colors being constructed by Algorithm 1 is shown in Figure 18.

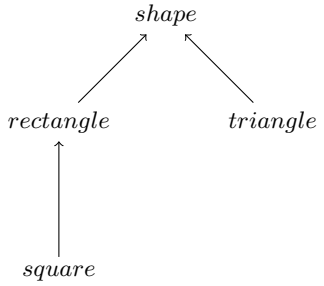


Fig. 17: Type hierarchy

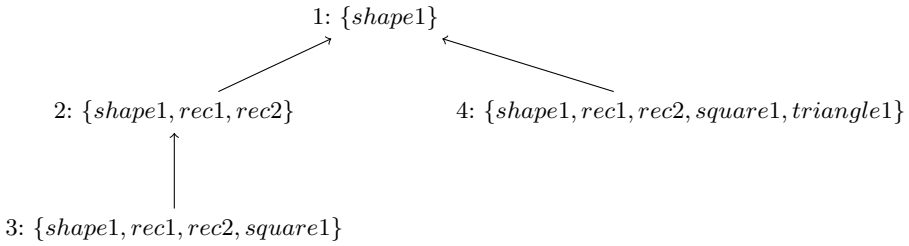


Fig. 18: The ordered set of colors being built by Algorithm 1. The numbers indicate the order it traverses the type hierarchy.

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

1. The require-relationship. If $(p_x \in pre) \wedge ((p_x \notin del) \vee (p_x \in del \cap add))$, then update $Arcs$ to:

$$Arcs'(p,t) = Arcs(p,t) + (x_1, ..., x_n)$$
$$Arcs'(t,p) = Arcs(t,p) + (x_1, ..., x_n)$$

.

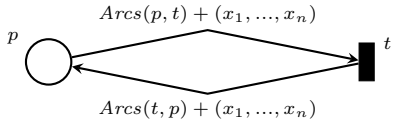


Fig. 19: The require-relationship

2. The consume-relationship. If $(p_x \in pre) \wedge (p_x \in del) \wedge (p_x \notin add)$, then update $Arcs$ to:

$$Arcs'(p,t) = Arcs(p,t) + (x_1, ..., x_n)$$

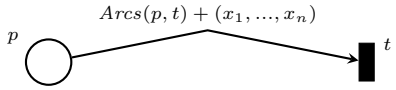


Fig. 20: The consume-relationship

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

$$Arcs'(t,p) = Arcs(t,p) + (x_1, ..., x_n)$$

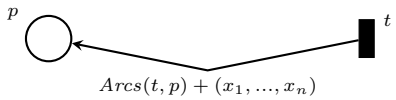


Fig. 21: The add-relationship

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.

3.3.3 Guard Expressions As described in Section 3.2, there are type restrictions regarding which objects, $o \in \mathcal{O}$, a parameter, $x \in \mathcal{X}$, 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 $validColors : \mathcal{X} \to 2^{\mathbb{C}_{atom}}$ returns an ordered set of colors corresponding to the objects that are valid for the parameter, using the fact that $\mathbb{C}_{atom} = \mathcal{O}$:

$$validColors(x) = \{o \mid o \in \mathbb{C}_{atom}, typeOf(o) \leq typeOf(x)\}$$

Let $parametersIn : \mathbb{AE} \to 2^{\mathcal{X}}$ return all parameters corresponding to the variables that occur inside a given arc expression. Let $allAE : \mathbb{T} \to 2^{\mathbb{AE}}$ be 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:

$$\bigwedge_{ae \in allAE(t)} \bigwedge_{x \in parametersIn(ae)} x \geq c_1 \wedge x \leq c_n$$

where $c_1 = min(validColors(x))$ and $c_n = max(validColors(x))$

3.3.4 Partial Translation Example - Shapes World In this example, the part of a Shapes World translation corresponding to the action schema Stack x y from the example in Section 3.2.1 is showcased. The action schema to be translated can also be seen in Figure 22 and the CPN being constructed is shown in Figure 23. Let the type hierarchy and objects be the same as the example in Section 3.3.1.

The action schema contains 4 predicates which are turned into places, and the action schema itself is modeled as a transition. The types of the 2 parameters are $typeOf(x) = shape$ and $typeOf(y) = rectangle$. These are turned into variables with types $typeOf(x), typeOf(y) = object$. The types of their original parameter counterparts are used when the guard expression is created.

Stack x y**:** $\{(pre, del, add)\}$ where $pre = \{$`Holding` x, `Clear` $y\}$, $del = \{$`Holding` x, `Clear` $y\}$, $add = \{$`Clear` x, `HandEmpty`, `On` x $y\}$	Stack x on y.

Fig. 22: The action schema for Stack x y.

Using the method from Section 3.3.2 to create the arc expressions, we note that `Holding` x and `Clear` y exhibit a consume-relationship with the transition. The parameterized predicates `Clear` x, `On` x y, and `HandEmpty` all fall into the category called add-relationships. Since Algorithm 1 was already used in the prior example to create the color type $object = \{shape1, rec1, rec2, square1, triangle1\}$, use this and the types of the parameters corresponding to variables x and y to determine the minimum and maximum color of each type:

$$shape1 = min(validColors(x)), triangle1 = max(validColors(x)),$$

$$rec1 = min(validColors(y)), square1 = max(validColors(y))$$

Following the method from Section 3.3.3 results in the guard expression seen below in Figure 23.

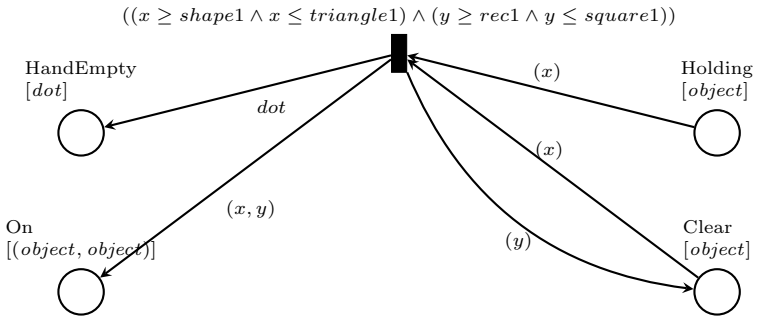


Fig. 23: The Shapes World excerpt corresponding to the action schema Stack x y and connected predicates as a CPN.

3.4 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 $translate : 2^F \rightarrow (\mathbb{P} \rightarrow 2^{\mathbb{C}})$.

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

3.5 Correctness

3.5.1 Finding a Plan with CPNs The function $translate : (\mathbb{T} \times (\mathbb{V} \times \mathbb{C}_{atom}))^* \rightarrow (\mathcal{A}_s \times (\mathcal{X} \times \mathcal{O}))^*$ 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, $\mathbb{T} = \mathcal{A}_s$, $\mathbb{V} = \mathcal{X}$ and $\mathbb{C}_{atom} = \mathcal{O}$, so the translation from a trace to a plan is the identity function.

Lemma 1. *If* $s \xrightarrow{a_s,b} s'$, *then* $translate(s) \xrightarrow{t_a,b'} translate(s')$ *if* a *does not have the delete-relationship.*

Lemma 2. *Let* $M = translate(s)$. *If* $M \xrightarrow{t,b} M'$ *then* $\exists a_s \in \mathcal{A}_s, b' \in \mathcal{B}_{Val}$ *s.t.* $s \xrightarrow{a_s,b'} s'$ *and* $M' = translate(s')$.

Transitions are constructed exactly such that Lemma 1 and 2 hold.

Theorem 1. *Given a lifted planning task* Π *with no delete-relationships with initial state* s_0, *then there exists a sequence of actions* $s_0 \xrightarrow{w} s$ *iff. for the translated CPN there exists a sequence of transition firings* $translate(s_0) \xrightarrow{w} M'$.

Proof. sketch
To prove Theorem 1, we first show that the existence of a plan implies the existence of a trace, and next that the existence of a plan implies the existence of a trace.

1. *The empty sequence is trivially both a plan and trace,*
2. *Lemma 1 implies that given a plan* $s_0 \xrightarrow{(a_s,b)_1} s_1 ... \xrightarrow{(a_s,b)_n} s_n$, *and no* a_s *has the delete-relationship, then there exists a trace* $translate(s_0) \xrightarrow{(t,b)_1} translate(s_1) ... \xrightarrow{(t,b)_n} translate(s_n)$
3. *Lemma 2 implies that given some state* s_0 *and trace* $translate(s_0) \xrightarrow{(t,b)_1} M_1 ... \xrightarrow{(t,b)_n} M_n$, *then there exists a plan* $s_0 \xrightarrow{(a_s,b)_1} s_1 ... \xrightarrow{(a_s,b)_n} s_n$, *where* $translate(s_1) = M_1, ..., translate(s_1) = M_n$

3.6 Implementation

This section showcases key implementation details. The most significant of these is the implementation of the translation itself, but the tool for validating Petri net traces as plans, and the test suite used for running experiments are also noteworthy.

3.6.1 Translations The translation from Section 3.3 is a proposed improvement to the one from our earlier work [7]. Therefore, the new implementation is also based on the that of the earlier translation. It is written in Python 3.9.2 and can be found on Github[1]. The added support for type inheritance required significant rewriting of the original code and additional implementations of particularly the type hierarchy data structure, Algorithm 1 for structuring the colors correctly, and the code generation of guard expressions.

In addition to the new translation, multiple implementation issues plaguing the old translation have been solved. These improvements alone have increased the number of domains the old translation could translate from 8 to 14.

[1] https://github.com/petrinet-planning/colored-lifted-planning/tree/TypeHierarchyV2

3.6.2 Plan validation To ensure that the implementation of the translation from Section 3.3 and the earlier implementation from our first work on the subject [7] is correct, we systematically ensure the found traces are solutions to the original planning problem with the plan validation system called VAL[2]. This tool takes a plan in a format called `sasplan` as input, while TAPAAL produces traces that are structured quite differently and refer to the unfolded transitions instead of their colored counterparts. To combat this, a parser was written for the traces, and an algorithm that goes through each step in the trace to restructure the bindings of each variable s.t. they are in the same order as in the original planning task was implemented.

With this validation tool setup, 30 randomly selected tasks from each viable domain were run, the traces translated to plans, and these plans were validated by VAL. The results were **550 valid plans** across both translations and **0 invalid plans**, which is valuable evidence that the translations and their implementations work as intended.

3.6.3 Test Suite The experiments were run and parsed through a testing suite initially developed in our first work on this subject [7]. The test suite has been enhanced greatly during the writing of this report to facilitate better ease of use and support for a wider range of configurations. The current version of the test suite is available on Github[3].

3.7 Experiments setup

To test the performance of the method, the implemented translation has been run on several benchmarks using different configurations. The experiments were run on Aalborg University's "naples" computer cluster, which consists of AMD EPYC 7551 CPUs. Each experiment was run with a timeout of 4 hours and 16 GiB of RAM.

The 4 configurations that translate a subset of lifted planning to CPNs used for the experiments are:

- Color, the implementation of the translation from Section 3.3,
- Color$_{NT}$, Color with the VerifyPN option disabling timeouts when calculating color fix-points (NT),
- OldColor, the implementation of the translation from Ginnerup H., Lassen S. [7].
- OldColor$_{NT}$, OldColor with the NT option.

As OldColor does not support planning domains that use type hierarchies with inheritance, it will naturally solve fewer tasks than Color but is included as it

[2] https://github.com/KCL-Planning/VAL
[3] https://github.com/petrinet-planning/benchmarking

is the only other implementation of a translation from lifted planning task to CPN. Additionally, it can also be used to highlight the larger set of domains translatable by the new translation and potentially signal if parts of the newer translation are significantly slower on domains supported by both. All of these configurations were run using a version of VerifyPN[4], the verification tool of TAPAAL, modified to use set-based semantics. All of the configurations were only run with options deemed sound when used with set-based CPNs in Section 2.3, and with the search strategy Random Potency First Search (RPFS)[12].

The other methods that the translations are compared against are:

- Grounded, a translation of a grounded planning task to P/T net,
- FD, the planning system Fast Downward.

The configuration of Grounded also utilizes the search strategy RPFS but is run using VerifyPN with multiset-based semantics. Therefore, it can also utilize all the structural reductions available. The FD configuration is `lama-first`[21], which is a powerful non-optimal planning strategy[6].

3.7.1 Benchmarks To measure the performance of the different methods and configurations, the Autoscale benchmark suite from Torralba et al. [23] designed for agile and satisficing planning has been used. The suite contains 42 planning domains, each containing 30 planning tasks of, generally, increasing difficulty. Some domains were not run for various reasons, which can be seen in the breakdown below:

- 19 domains included, with 5 using type-inheritance,
- 14 domains excluded for using the delete-relationship from Section 3.3.2,
- 7 domains excluded as Unified Planning[5] fails to parse the domain,
- 2 domain excluded for using negative preconditions.

Note that the method used to assess the domains starts by attempting to parse these with the Unified Planning library, then looks for the unsupported relationship, before finally reporting any other issues. Therefore, even if an issue that excludes n domains is fixed, these may still fall into other issues.

3.8 Experiments

In this section, the results of running experiments using the configurations from Section 3.7 are showcased and discussed. Each configuration is run with a sample size of 10 on each task, and the median times are reported, as the RPFS search strategy used for every Petri net-based approach contains significant random behavior[12]. The reported median times are always with everything included, from the initial call of the translation until the search concludes. If this is not the case, it is always explicitly stated at the top of the plot.

[4] https://github.com/TAPAAL/verifypn/tree/53b047edca7a880135c29fd1659fc6ddccebe057
[5] https://github.com/aiplan4eu/unified-planning

	Color	Color$_{NT}$	OldColor	OldColor$_{NT}$	Grounded	FD
Plans Found	221	268	154	159	360	**469**
Unique Plans	0	0	0	0	12	**130**
Fastest Response	0	0	1	2	137	**356**

Table 1: Overall comparison between all configurations. The total number of tasks is 570. As OldColor does not support type inheritance, it can only compete on 420 of these.

As showcased in Table 1, the dedicated planning system FD significantly outperforms all of the Petri net-based methods. The Grounded method which utilizes the grounding process of Fast-Downward[11] on the lifted planning task also solves nearly a hundred more tasks compared to the best performing CPN-based configuration, Color$_{NT}$. As there are six configurations in the same table, there is hard competition for the number of unique and fastest plans. However, when comparing just the overall performance of Color$_{NT}$ and Grounded in Table 2, it is clear that Grounded still significantly outperforms wrt. speed and the number of plans found.

	Color$_{NT}$	Grounded
Plans Found	268	**360**
Unique Plans	3	**95**
Fastest Response	36	**327**

Table 2: Overall comparison between our best-performing configuration and Grounded. The total number of tasks is 570.

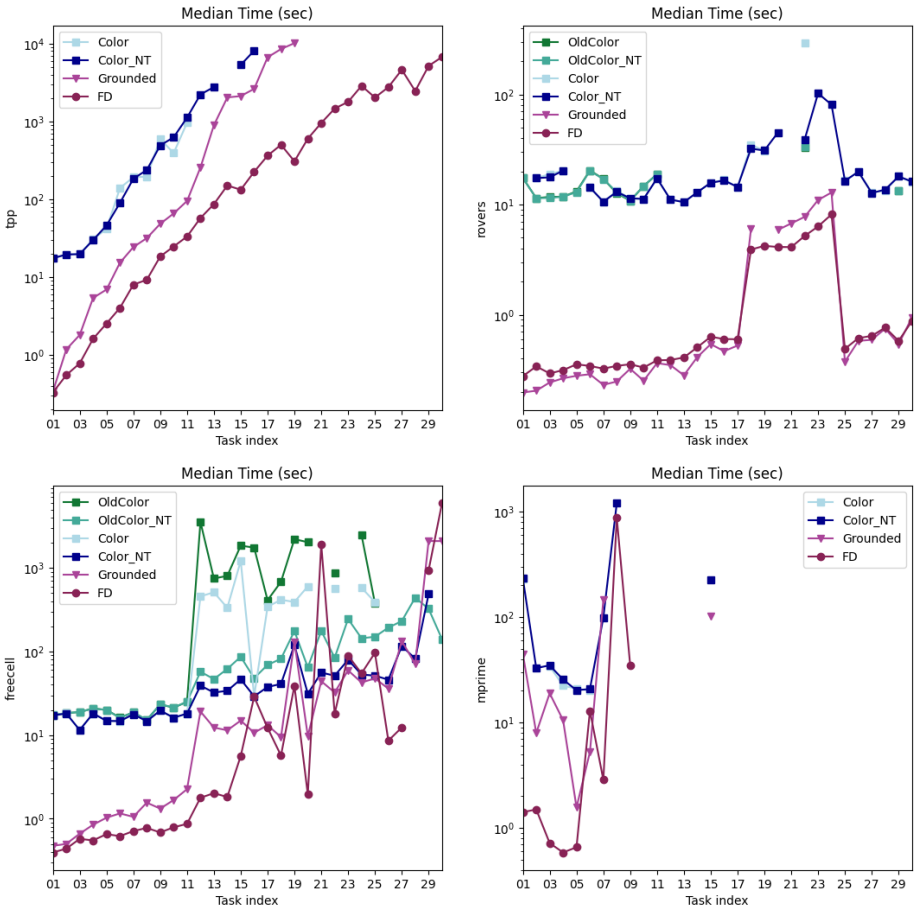


Fig. 24: Typical median times shown across 4 domains.

When examining the results for the individual domains, the trend indicated by the overall results is generally confirmed, with most of the results being similar to those shown in Figure 24. However, there is also a large part of the domains where the Color configurations perform almost identically to the Grounded configuration.

Examples of these can be seen in Figure 25, which also includes the domain ged where especially Color_{NT} consistently performs better than Grounded. When solving planning problems with CPNs, it is generally an advantage for the configurations to have the NT option, with it only performing worse in a few cases such as driverlog in Figure 25. As can be seen in Figure 25, it is particularly effective in domains such as nomystery where the translated CPN contains a huge amount of colors.

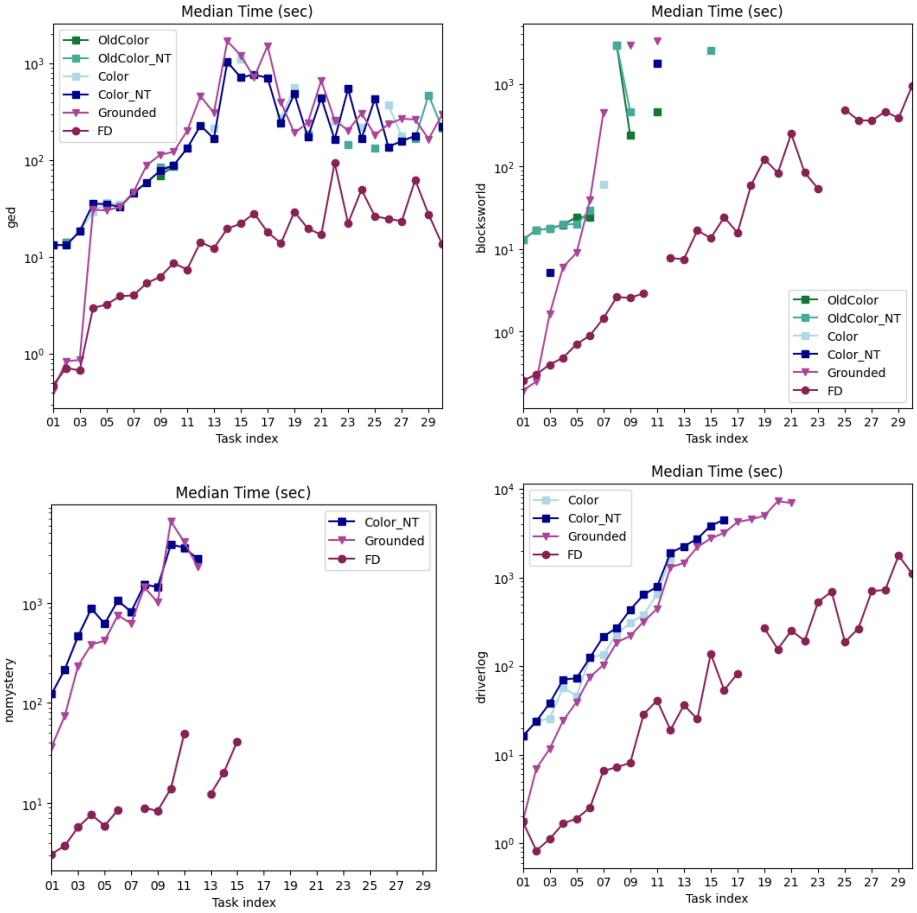


Fig. 25: The most competitive median times between Grounded and the Color configurations

Another noticeable trend in the domains with very low median times, is that the translation time of every CPN-based configuration is an order of magnitude slower than both FD and Grounded's translation times. This can be observed in Figure 26, and is likely a side-effect of the implementation being in Python whereas Fast Downward and Grounded are written in C++. Intuitively, it should be computationally cheaper to translate a high-level representation such as that of a lifted planning task to a CPN, which is also high-level, than ground the lifted task and then translate it to a P/T net, which is what Grounded does. In Figures 26 and 27, the translation time of Color remains almost constant with minor fluctuations which also hints at the increasing complexity of the tasks in the domain not being the deciding factor in the translation speed as it is for Grounded and FD.

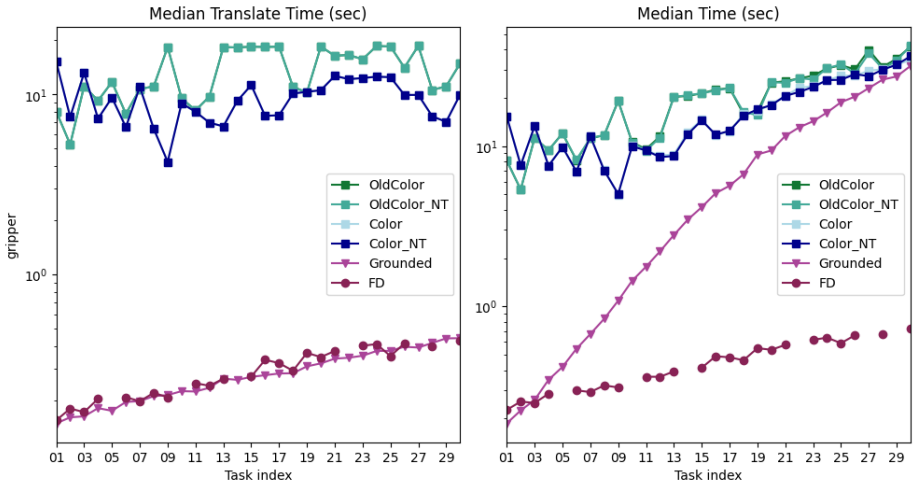


Fig. 26: The CPN-based configurations' translation times take up a substantial part of the overall time.

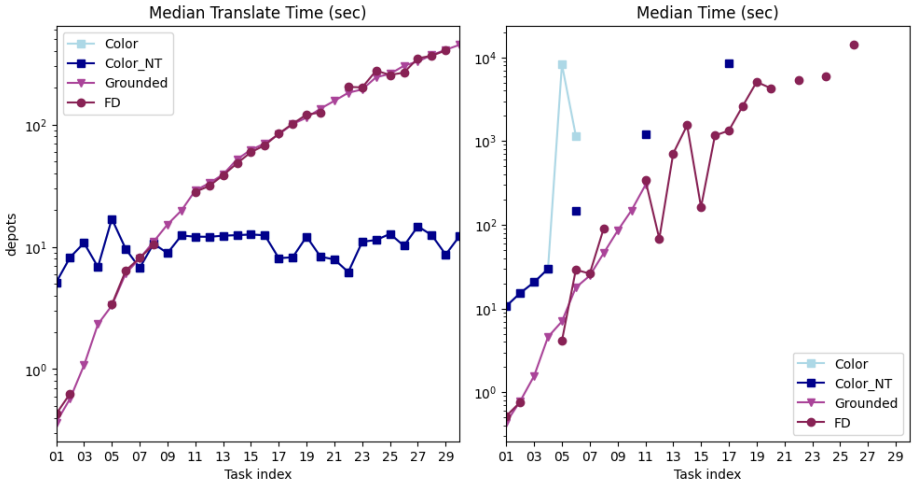


Fig. 27: The translation time and overall time of the depots domain.

Besides the translation time, other deciding factors when comparing the results of the different configurations are:

- **Color vs OldColor**: Color uses guard expressions to uphold type restrictions whereas OldColor uses arc expressions with variables of different color types. When examining the domains where both appear, it does not appear that Color is substantially slower, and since it supports a larger subset of lifted planning domains, it is, therefore, clearly preferred.

- **Color vs Grounded**: As both use the same search strategy, a major difference in approach is the translated CPN being unfolded whereas Grounded takes advantage of Fast Downward's grounding algorithm. For some domains it is close, but since Grounded solves substantially more tasks than Color, it seems FD's grounding is better at reducing planning tasks than the unfolding techniques and colored reductions utilized by Color.

- **Grounded vs FD**: As both use the same grounding algorithm, the major differences are search-related. Here, FD generally outperforms with its `lama-first` configuration, though Grounded *does* solve about a third of its solves tasks faster than FD.

The full collected data along with plots for every domain are available online[6].

[6] https://drive.google.com/drive/folders/16JyjidUK_O2r7ulyFS1mNJ6Dnqha0f5M

4 Colored Petri Nets to Lifted Numeric Planning Tasks

This section defines lifted numeric planning, shows how it is translated to a multiset based CPN, argues for the correctness of the translation, and then discusses the implementation of the translation.

4.1 Grounded numeric planning

Numeric planning extends the definition of classical grounded planning from Section 3.1 with a finite set of numeric variables $\mathcal{V}$ [14][9], along with conditions on these, and effects that increase or decrease their assigned value. For many planning tasks, it is convenient to represent parts as numeric variables. An example is a variation of the Traveling Salesman Problem displayed in Figure 28, where driving between locations costs fuel, which is best represented as a numerical variable that is continually decreased by performing drive-actions.

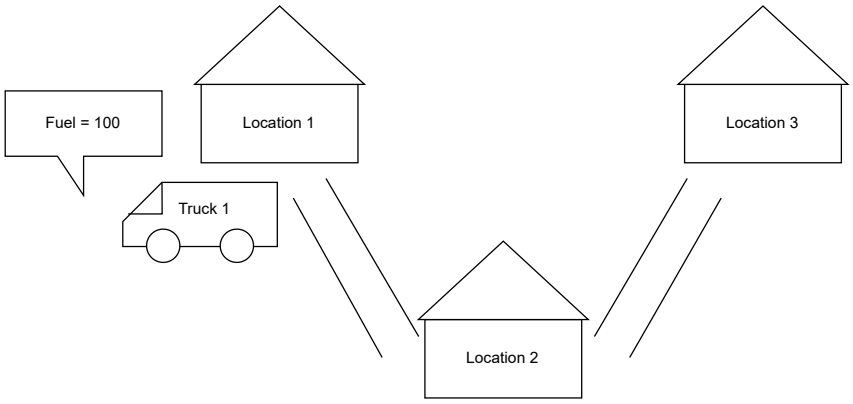


Fig. 28: A Traveling Salesman Problem with fuel modeled as a numeric variable.

In this section, we define the syntax and semantics of numeric planning as an extension to Definition 2. Therefore, when the numeric extension is combined with the definition of a grounded planning task, the syntax and semantics of the original elements are exactly as previously defined in Section 3.1.

Numeric states A numeric state s_{num} is defined as a function that assigns an integer to each numeric variable $v \in \mathcal{V}$. Let S_{num} be the infinite set of every such function $s_{num} : \mathcal{V} \to \mathbb{Z}$.

Numeric conditions The set $\mathcal{C}$ denotes all numeric conditions of the form:

$$c ::= \text{LHS} \ \bowtie \ \text{RHS}$$

where $\bowtie = \{<, \leq, =, \neq, \geq, >\}$, and LHS and RHS are arithmetic expressions over numeric variables $v \in \mathcal{V}$ or constants $n \in \mathbb{Z}$.

We write $s_{num} \models c$ if the state satisfies the numeric condition. This is determined by replacing each variable v in the condition with its value in the state $s_{num}(v)$ and evaluating c as a boolean expression.

Definition 4. *A numeric extension to a grounded planning task is a 4-tuple* $\Pi_{num} = \langle \mathcal{V}, s_{num0}, G_{num}, \mathcal{A}_{num} \rangle$ *where:*

1. $\mathcal{V}$ *is a finite set of numeric variables,*
2. $s_{num0} \in S_{num}$ *is an initial numeric state,*
3. $G_{num} \subseteq \mathcal{C}$ *is a finite set of required, numeric goal conditions, and*
4. $\mathcal{A}_{num} \subseteq 2^{\mathcal{C}} \times 2^{EFF_{num}}$ *is a finite set of numeric actions where* EFF_{num} *is the set of all numeric effects of the form:*

$$\mathit{eff}_{num} ::= v \ + \ n$$

where $v \in \mathcal{V}$ *and* $n \in \mathbb{Z}$*. There is exactly one effect for each variable. Effects with* $n = 0$ *are not explicitly noted.*

The semantics of the numeric extension is, like its classical counterpart in Section 3.1, defined as a transition system that is traversed by applying actions.

To apply a numeric action, the numeric preconditions must be satisfied in the given numeric state. We write $s_{num} \xrightarrow{a_{num}} s_{num}{}'$ if $\exists a_{num} = (pre_{num}, \text{eff}_{num}) \in \mathcal{A}_{num}$ s.t. $\forall c \in pre_{num}.s_{num} \models c$. The numeric effects of an action are $s_{num}{}' = \forall v \in \mathcal{V}.s_{num}{}'(v) = s_{num}(v) + n$ where $v + n \in \text{eff}_{num}$.

Enhancing the grounded planning definition from Section 3.1 with a numeric extension yields a grounded numeric planning task, defined as a 7-tuple $\Pi = \langle F, \mathcal{V}, s_0, s_{num0}, G, G_{num}, \mathcal{A} = \mathcal{A}_{classic} \times \mathcal{A}_{num} \rangle$. A valid plan π for a numerical planning task is a sequence of actions $\omega \in \mathcal{A}^*$ s.t. $(s_0, s_{num0}) \xrightarrow{\omega} (s_{classic}, s_{num})$, $G \subseteq s_{classic}$, and $s_{num} \models G_{num}$.

4.1.1 Grounded numeric planning example - Driving In this small numeric planning task, the goal is to drive a truck to a desired location. The numeric components of the task are the fuel level of the truck, which is modeled as a numeric variable, and the numeric precondition and effect of the **Drive** actions.

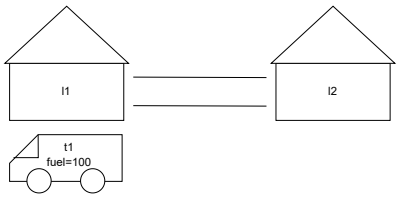


Fig. 29: The initial state.

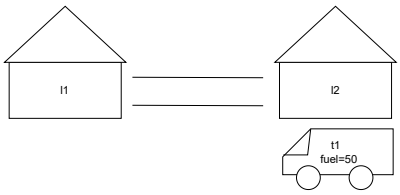


Fig. 30: A goal state.

The facts and numeric variables are shown in Figure 31a, the actions in Figure 31b, and the initial facts, initial numeric state, goal facts, and numeric goal conditions in Figure 31c.

Facts $F =$	**Description**
{`At l1 t1, At l2 t1,`	The position of truck t1
`Connected l1 l2, Connected l2 l1`}	The connections of locations
Numeric variables $\mathcal{V} =$	
{`fuel t1`}	The fuel level of truck t1

(a) The facts and numeric variables in the grounded numeric planning task

Actions $\mathcal{A} =$	**Description**
Drive t $loc1$ $loc2$ **:** $\{(pre, del, add, \text{pre}_{num}, \text{eff}_{num}) \mid t \in \{t1\} \wedge loc1, loc2 \in \{l1, l2\}$ where	
$pre = \{$`At` $loc1$ t, `Connection` $loc1$ $loc2\}$,	Drive truck t
$del = \{$`At` $loc1$ $t\}$,	from location
$add = \{$`At` $loc2$ $t\}$	$loc1$ to $loc2$
$\text{pre}_{num} = \{$`fuel` $t \geq 50\}$	
$\text{eff}_{num} = \{$`fuel` $t + (-50)\}$	

(b) The actions in the grounded numeric planning task.

Initial facts $s_0 =$	**Description**
{`At l1 t1,`	See Figure 29
`Connected l1 l2, Connected l2 l1`}	
Initial numeric state $s_{num0} =$	
s_{num0}(`fuel t1`) $= 100$	See Figure 29
Goal facts $G =$	
{`At l2 t1`}	See Figure 30
Numeric goal conditions $G_{num} =$	
$\emptyset$	None

(c) The start facts, initial numeric state, goal facts, and numeric goal conditions in the grounded numeric planning task.

Fig. 31: The full grounded numeric planning task $\Pi = \langle F, \mathcal{V}, s_0, s_{num0}, G, G_{num}, \mathcal{A} \rangle$ displayed across three tables.

The solution to this planning task is the action sequence:

$$\pi = \texttt{Drive l1 l2 t1}$$

The plan traverses the following states:

$$(\{\texttt{At l1 t1, Connected l1 l2, Connected l2 l1}\}, (s_{num}(\texttt{fuel t1}) = 100)) \xrightarrow{\texttt{Drive l1 l2 t1}}$$
$$(\{\texttt{At l2 t1, Connected l1 l2, Connected l2 l1}\}, (s_{num}(\texttt{fuel t1}) = 50)) \quad \textit{where}$$

$\{\texttt{At l2 t1, Connected l1 l2, Connected l2 l1}\} \supseteq \{\texttt{At l2 t1}\}$, and
$(s_{num}(\texttt{fuel t1}) = 50) \models \emptyset$

4.2 Lifted numeric planning

As in the previous Section **??**, the numeric part of a lifted numeric planning domain is an extension to the lifted domain in Definition 3. Here, the combined tuple of a lifted numeric planning domain is defined along with the semantics of how the numeric part is grounded, as the classical part is exactly the same as in Section 3.2.

Definition 5. *A lifted numeric planning domain is a 6-tuple* $\Pi = \langle (\mathcal{T}, \leq), \mathcal{O}, \mathcal{X}, \mathcal{P}, \mathcal{P}_{\mathcal{X}}, \mathcal{A}_s \rangle$ *where:*

1. $(\mathcal{T}, \leq)$ *is a finite, partially ordered 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. $\mathcal{X}$ *is a finite set of parameters, each with a type indicated by* $typeOf : \mathcal{X} \to \mathcal{T}$,
4. $\mathcal{P} = \mathcal{P}_{classic} \uplus \mathcal{P}_{num}$ *is a finite set of classical and numeric first-order predicates, each with an n-tuple of types indicated by the signature* $sig : \mathcal{P} \to \mathcal{T}^*$,
5. $\mathcal{P}_{\mathcal{X}} = \mathcal{P}_{classic\mathcal{X}} \uplus \mathcal{P}_{num\mathcal{X}}$ *is the set of all parameterized classical and numeric predicates* $p(x_1, ..., x_n)$ *s.t.* $p \in \mathcal{P}$, $x \in \mathcal{X}_i \cup \mathcal{O}$, *and* $\forall i \in (1...n).typeOf(x_i) \leq \tau_i$ *where* $sig(p) = (\tau_1, ..., \tau_n)$, *and*
6. $\mathcal{A}_s \subseteq 2^{\mathcal{P}_{classic\mathcal{X}}} \times 2^{\mathcal{P}_{classic\mathcal{X}}} \times 2^{\mathcal{P}_{classic\mathcal{X}}} \times 2^{PRE_{num}} \times 2^{EFF_{num}}$ *is a finite set of action schema where:*
   PRE_{num} *is the set of all numeric preconditions of the form:*

$$pre_{num} ::= p_{num\mathcal{X}} \ \bowtie \ n$$

   where $p_{num\mathcal{X}} \in \mathcal{P}_{num\mathcal{X}}, \bowtie = \{<, \leq, =, \geq, >\}$, *and* $n \in \mathbb{Z}$,

   and EFF_{num} *is the set of all numeric effects of the form:*

$$eff_{num} ::= p_{num\mathcal{X}} \ + \ n$$

   where $p_{num\mathcal{X}} \in \mathcal{P}_{num\mathcal{X}}$ *and* $n \in \mathbb{Z}$

To construct a grounded numeric planning task $\Pi = \langle F, \mathcal{V}, s_0, s_{num0}, G, G_{num}, \mathcal{A} \rangle$ from the lifted numeric planning domain, we start by noting that both the initial numeric state s_{num0} and the numeric goal conditions G_{num} are always given in grounded form. To construct the remaining numeric elements of the tuple, as was done in Section 4.2 with the classical elements, we use the valid bindings to ground each numeric element. The set of numeric variables is constructed by replacing each parameter of every parameterized numeric predicate by valid objects $\mathcal{V} = \{p[b] \mid p \in \mathcal{P}_{num\mathcal{X}}, b \in \mathcal{B}_{Val}\}$. Similarly, the set of actions can be constructed by replacing each parameter of every parameterized, numeric predicate in each of its elements by a valid object $\mathcal{A}_{num} = \{\text{pre}_{num}[b], \text{eff}_{num}[b]) \mid (\text{pre}_{num}, \text{eff}_{num}) \in \mathcal{A}_s, b \in \mathcal{B}_{Val}\}$.

4.2.1 Preconditions as boolean expressions To consider preconditions as boolean expressions, note that they are defined as sets of bound parameterized predicates that must all evaluate to true, by being in state s, for an action to be applicable. Therefore, a precondition can intuitively be written in the form:

$$pre := p_{\mathcal{X}} \in \mathcal{P}_{\mathcal{X}} \mid pre_1 \wedge pre_2$$

This can then be extended to allow *or* ($\vee$) in the form:

$$pre^{or} := pre \mid pre_1^{or} \vee pre_2^{or}$$

An action schema with the precondition $\phi_1 \vee ... \vee \phi_n$ is then equivalent to having n action schemas $a_{s1}...a_{sn}$ where the precondition of a_{sn} is ϕ_n. This way, a lifted precondition can be expressed in disjunctive normal form without negation.

4.3 Translating a Colored Petri Net to a Lifted Numeric Planning Task

This section shows the general translation from a multiset-based colored Petri net to a lifted numeric planning task. For this, let $\mathcal{N}$ be some fixed CPN with colors $\mathbb{C}$, types $\mathbb{C}_T$ and initial marking M_0.

$$\text{CPN } \mathcal{N} = (\mathbb{P}, \mathbb{T}, \mathbb{V}, \mathit{Arcs}, \mathit{Guards})$$

The translated, numeric lifted planning domain $\Pi = \langle (\mathcal{T}, \leq), \mathcal{O}, \mathcal{X}, \mathcal{P}, \mathcal{P}_{\mathcal{X}}, \mathcal{A}_s \rangle$ is constructed as follows:

- Let $\mathcal{T} = \mathbb{C}_{T_{atom}} \cup \{\mathit{objects}\}$, where every $t \in \mathbb{C}_{T_{atom}}$ is an immediate subtype of $\mathit{objects}$,
- $\mathcal{O} = \mathbb{C}_{atom}$ and $(\mathit{typeOf} : \mathcal{O} \to \mathcal{T}) = (\mathit{typeOf} : \mathbb{C}_{atom} \to \mathbb{C}_{T_{atom}})$,
- $\mathcal{X} = \mathbb{V}$ initially, and then more parameters are generated and added during the construction of Action Schemas in Section 4.4,
- $\mathcal{P}$ and $\mathcal{P}_{\mathcal{X}}$ are constructed below, followed by the construction of $\mathcal{A}_s$.

The translation is shown on the concrete CPN $\mathcal{N}_{ex}$ in Figure 32, but using the general method for any CPN.

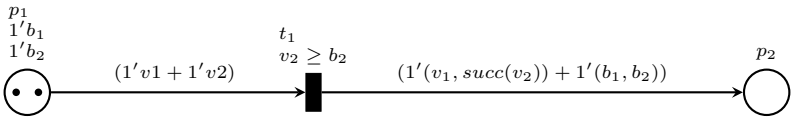


Fig. 32: A CPN where $\mathit{typeOf}(p_1) = C$ and $\mathit{typeOf}(p_2) = (C, C)$, and $\mathit{typeValues}(C) = \{b_1, b_2\}$.

4.3.1 Predicates Predicates are used for two purposes in the translated planning task:

1. Maintaining the partial order of colors, after they are mapped to unordered planning objects, and
2. A numeric lifted predicate is constructed for each place to represent the marking.

First, to maintain the partial order of colors, let $isPredecessor$ and $isSuccessor$ be classical lifted predicates with signatures of $(objectsobjects)$, and $getIndex$ be a numerical lifted predicate with a signature of $(objects)$. These predicates will be in any translated CPN.

Secondly, to represent the marking, a numeric predicates is constructed for each place. For the example $\mathcal{P}$ contains $\{(isPredecessor\ objects\ objects), (isSuccessor\ objects\ objects), (getindex\ objects)\}$ as it will for any translation, but then to represent any marking, it also has the numeric predicates $(p1\ C)$ and $(p2\ C\ C)\}$ to represent the marking of p_1 and p_2 respectively.

For the general case, construct a unique, numeric, lifted predicate with a signature of $typeOf(p)$ for every place $p \in \mathbb{P}$. Let the constructed predicate be referred to through the function:

$$getPred : \mathbb{P} \to \mathcal{P}_{num\mathcal{X}}$$

$$\mathcal{P} = \{getIndex, isPredecessor, isSuccessor\} \cup \{getPred(p) \mid p \in \mathbb{P}\}$$

Parameterized predicates $\mathcal{P}_{\mathcal{X}}$ is constructed as defined in Definition 3 of lifted numerical planning.

4.3.2 Translating a Marking to a State In this translation a state represents a marking as well as the partial order of the colors that have become objects.

To translate the order of colors, let $index : \mathbb{C}_{atom} \to \mathbb{Z}^{\geq 0}$ be the index of a color within its type, s.t. given $typeValues(\tau) = \{c_1, ...c_n\}$, then $index(c_1) = 1, ..., index(c_n) = n$.

Then the initial state of the example is $s_0 = (s_{classic0}, s_{num0})$ where:

$$
\begin{array}{ll}
s_{classic0} = \{ & \\
\quad (isPredecessor\ \ b1\ b2), & s_{num0}((getIndex\ b1)) = 1 \\
\quad (isPredecessor\ \ b2\ b1), & s_{num0}((getIndex\ b2)) = 2 \\
\quad (isSuccessor\ \ \ b2\ b1), & s_{num0}((p1\ b1)) = 1 \\
\quad (isSuccessor\ \ \ b1\ b2) & s_{num0}((p1\ b2)) = 1 \\
\} & s_{num0}((p2\ b1\ b1)) = 0 \\
 & s_{num0}((p2\ b1\ b2)) = 0 \\
 & s_{num0}((p2\ b2\ b1)) = 0 \\
 & s_{num0}((p2\ b2\ b2)) = 0
\end{array}
$$

For the general case, as the order of colors never changes, let $(s_{classic}[\bullet], s_{num}[\bullet])$ be *any* reachable state of the translated problem, then:

$$
\begin{aligned}
s_{classic}[\bullet] = \{\ & (isPredecessor\ c_{pred}\ c) \mid c \in C, c_{pred} = pred(c)\ \} \cup \\
\{\ & (isSuccessor\ c_{succ}\ c) \mid c \in C, c_{succ} = succ(c)\ \}
\end{aligned}
$$

And for every atomic color $c \in \mathbb{C}_{atom}$:

$$s_{num}[\bullet]((getIndex\ c)) = index(c)$$

Given some marking M, let $s_{num}[M]$ be the translated marking, s.t. for every place $p \in \mathbb{P}$ and every color thereof $(c_1, ..., c_n) \in typeValues(typeOf(p))$, let $pred = getPred(p)$ then:

$$s_{num}[M]((pred\ c_1\ ...\ c_n)) = M(p)((c_1, ..., c_n))$$

The function $s_{num}[M] : \mathcal{V} \to \mathbb{Z}$ is well-defined because $v \in \mathcal{V}$ can either be the predicate $getIndex$ with an atomic color, which is defined for *any* state $s_{num}[\bullet]$, or v can be a place-predicate with a color, which is defined explicitly for $s_{num}[M]$.

The function $translate : (\mathbb{P} \to (\mathbb{C} \to \mathbb{Z}^{\geq 0})) \to (S_{classic} \times S_{num})$ translates any marking of the CPN to a state, defined as:

$$translate(M) = (s_{classic}[\bullet], s_{num}[M])$$

Finally, the initial state s_0 is the translation of the initial marking M_0:

$$s_0 = translate(M_0)$$

4.4 Action Schemas

Each transition $t \in \mathbb{T}$ is translated to an individual action schema $a_s \in \mathcal{A}_s$ through Algorithm 2, which is split into multiple blocks with the stepwise translation of t_1 from Figure 32.

We start with a new, empty action schema $a_{st_1} = ($

$$
\begin{aligned}
&\mathit{preconditions} && = \emptyset,\\
&\mathit{add} && = \emptyset,\\
&\mathit{del} && = \emptyset,\\
&\mathit{numericPrecondition} && = \emptyset,\\
&\mathit{numericEffects} && = \emptyset
\end{aligned}
$$

$)$

Algorithm 2 : Construct Action Schemas

Input: CPN $(\mathbb{P}, \mathbb{T}, \mathbb{V}, \mathit{Arcs}, \mathit{Guards})$
Output: A set of Action Schemas $\mathcal{A}_s \subseteq 2^{\mathcal{P}_\mathcal{X}} \times 2^{\mathcal{P}_\mathcal{X}} \times 2^{\mathcal{P}_\mathcal{X}} \times 2^{PRE_{num}} \times 2^{EFF_{num}}$

1: $\mathcal{A}_s = \emptyset$
2: **for** $t \in \mathbb{T}$ **do**
3: $\quad \mathit{preconditions} \leftarrow \emptyset$ ▷ No add- or del effects, as they are unaffected
4: $\quad \mathit{numericPreconditions} \leftarrow \emptyset$
5: $\quad \mathit{numericEffects} \leftarrow \emptyset$

Variables do not have an order without a binding, so to translate $succ(v_2)$, we replace $succ(v_2)$ with a new variable $v2_{succ}$ and insert a precondition in a_{st_1} to enforce that $v2_{succ}$ is the successor of v_2.

$a_{st_1} = ($

$$
\begin{aligned}
&\mathit{preconditions} && = \{(\mathit{isSuccessor}\ v2_{succ}\ v_2)\},\\
&\mathit{add} && = \emptyset,\\
&\mathit{del} && = \emptyset,\\
&\mathit{numericPrecondition} && = \emptyset,\\
&\mathit{numericEffects} && = \emptyset
\end{aligned}
$$

$)$

The transition will then be considered as shown in Figure 34 with replaced variables for the rest of the translation.

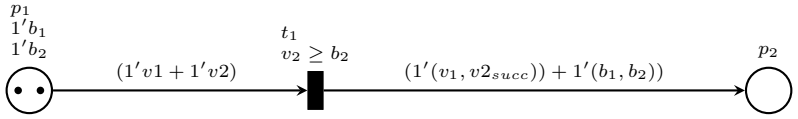


Fig. 34: CPN from Figure 34, but v_2 has been replaced with $v2_{succ}$

To do the general translation of an atom a in the form $pred(a)$ or $succ(a)$, then

1. construct a new variable v and add it to $\mathcal{P}_{\mathcal{X}}$,
2. Add a predicate to the precondition of a_s, that enforces the original relationship between a and v,
3. Replace every instance of a with v in the transition, both in arc- and guard expression and
4. Atoms can be nested to a finite depth, $pred(pred(...))$, so repeat from step 1 until all remaining atoms are either a color or variable.

6: **for** Atom $a \in \mathbb{A}$ **do**
7: **if** a is in the form $pred(a')$ **then**
8: Create some unique symbol a_{pred}
9: Add ($isPredecessor\ a_{pred}\ a'$) to $preconditions$
10: Substitute every $pred(a')$ to a_{pred}
11: **else if** a is in the form $succ(a')$ **then**
12: Create some unique symbol a_{succ}
13: Add ($isSuccessor\ a_{succ}\ a'$) to $preconditions$
14: Substitute every $pred(a')$ to a_{succ}
15: **end if**
16: **end for**

Let $translate : \mathbb{GE} \rightarrow 2^{\mathcal{P}_{\mathcal{X}}}$ translate a guard expression to a precondition expressions from Section 4.2.1.

As guard expressions compare the order of colors, each atom a is replaced with $(getIndex\ a)$, so $translate(\ v_2 \geq b_2\ ) = ((getIndex\ v2) \geq (getIndex\ b2))$ and

$$
\begin{aligned}
a_{st_1} = (\\
&preconditions && = \{(isSuccessor\ v2_{succ}\ v2)\},\\
&add && = \emptyset,\\
&del && = \emptyset,\\
&numericPrecondition && = ((getIndex\ v2) \geq (getIndex\ b2)),\\
&numericEffects && = \emptyset\\
)
\end{aligned}
$$

To translate any guard expression ge to a precondition expression, the first step is to transform ge to be in DNF (disjunctive normal form) without negation, which can be done by the following steps:

1. Transform ge to DNF - this is possible for any logical expression by applying De Morgan's laws.
2. Remove any negations, this is trivial aside from $\neg(a = b)$, which becomes $a < b \vee a > b$.
3. The flip of equality in step 2 potentially breaks DNF, so repeat from step 1 until valid DNF.

For a more extensive example, let $ge =\ (\neg(v1 = v2) \vee (v3 \geq v4)) \wedge \neg(v5 > v6)$, then the normalization is done over 6 steps:

Operation	**Effect of operation**
1. Normalize to DNF	$(\neg(v1 = v2) \wedge \neg(v5 > v6)) \vee ((v3 \geq v4) \wedge \neg(v5 > v6))$
2. Remove negations	$(((v1 < v2) \vee (v1 > v2)) \wedge (v5 \leq v6)) \vee ((v3 \geq v4) \wedge (v5 \leq v6))$
3. Valdiate DNF	Repeat
1. Normalize to DNF	$((v1 < v2) \wedge (v5 \leq v6)) \vee ((v1 > v2) \wedge (v5 \leq v6)) \vee ((v3 \geq v4) \wedge (v5 \leq v6))$
2. Remove Negations	No change
3. Validate DNF	Done

The precondition expression is then constructed by replacing every atom a in the normalized guard expression with the corresponding parameterized predicate $(getIndex\ a)$.

17: Add $translate(Guards(t))$ to $preconditions$

There are two conditions to fire a multiset transition, which are repeated here from Section 2.

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

The first condition has been translated by adding the guard to the precondition. For the example, the second condition is translated by adding a numeric preconditions for $Arcs(p1, t_1)$ and $Arcs(p1, t_1)$.

Additionally, the effects of applying t_1 is translated as numeric effects, and we have the translated a_{st_1}:

$$
\begin{aligned}
a_{st_1} = (\\
&preconditions && = \{(isSuccessor\ v2_{succ}\ v2)\}\\
&add && = \emptyset\\
&del && = \emptyset\\
&numericPrecondition && = ((getIndex\ v2) \geq (getIndex\ b2)) \wedge\\
& && \quad ((p1\ v1) \geq 1) \wedge ((p1\ v2) \geq 1)\\
&numericEffects && = \{\\
& && \quad ((p1\ v1) - 1),\\
& && \quad ((p1\ v2) - 1),\\
& && \quad ((p2\ v1\ v2_{succ}) + 1),\\
& && \quad ((p2\ b1\ b2) + 1)\\
& && \}\\
)
\end{aligned}
$$

Note that an issue occurs if the action schema is applied with the binding $b = \{v_1 \Rightarrow b_2, v_1 \Rightarrow b_2, v2_{succ} \Rightarrow b_1\}$. This issue is explained in Section 35 along with a workaround. This issue is prevalent for the rest of this section and the implementation.

Lastly, the general translation of arcs is as follows:

18: **for** $(p, (a_1, ..., a_n)) \in \mathbb{P} \times \mathbb{A}^+$ **do**
▷ Implied $typeOf(p) = typeOf((a_1, ..., a_n))$, as nothing in the loop has an effect otherwise

19: $multRequired \leftarrow Arcs(p,t)[]((a_1, ..., a_n))$
20: $multChange \leftarrow Arcs(t,p)[]((a_1, ..., a_n)) - Arcs(p,t)[]((a_1, ..., a_n))$

21: $pPlacePred \leftarrow getPred(p)$
22: $pred \leftarrow (pPlacePred\ a_1\ ...\ a_n)$

23: **if** $multRequired > 0$ **then**
24: Add $(pred >= multRequired)$ to $numericPreconditions$
25: **end if**

26: Add $(increase\ pred\ multChange)$ to $numericEffects$
27: **end for**

28: Add $(preconditions, \emptyset, \emptyset, numericPreconditions, numericEffects)$ to $\mathcal{A}_s$
29: **end for**

30: **return** $\mathcal{A}_s$

4.5 Translation Issue

There is an issue in the translation of arcs from a place to a transition, which is best illustrated with an example. Let $\mathbb{C}_{atom} = \{c\}$, then Figure 35 shows CPN $\mathcal{N} = (\{p\}, \{t\}, \{v\}, \{(p,t) \Rightarrow \boxed{c+v}\}, \{t \Rightarrow \boxed{true}\})$ with marking $M = \{p \Rightarrow 1'c\}$.

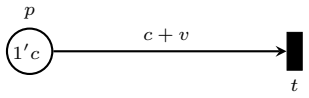


Fig. 35: CPN $\mathcal{N}$.

In CPN semantics, it is impossible to fire t from M with v bound to c, as this would consume $2'c$ from p. It is however possible in the translation, as the precondition constructed from $Arcs(p,t)$ becomes $((p\ c) \geq 1) \wedge ((p\ v) \geq 1)$. If the action is then applied with v bound to c, then precondition is reduced to $((p\ c) \geq 1)$ and it becomes possible to apply an illegal action and reach what is equivalent to a negative multiplicity in the marking.

A solution is proposed in Future Work, Section 5.1.1.

4.6 Correctness

Lemma 3. *If* $M \xrightarrow{t,b} M'$*, then* $\exists a_s \in \mathcal{A}_s, b \in \mathcal{B}_{Val}$*, s.t.* $translate(M) \xrightarrow{a_s,b'} translate(M')$*.*

Proof. To prove Lemma 3, we first show that a planning binding can be used as a CPN binding, implying $b \in \mathcal{B}_{Val}$, then we show that if t can be fired, then a_s can be applied. Then we show that applying a_s results in the same s' as firing t and then translating M'.

The planning binding b' can be constructed from the CPN binding b, as every mapping of b' is either from a CPN variable where $b'(v) = b(v)$, or a *pred* or *succ* placeholder that was constructed from the translation.

Let $a_{st} = translate(t)$, then by construction of the precondition of a_{st}, a_{st} can be applied with binding b' if t can be fired with binding b.

Therefore if $M \xrightarrow{a_{st},b} M'$, then $s \xrightarrow{a_s,b'} s'$

Let M be a marking and $translate(M) = (s_{classic}, s_{num})$. For every place $p \in \mathbb{P}$, $pred = getPred(p)$ and $(c_1, ..., c_n) \in \mathbb{C}$ then:

$M(p)((c_1, ..., c_n)) = s_{num}((pred\ c_1\ ...\ c_n))$ by construction.

Let x be the change in marking, $x = M'(p)((c_1, ..., c_n)) - M(p)((c_1, ..., c_n))$, then by the construction of numeric effects, ${s_{num}}'((pred\, c_1\ ...\ c_n)) = s_{num}((pred\, c_1\ ...\ c_n)) + x$.

$s_{num}((pred\ c_1\ ...\ c_n)) + x = translate(M')((pred\ c_1\ ...\ c_n))$,
or simply $s' = translate(M')$

Lemma 4. *Let* $s = translate(marking\ M)$*. If* $s \xrightarrow{a_{st},b} s'$ *and the precondition of* (a_{st}, b) *does not contain the same fact more than once, then* $\exists$*binding* b' *s.t.* $M \xrightarrow{t,b'} M'$ *and* $s' = translate(M')$

Proof. Following the same idea as the proof for Lemma 3, a planning binding is already a CPN binding, and by construction there is always a transition t, s.t. $translation(t) = a_s$, where they change the marking or state for a given place and color by the same amount as the associated or predicate and objects.

So if $s = translate(marking\ M)$, $s \xrightarrow{a_{st},b} s'$ and $M \xrightarrow{t,b'} M'$, then $s' = translate(M')$ is already established.

There is a semantic difference in the evaluation of the guard expression vs the precondition, so sometimes an action can be fired despite a transition cannot be fired. If however every proposition in the precondition of a_s with binding b is

unique, then the precondition is constructed s.t. if a_s can be applied, then t can be fired.

Theorem 2. *For every trace in a CPN $\mathcal{N} = (\mathbb{P}, \mathbb{T}, \mathbb{V}, Arcs, Guards)$, there is a plan in the lifted, numeric planning task $translate(\mathcal{N}) = \langle (\mathcal{T}, \leq), \mathcal{O}, \mathcal{X}, \mathcal{P}, \mathcal{P}_{\mathcal{X}}, \mathcal{A}_s \rangle$, and for every plan in $translate(\mathcal{N})$, there is a trace, if the plan contains no application of an action schema with a binding that causes the same fact to appear more than once in the precondition.*

Proof. sketch To prove Theorem 2, we use Lemma 3 and 4 show that for any trace there is a plan, and then we show that for any plan that satisfies the restriction, then there is a trace.

1. The empty plan is the empty trace and opposite.

2. Lemma 3 means that given any trace, $M \xrightarrow{(t,b)w} M'$, $M \xrightarrow{(t,b)} M''$ and $M'' \xrightarrow{w} M'$, where $w \in (\mathbb{T} \times \mathbb{B})^*$ and $(t, b) \in w$,
   Then there is a plan: $translate(M) \xrightarrow{(t,b)w} translate(M'), translate(M) \xrightarrow{(t,b)} translate(M'')$ and $translate(M'') \xrightarrow{w} translate(M')$, where $w \in (\mathcal{A}_s \times \mathcal{B}_{Val})^*$ and $(t, b) \in w$

3. Lemma 4 means that given a translated CPN with a marking M_0, then for any plan $translate(M_0) \xrightarrow{(a_s,b)_1} s_1 ... s_{n-1} \xrightarrow{(a_s,b)_n} s_n$, where the restriction is satisfied,
   Then there exists some trace $M_0 \xrightarrow{(t,b)_1} M_1 ... M_{n-1} \xrightarrow{(t,b)_n}$, where $s_1 = translate(M_1)$, ... $s_n = translate(M_n)$.

4.7 Implementation

The full implementation of the translation from Section 4.3 can be found on Github.com[7]. It is implemented within the Java frontend of TAPAAL, with 2260 additions and 43 files changed according to GitHub.

When the TAPAAL frontend is run with the parameters *-pddl-test model.pnml queries.xml*, the implementation will output whether the model can be translated and which queries are supported, as the queries.xml can contain many queries in a single file.

The implementation covers the entire CPN syntax defined in Section 2, as well as some more constructs in PNML, such as *all* colors and *scalars*. The implementation uses the PNML parser of TAPAAL and most subexpressions are handled, with the exceptions of subtraction in arc expressions, "PlaceHolderExpression" which seems to be some internal implementation class of TAPAAL and anything that is specifically for "timed" CPNs.

For a query to be supported, it must be a reachability query, which is in the following syntax:

$$value : p \in \mathbb{P} \mid i \in \mathbb{Z}$$

$$expr : expr_1 \wedge expr_2 \mid expr_1 \vee expr_2 \mid value_1 \bowtie value_2$$

$$query : \text{EF } expr$$

Where $\bowtie \in <, \leq, =, \neq, \geq, >$ and an "EF *expr*" query asks if there exists a trace, s.t. the *expr* is satisfied. The query EF $6 < p$ asks whether it is possible to reach a marking M s.t.

$$6 < \sum_{c \in \mathbb{C}} M(p)(c)$$

When the implementation is called with the parameters *-pddl-out outDir model.pnml query.xml*, then the translated domain as well as a separate task is output as separate PDDL files in *outDir*.

Let *model.pnml* contain the CPN shown in Figure 36, and let *query.xml* be the query $6 < p_2$.

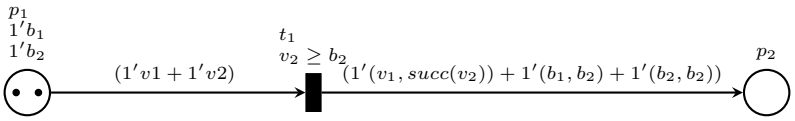


Fig. 36: The CPN in model.pnml where $typeOf(p_1) = C$ and $typeOf(p_2) = (C, C)$, and $typeValues(C) = \{b_1, b_2\}$.

[7] https://github.com/petrinet-planning/tapaal-gui/tree/f73554efad7456883d780ae2b4ba97dd3dd38799

Listing 2.1 and 2.2 show the content of *outDir/model.pddl* and *outDir/query.pddl* respectively. The content is the exact output, aside from a few automatic linebreaks where the linenumbers skip. Note that *dot* only exists here as an unused color added by Tapaal.

Listing 2.1: outDir/model.pddl

```
1  (define (domain ComposedModel)
2          (:requirements :typing :fluents)
3          (:types
4                  C
5                  dot
6          )
7          (:constants
8                  b1 b2 - C
9                  dot_obj - dot
10         )
11         (:predicates
12                 (isPredecessor ?x ?xPred)
13                 (isSuccessor ?x ?xSucc)
14         )
15         (:functions
16                 (P1 ?x - C)
17                 (P2 ?x1 - C ?x2 - C)
18         )
19         (:action T1
20                 :parameters ( ?v2_CpnColorSuccessor - C ?
                       v1 - C ?v2 - C)
21                 :precondition (and
22                         (isSuccessor ?v2 ?
                               v2_CpnColorSuccessor)
23                         (>= (toInt ?v2) (toInt b2))
24                         (>= (P1 ?v1) 1)
25                         (>= (P1 ?v2) 1))
26                 :effect (and
27                         (increase (P2 b2 b2) 1)
28                         (increase (P2 b1 b2) 1)
29                         (increase (P2 ?v1 ?
                               v2_CpnColorSuccessor) 1)
30                         (decrease (P1 ?v1) 1)
31                         (decrease (P1 ?v2) 1))
32         )
33 )
```

Listing 2.2: outDir/query.pddl

```
(define (problem ComposedModel_problem)
        (:domain ComposedModel)
        (:objects
        )
        (:init
                (= (toInt dot_obj) 0)
                (= (toInt b1) 0)
                (= (toInt b2) 1)
                (isPredecessor dot_obj dot_obj)
                (isSuccessor dot_obj dot_obj)
                (isPredecessor b1 b2)
                (isSuccessor b2 b1)
                (isPredecessor b2 b1)
                (isSuccessor b1 b2)
                (= (P2 b1 b1) 0)
                (= (P2 b2 b2) 0)
                (= (P2 b2 b1) 0)
                (= (P2 b1 b2) 0)
                (= (P1 b1) 1)
                (= (P1 b2) 1)
        )
        (:goal
                (and
                        (< 6 (+ (P2 b2 b2) (+ (P2 b2 b1)
                            (+ (P2 b1 b2) (P2 b1 b1))))))
        )
)
```

4.8 Experiments Setup

To test the implementation, the colored nets of the Model Checking Competition[16] 2023 has been used, and the exact models can be found on their website[8].

Experiments were run on Aalborg Univeristy's "naples" cluster computer with an AMD EPYC 7551 CPU, each experiment with a timeout of 4 hours and 16 GiB RAM. The Python script generate_experiment_scripts_c2p.py[9] generates the scripts run on the cluster.

VerifyPN, the verification tool of TAPAAL with the exact version found GitHub.com[10], was used to test the satisfiability of supported queries on CPNs. Two special parameters were used with VerifyPN. It is set to use the search strategy "Random

[8] https://mcc.lip6.fr/2023/models.php
[9] https://github.com/petrinet-planning/benchmarking/blob/b9d61673ce2cb579f5ed22897547936837739 1dc/generate_e
[10] https://github.com/TAPAAL/verifypn/tree/53b047edca7a880135c29fd1659fc6ddccebe057

Potency First Search"[12], and to validate only the one exact query that was tested at each run.

ENHSP or Expressive Numeric Heuristic Search Planner[22] with the exact version found on GitLab.com[11], is a planning system that supports numeric tasks. It has been used to run the translated CPNs as numeric lifted planning tasks.

Experiments have been timed using the unix *time* command, where "search time" is the seconds taken for VerifyPN or ENHSP, and "total time" is either the time of VerifyPN or our the time of our translation + ENHSP. The time is the median of running VerifyPN or ENHSP 3 times, or our translation once.

4.9 Experiments

In the MCC2023-Color benchmarks used, there are 465 queries over 207 CPNs, that are of the supported format. Of these, we set the threshold for including them in the results further down, that the translation succeeded, and the verifier and planner must understand the problem. 323 queries pass this test, and 142 queries were skipped for the following reasons:

- 71 queries were skipped because ENHSP reported syntax errors when a color name was an invalid object name, such as when a color name was just an integer,
- 71 were skipped because the translation failed, of which
  - 65 times the translator had a stack overflow due to expressions being nested too deeply, and
  - 6 times the translator ran out of memory.

20 queries caused issues, but are included in the results because they passed the threshold. These are included in the results and considered the same as the system running out of time or memory.

- 15 stack overflows by ENHSP during the search,
- 5 null pointer exceptions from ENHSP during the search

To test the implementation, we translate the 323 queries and pass them through TAPAAL and ENSHP, where they are categorized into *satisfiable*, *unsolvable* or *undecided*.

To estimate whether a plan was affected by the issue from Section 4.5, a plan gets *flagged* if any application in the plan contains two parameters assigned to the same object.

There is 1 query where TAPAAL and ENHSP disagree. TAPAAL returns unsolvable, but ENHSP states satisfiable. This is however a flagged instance with

[11] https://gitlab.com/enricos83/ENHSP-Public/-/tree/92053ac811ea8c76b8fe9d994e7f8ebc78a2427c

the same object assigned to two variables in the plan, and is therefore not a counterexample to the theorem nor correctness of the implementation. It would however be beneficial to implement the workaround in Section 5.1.1.

Aside from the 1 instance where they disagree, the results are shown in Table 3

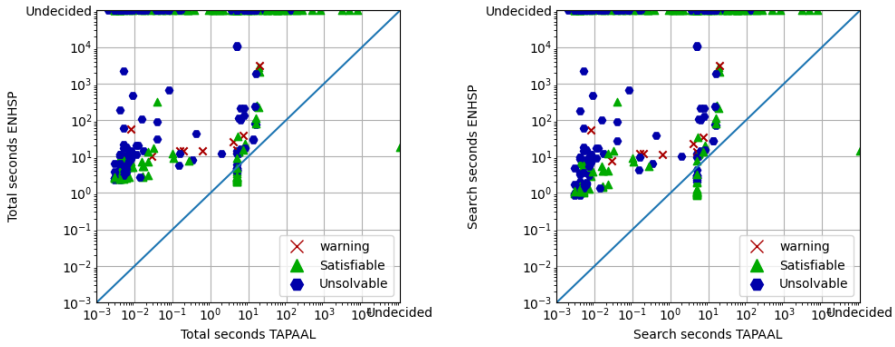
	TAPAAL	ENHSP
Satisfiable Queries Answered	**92**	**70**(+**9** flagged)
Satisfiable Uniquely Answered	15	2
Satisfiable Total Faster	51	18
Satisfiable Search Faster	51	18
Unsolvable Queries Answered	**219**	**86**
Unsolvable Queries Uniquely Answered	133	0
Unsolvable Total Faster	76	10
Unsolvable Search Faster	75	11
Undecided	**12**	**158**
Undecided (Timed out)	0	5
Undecided (Out of Memory)	12	133
Undecided (Stack Overflow)	0	15
Undecided (Null Pointer)	0	5

Table 3: The numeric comparisons after calling TAPAAL and ENHSP on 323 queries. The bold numbers in each column sum to 323, and the rest are further categorization.

TAPAAL generally outperforms ENHSP, and especially for unsolvable queries. It is somewhat expected for TAPAAL to outperform ENSHP in unsolvable queries, as it is a verification tool vs a planning tool. In verification, there is a stronger focus on detecting satisfiability vs unsolvability than there is in planning. Planning is typically more focused on finding a plan for satisfiable problems. An example of this is that IPC2023 (International Planning Competition 2023) for numeric tracks, which ENHSP would more typically compete in, has no unsolvable benchmarks[6].

Another observation is that ENHSP generally ran out of memory rather than timing out. Part of it may be that there is a long timeout of 4 hours, but it would seem that ENHSP may be using its memory inefficiently for unsolvable queries, compared to TAPAAL. Alternatively the lifted planning domain could be more complex to represent than a CPN, but it seems unlikely that TAPAAL does something that is not possible to do for a numeric planner, as it seems possible to translate a lifted numeric planning problem back to CPN, as reasoned about in Section 5.1.2

The time spent by each system is shown in Figure 37.



(a) Translation + Search time (TAPAAL translation time is always exactly 0, as it goes from CPN to CPN).

(b) The time taken when calling just ENHSP or TAPAAL.

Fig. 37: Logscale for seconds taken for ENHSP vs TAPAAL. Warning means the plan of ENHSP had the same object assigned to different parameters twice in a single action application. The further a point is to the right, the more time TAPAAL spent to answer the query. Likewise the further up a point is, the longer time ENHSP spent. If a point is above the line, it means ENHSP was slower than TAPAAL. 11 points were omitted, as they were undecided by both systems, and would just be a single point in the top right corner.

Some tendencies appear in the data:

1. TAPAAL seems to have some results spread out over most of the timespan, though most of the results are shown in the top where ENHSP didn't answer.
2. There appear to be some clusters, treating the graph as a coordinate system to refer to positions, there is a cluster at $(10^{-2}, 10^{-1})$ and one at $(10^{1}, 10^{1})$. By manual inspection, a lot of the queries are trivially solved from the initial marking, so this is likely to cause the first cluster. Some of the queries also seem to be very unsolvable, meaning that one of them requires multiplicity of 86 in a place that at most ever have a multiplicity of 1. TAPAAL can analyse and answer some of these queries without running the search, the second cluster could be caused by ENHSP doing something similar.
3. It takes much longer for the fastest answer of ENHSP than the faster answer of TAPAAL. Part of it may be preprocessing, but there may also be some overhead in ENHSP's Java implementation and loading the JVM, compared to the C++ implementation of TAPAAL's verification engine.

5 Conclusion

The improved translation from classical lifted planning to a set-based CPN is a step in the right direction, however, it is still slower in almost every domain when compared to both the Grounded implementation and especially Fast Downward. However, the ability to translate planning tasks may also be useful for other things than solving benchmarks the fastest, as both the old and new translations have been used to generate novel colored Petri nets for the MCC2024. Additionally, a novel translation from multiset-based CPNs to lifted numerical planning has been defined, implemented, and had its correctness argued. One issue has been identified but a proposed solution is offered in Future Work. The implementation of the translation has been tested with the MCC2023 benchmarks. The results show that the combination of our translation and the ENHSP planning system is generally slower than TAPAAL for solving queries on CPNs, and does poorly on unsolvable queries, but when they both answer a query, they always agree on unsolvability. There is just 1 case where the identified issue caused a wrong answer, but otherwise, it is promising that they agree on the reachability of markings.

Acknowledgements We would like to extend special thanks to Álvaro Torralba and Jiří Srba for providing expertise and guidance throughout our master's thesis. Particularly Álvaro with focus on planning, and Jiří for Petri nets. We would also like to thank the Institute of Computer Science at AAU for bearing with us while we occupied a large chunk of the computer cluster for a week.

5.1 Future Work

5.1.1 Workaround Semantic Issue A workaround for the issue in Section 4.5 is to translate transition t into two action schemas $a_{sdecrement}$ and $a_{sincrement}$, which must be applied in sequence.

Let the precondition of $a_{sdecrement}$ be $translate(Guards(t))$, and the effect be every multiplicity decrement from arcs going towards t.

The precondition of $a_{sincrement}$ is that no place has a negative multiplicity and the effect is the union of increments from arcs going out of t.

With this construction, any illegal plan will deadlock after applying $a_{sdecrement}$.

The sequence can be enforced by having some *lock* predicate that is initially *true* and a predicate $(isNext\ t_i)$ per transition $t_i \in \mathbb{T}$. $a_{sdecrement}$ then requires *lock* and makes $(isNext\ t)$ true for its own transition. $a_{sincrement}$ then requires $(isNext\ t)$, makes it $false$ again and sets *lock* to *true*.

5.1.2 Translate Lifted Numeric Planning to Colored Petri Net It seems reasonable that a lifted numeric planning task could be translated to a multiset-based CPN. Taking inspiration from our previous translations:

1. Objects can be represented as colors. Types work the same in classical and numerical planning, so the type structure and algorithm for ordering colors from our set-based translation may be useful here,
2. Numeric predicates can be represented as places, and
3. Action schemas can be represented as transitions. Action schema preconditions can be translated to guard expressions.

The biggest challenge here is that numeric planning is a superset to classical, so the constructed CPN will have to work with both multiset and set-based semantics.

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. Bønneland, F., Dyhr, J., Jensen, P., Johannsen, M., Srba, J.: Stubborn versus structural reductions for petri nets. Journal of Logic and Algebraic Programming **102**, 46–63 (2019). https://doi.org/10.1016/j.jlamp.2018.09.002
3. Jesper A. van Diepen, N.J., Sørensen, M.M.: Improvements to structural reductions of petri nets with inhibitor arcs. Aalborg University (2022)
4. Jesper A. van Diepen, N.J., Sørensen, M.M.: Structural Reductions of Colored Petri Nets and P/T Nets with Inhibitor Arcs. Master's thesis, Aalborg University (2022)
5. 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
6. Fišer, D., Pommerening, F.: International planning competition 2023 classical tracks. Presentation of results (2023), https://ipc2023-classical.github.io/results/presentation.pdf
7. Ginnerup, H., Lassen, S.: Planning via colored petri nets. Aalborg University (2024)
8. Gnad, D.: Star-Topology Decoupled State-Space Search in AI Planning and Model Checking. Ph.D. thesis, Saarland University (2021)
9. Helmert, M.: Decidability and undecidability results for planning with numerical state variables. In: Ghallab, M., Hertzberg, J., Traverso, P. (eds.) Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems, April 23-27, 2002, Toulouse, France. pp. 44–53. AAAI (2002), http://www.aaai.org/Library/AIPS/2002/aips02-005.php
10. Helmert, M.: The fast downward planning system. J. Artif. Intell. Res. **26**, 191–246 (2006), https://api.semanticscholar.org/CorpusID:17305
11. 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
12. 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)
13. 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
14. Illanes, L., McIlraith, S.A.: Numeric planning via abstraction and policy guided search. In: Sierra, C. (ed.) Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017. pp. 4338–4345. ijcai.org (2017). https://doi.org/10.24963/ijcai.2017/606, https://doi.org/10.24963/ijcai.2017/606
15. 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

16. Kordon, F., Bouvier, P., Garavel, H., Hulin-Hubard, F., Amat., N., Amparore, E., Berthomieu, B., Donatelli, D., Dal Zilio, S., Jensen, P., Jezequel, L., Paviot-Adet, E., Srba, J., Thierry-Mieg, Y.: Complete Results for the 2023 Edition of the Model Checking Contest. https://mcc.lip6.fr/2023/results.php (April 2023)
17. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4), 541–580 (1989). https://doi.org/10.1109/5.24143
18. Petri, C.A.: Communication with automata (1966), `https://api.semanticscholar.org/CorpusID:8420535`
19. Poole, D.L., Mackworth, A.K.: Artificial intelligence, pp. 239–240. Cambridge University Press, Cambridge, England, 2 edn. (Sep 2017)
20. 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)
21. 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`
22. Scala, E., Haslum, P., Thiébaux, S., Ramírez, M.: Interval-based relaxation for general numeric planning. In: ECAI. Frontiers in Artificial Intelligence and Applications, vol. 285, pp. 655–663. IOS Press (2016)
23. 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)