Planning with Effectively Propositional Logic

Juan Antonio Navarro-Pérez and Andrei Voronkov

The University of Manchester

Abstract. We present a fragment of predicate logic which allows the use of equality and quantification but whose models are limited to finite Herbrand interpretations. Formulae in this logic can be thought as syntactic sugar on top of the Bernays-Schönfinkel fragment and can, therefore, still be effectively grounded into a propositional representation. We motivate the study of this logic by showing that practical problems from the area of planning can be naturally and succinctly represented using the added syntactic features. Moreover, from a theoretical point of view, we show that this logic allows, when compared to the propositional approach, not only more compact encodings but also exponentially shorter refutation proofs.

1 Introduction

Planning has been the focus of attention of many researchers in the field of artificial intelligence, where it was originally conceived as a formalisation of deduction processes [3]. Alternatively, the problem of finding a sequence of actions to reach, from an initial state, a set of desired goals, has also been reduced to the problem of finding a satisfying truth assignment for a propositional logic formula [5, 6].

In this paper we follow a similar approach but, instead of a propositional encoding, we use a fragment of predicate logic which allows some limited use of equality and quantification. This fragment, which we call *finite domain predicate logic*, allows a much more succinct and natural representation of problems. The size of the resulting formula is linear in the size of, for example, a STRIPS description [2] of the original planning problem.

We show, moreover, that any formula in the proposed logic can be translated to an equisatisfiable formula in the Bernays-Schönfinkel fragment of predicate logic. Formulae in this fragment have an $\exists^*\forall^*$ prefix when written in prenex normal form and do not allow the use of function symbols. This makes their Herbrand universe finite and, therefore, to test satisfiability one can effectively replace a formula by all its propositional ground instances. This is why formulae in this class are also referred to as *effectively propositional* (EPR), such as in one of the categories of the CASC system competitions [9].

Another motivation for using this logic as a formalism to represent problems is the fact that, as we show in this paper, not only descriptions can be much more concise, but inference steps can also be exponentially more efficient than in the propositional case. On the other hand, our encoding may also turn out to be useful for propositional, SAT-based, approaches to planning. Indeed, it preserves the structure of the original planning problem in the obtained effectively propositional formula and reduces the problem of finding an optimised propositional encoding to the problem of finding an optimised propositional instantiation of the EPR description. Thinking in this more general fragment of first order logic, often allows one to find simplifications or alternative encodings that one might miss if only focused in the propositional case.

Reasoning with effectively propositional theories is an relatively new area of research, which seems to offer a language with a good compromise between expressibility and complexity. There are many computer scientists currently developing ideas and procedures in order to more efficiently deal with this kind of formulae. Unfortunately, there is also a lack of benchmarks for these researchers to experiment and test their systems. An important contribution of this paper is to aid filling in this gap by providing a new and rich source of problems with close links to real-life applications.

Our paper is structured as follows: In Section 2 we introduce the syntax and semantics of the finite domain predicate logic and show that it can be reduced to the Bernays-Schönfinkel fragment of predicate logic. We also present an example of a family of effectively propositional unsatisfiable formulae, whose refutation proofs are exponentially shorter that those possible in the propositional setting. Then in Section 3 we formally introduce the notions of planning to later give, in Section 4, the encoding of planning problems in terms of our finite domain predicate logic.

2 Finite domain predicate logic

In this section we introduce the finite domain predicate logic. It allows the use of equality, evaluated under the unique name assumption, and quantification over finite domains. We will also later show that formulae in this logic can be reduced to the Bernays-Schönfinkel fragment of predicate logic. The added syntactic sugar will be useful in later sections to describe our encodings of planning problems more naturally.

Definition 1. The language of finite domain predicate logic consists of a set of *predicate symbols* $\mathcal{P}$, a finite set of *constant symbols* $\mathcal{D}$, and a set of *variables* $\mathcal{V}$. Predicate symbols are, moreover, associated with a positive integer which we call its *arity*. The set $\mathcal{D}$ is also referred to as the *domain* of the logic. A *term* is either a variable or a constant symbol. A *predicate atom* is an expression of the form $p(t_1, \ldots, t_n)$ where $p \in \mathcal{P}$ is a predicate symbol of arity n and each t_i is a term. An *equality atom* is an expression of the form $t = t'$ where both t and t' are terms. An *atom* is either a predicate or an equality atom. A *ground atom* is an atom all whose terms are constant symbols.

We consider the following as primitive connectives of the logic: *falsity* ($\bot$), *negation* ($\neg\phi$), *conjunction* ($\phi \wedge \psi$) and *quantification* ($\forall X \in C.\, \phi$); where ϕ and ψ are formulae, X a variable and $C \subseteq \mathcal{D}$ a set of constant symbols. Duals of these operators and additional connectives can be introduced as abbreviations:

$$\top \equiv \neg\bot \qquad \exists X \in C.\,\phi \equiv \neg(\forall X \in C.\,\neg\phi)$$
$$\phi \vee \psi \equiv \neg(\neg\phi \wedge \neg\psi) \qquad \phi \rightarrow \psi \equiv \neg\phi \vee \psi$$

The standard notion of *free* and *bound* variables with respect to the scope of quantifiers also applies here. A *closed formula* is a formula with no free variables. We will often use $\overline{X}$ to denote a sequence of variables $X_1, \ldots, X_n$ whose length is specified in the context where it is used. This allows, for example, to write $\exists \bar{X} \in C^n.\,\phi$ instead of the longer expression $\exists X_1 \in C \ldots \exists X_n \in C.\,\phi$. Similarly we will write $\overline{X} = \overline{Y}$ as a shorthand for $\bigwedge_{i=1}^{n} X_i = Y_i$.

A *substitution* is a function σ that maps variables to terms, and behaves like the identity function almost everywhere. We denote by $\phi\sigma$ the result of applying a substitution σ to a formula ϕ, i.e. the formula obtained after uniformly replacing every free variable X in ϕ with the term $X\sigma$. We also say that ϕ' is an *instance of* ϕ if there is a substitution σ such that $\phi' = \phi\sigma$; and that σ is an *unifier* of the formulae ϕ and ϕ' if $\phi\sigma = \phi'\sigma$. A substitution is often denoted by explicitly enumerating its mappings, e.g.: $\{X_1 \rightarrow t_1, \ldots, X_n \rightarrow t_n\}$.

A *Herbrand interpretation* is a set of ground predicate atoms. The notion of whether a Herbrand interpretation $\mathcal{I}$ is a *model* of a closed formula ϕ, denoted by $\mathcal{I} \models \phi$, is defined as follows:

$$\begin{array}{lll}
\mathcal{I} \not\models \bot & & \\
\mathcal{I} \models p(\bar{c}) & \text{iff} & p(\bar{c}) \in \mathcal{I}, \\
\mathcal{I} \models c = c' & \text{iff} & c \text{ coincides with } c', \\
\mathcal{I} \models \neg\phi & \text{iff} & \mathcal{I} \not\models \phi, \\
\mathcal{I} \models \phi \wedge \psi & \text{iff} & \mathcal{I} \models \phi \text{ and } \mathcal{I} \models \psi, \\
\mathcal{I} \models \forall X \in C.\,\phi & \text{iff} & \mathcal{I} \models \phi\,\{X \rightarrow c\} \text{ for every constant } c \in C,
\end{array}$$

where $\bar{c}$ denotes a tuple of constant symbols of the length equal to the arity of p. When we we speak about models of non-closed formulae, we assume the free variables of these formulae to be implicitly universally quantified. A formula is said to be *satisfiable* if it has at least one model. ■

In the sequel we assume to deal with a logic having a fixed domain $\mathcal{D}$. For example, all formulae may only use constants from $\mathcal{D}$. Observe that equality is evaluated syntactically with respect to the constant symbol names, i.e. using the unique name assumption, and does not depend on the interpretation $\mathcal{I}$. We will now show how this restricted kind of equality can be removed from formulae while preserving its satisfiability status.

We will also call formulae *constraints* and assume that a finite set of constraints represents the conjunction of all its elements. A *clause* is a simple disjunction of literals, that is, atoms or their negations. A formula is said to be in *clause normal form* if it is a conjunction of clauses. Although we do not always write sets of constraints in clause normal form, they can often be easily rewritten in such form using simple logical identities (e.g. changing implication for disjunction).

Definition 2. Let $C = \{c_1, \dots c_k\}$ be a set of constant symbols. We introduce the fresh new predicate symbols succ_C, less, in_C, and eq and define the set of constraints Ord_C using these symbols as follows:

$$\begin{aligned}
&\mathsf{succ}_C(c_1, c_2) \wedge \cdots \wedge \mathsf{succ}_C(c_{k-1}, c_k)\\
&\mathsf{succ}_C(X, Y) \rightarrow \mathsf{in}_C(X) \wedge \mathsf{in}_C(Y)
\end{aligned}$$

also the set of constraints EQ is defined as:

$$\begin{aligned}
&\mathsf{succ}_{\mathcal{D}}(X, Y) \rightarrow \mathsf{less}(X, Y)\\
&\mathsf{less}(X, Y) \wedge \mathsf{less}(Y, Z) \rightarrow \mathsf{less}(X, Z)\\
&\mathsf{less}(X, Y) \rightarrow \neg\mathsf{eq}(X, Y) \wedge \neg\mathsf{eq}(Y, X)\\
&\mathsf{eq}(X, X)
\end{aligned}$$

The intuition behind the set Ord_C is to enumerate all constant symbols in the set C by providing a predicate succ_C that allows one to iterate over them. The intended meaning of $\mathsf{in}_C(X)$ is to represent that $X \in C$. Note that Ord_C defines only the positive polarity of the predicate symbols succ_C, less and in_C. In other words, if $c \in C$ then the predicate $\mathsf{in}_C(c)$ should be true in models of Ord_C, but the converse does not necessarily have to hold.

When we use this construction to enumerate all the elements in the domain, i.e. $\mathsf{Ord}_{\mathcal{D}}$, then the set EQ defines an additional predicate $\mathsf{eq}(X, Y)$ that can be used to replace the built-in equality $X = Y$.

Theorem 1. *Let ϕ be a closed formula, and let ϕ^{eq} be the formula obtained by replacing all equality atoms $t = t'$ in ϕ with $\mathsf{eq}(t, t')$. The pair of formulae ϕ and $\mathsf{Ord}_{\mathcal{D}} \wedge \mathsf{EQ} \wedge \phi^{\mathsf{eq}}$ are equisatisfiable.*

Proof. The result follows from the fact that, if an interpretation $\mathcal{I}$ is a model of the formula $\mathsf{Ord}_{\mathcal{D}} \wedge \mathsf{EQ}$ then $i < j$ implies $\mathcal{I} \models \mathsf{less}(c_i, c_j)$ and, using the rest of the constraints in EQ, we have $\mathcal{I} \models \mathsf{eq}(c_i, c_j)$ iff $i = j$ iff $c_i = c_j$ (from the unique name assumption). $\square$

Also note that, since quantification is done with respect to a fixed finite set of constant symbols, a quantified subformula $\forall X \in C.\, \phi(X)$ can actually be unfolded into the conjunction $\bigwedge_{c \in C} \phi\, \{X \rightarrow c\}$. Although naively unfolding quantifiers this way could produce an exponential blow-up in the size of the formula, we now show an alternative approach that only incurs in a linear increase. It follows the style of structural clause form translations as proposed by Plaisted and Greenbaum [7].

In order to simplify the exposition, and avoid dealing with the polarity of subformulae, we assume that formulae are first put in *negation normal form.* This can be easily achieved by pushing negation inwards and replacing implications with disjunctions. The resulting logic formulae can therefore use any of the connectives: $\bot, \top, \wedge, \vee, \forall, \exists$; and negation is restricted to the front of atoms only.

Definition 3. Let Γ be a set of constraints. The set Γ^{ea} is defined as the result of iterating the following procedure to remove all explicit quantifiers in Γ

– If there is a subformula $\psi = \forall X \in C.\, \phi(X, \overline{Y})$, where $X, \overline{Y}$ are all the free variables in ϕ and $C = \{c_1, \ldots, c_d\}$, then replace the subformula ψ with the atom $\mathsf{forall}_\psi(\overline{Y})$ and add the constraint:

$$\mathsf{in}_C(X) \land \mathsf{forall}_\psi(\overline{Y}) \to \phi(X, \overline{Y})$$

– Similarly, replace a subformula of the form $\psi = \exists X \in C.\, \phi(X, \overline{Y})$ with the atom $\mathsf{exists}_\psi(\overline{Y})$ and add the constraints:

$$\begin{array}{l}\mathsf{exists}_\psi(\overline{Y}) \to \mathsf{find}_\psi(c_0, \overline{Y}) \\ \mathsf{find}_\psi(X, \overline{Y}) \to \phi(X, \overline{Y}) \lor \mathsf{xfind}_\psi(X, \overline{Y}) \\ \mathsf{succ}_C(X, Z) \land \mathsf{xfind}_\psi(X, \overline{Y}) \to \mathsf{find}_\psi(Z, \overline{Y}) \\ \mathsf{xfind}_\psi(c_d, \overline{Y}) \to \bot\end{array}$$

The set of constraints Ord_C is also appended to Γ^{ea} for each set C originally appearing in a quantifier. ■

Theorem 2. *The sets of constraints Γ and Γ^{ea} are equisatisfiable.*

Proof. The argument is similar to the one used in structural clause form translations where subformulae are replaced by fresh new atoms and constraints are added to give a meaning to those atoms.

In particular it can be shown that if $\mathcal{I} \models \psi^{\mathsf{ea}} \land \mathsf{forall}_\psi(\overline{Y})$ for a given interpretation $\mathcal{I}$, then $\mathcal{I} \models \forall X \in C.\, \psi$ and, for the converse, that if $\mathcal{I} \models \forall X \in C.\, \psi$ then one can always find an interpretation $\mathcal{I}'$ such that $\mathcal{I}' \models \psi^{\mathsf{ea}} \land \mathsf{forall}_\psi(\overline{Y})$. And analogously for the existential quantifier. □

Using the previous two results, it is then possible to eliminate equality and quantifiers from finite domain predicate formulae thus leaving formulae in the Bernays-Schönfinkel fragment. Moreover, the resulting formula is of linear size with respect to the original input. Additionally, these results complement the translation of Plaisted and Greenbaum [7] allowing one to write arbitrary finite domain predicate formulae in clause normal form. In the following we will then freely use equality and finite quantification knowing that they do not add any complexity to the logic.

2.1 Compact proofs

In order to further motivate the use of Bernays-Schönfinkel formulae as a formal language to represent problems, we prove in this section that reasoning within this fragment can also be exponentially more efficient than in propositional logic. This shows that the language not only provides means for creating more compact encodings, but that the actual solving time could also be reduced by the use of this approach.

We consider, in particular, proofs using the resolution inference system which operates on sets of clauses. It consists of two inference rules: resolution and factoring. We refer to the work of Bachmair and Ganzinger [1] for the definitions.

Given a set of clauses Γ, a *proof of* ϕ from Γ is a sequence of clauses $\phi_1, \ldots, \phi_l = \phi$ such that each ϕ_i is either an instance of a clause in Γ or the result of applying resolution to two previous clauses. If one can obtain a proof of the empty clause from Γ, then the set Γ is unsatisfiable and the proof is known as a *refutation* of Γ. A *propositional proof* is a proof where all the clauses in the sequence are ground.

The following example shows that there is a family of sets of clauses Γ_m with respective unsatisfiability proofs Φ_m, where the shortest propositional refutation of Γ_m is exponentially larger than Φ_m.

Theorem 3. *There is a sequence of sets of clauses $S_1, S_2, \ldots$ of increasing sizes such that each S_i has a refutation of a size quadratic in i and the shortest propositional refutation of S_i has a size exponential in i.*

Proof. In the set S_i we will use two constants: 0 and 1 and a single predicate symbol s of arity i. We will denote by $\bar{0}$, $\bar{1}$, $\bar{x}$ etc. sequences of constants 0, 1 and variables, respectively, whose length will be clear from the context. The set S_i consists of the following clauses:

$$s(\bar{0}). \tag{1}$$

i clauses of the form:

$$\neg s(\bar{x}, 0, \bar{1}) \vee s(\bar{x}, 1, \bar{0}). \tag{2}$$

The clause

$$\neg s(\bar{1}). \tag{3}$$

This set of clauses is unsatisfiable and its size is quadratic in i.

Note that every ground atom is of the form $s(\bar{b})$, where $\bar{b}$ is a sequence of bits representing a number between 0 and $2^i - 1$ written in binary notation. For a number n such that $0 \leq n < 2^i$ let us denote by $\underline{n}$ the sequence of bits denoting this number. Then (1) asserts $s(\underline{0})$ and (3) asserts $\neg s(\underline{2^i - 1})$, while the ground instances of clauses in (2) assert $\neg s(\underline{n}) \vee s(\underline{n+1})$. Using this observation it is not hard to argue that every unsatisfiable set of ground instances of clauses in S_i contains all ground instances of (2), and so all propositional refutations of this set have a size exponential in i.

Let us show that S_i has a non-ground refutation of a quadratic size. To this end, we will show, by induction on the length of a non empty sequence of constants $\bar{1}$, resolution proofs of the clauses

$$\neg s(\bar{x}, \bar{0}) \vee s(\bar{x}, \bar{1}), \tag{4}$$

having a number of steps linear in the length of $\bar{1}$.

When the length is 1, then (4) is an instance of (2). When the length is greater than 1, we can assume, by induction, that there is such a refutation of a clause

$$\neg s(\bar{x}, y, \bar{0}) \vee s(\bar{x}, y, \bar{1}). \tag{5}$$

From this and (2) we can derive by a resolution inference the clause

$$\neg s(\bar{x}, 0, \bar{0}) \vee s(\bar{x}, 1, \bar{0}).$$

From this and (5) we can derive by a resolution inference the clause

$$\neg s(\bar{x}, 0, \bar{0}) \vee s(\bar{x}, 1, \bar{1}).$$

and we are done.

This implies that there is a resolution proof of the clause

$$\neg s(\bar{0}) \vee s(\bar{1})$$

having a number of steps linear in i, and hence a refutation having a number of steps linear in i. Moreover, the size of each clause in the refutation is linear in i, so the size of the refutation is quadratic in i. $\square$

3 Planning

In this section we formally introduce several notions and concepts related to planning. We first introduce the notion of a planning domain where applicable actions, their preconditions and consequences, are described. We then proceed to define what a planning problem and its solutions are. Our formalism corresponds to STRIPS style planning as introduced by Fikes and Nilsson [2].

Definition 4. The language of a planning domain consists of a triple of finite sets of symbols $(\mathcal{O}, \mathcal{F}, \mathcal{A})$ which are respectively called *object*, *fluent* and *action symbols*. Fluent and action symbols have, moreover, an associated natural number which we call the *arity* of the symbol. If f is a fluent symbol of arity m, then an expression of the form $f(t_1, \ldots, t_m)$, where each t_i is either a variable or an object symbol, is called a *fluent*.

An *action* is a triple $(\alpha_{\text{req}}, \alpha_{\text{add}}, \alpha_{\text{del}})$ where $\alpha = a(X_1, \ldots, X_n)$, for an action symbol $a \in \mathcal{A}$ of arity n, is the *signature* of the action. Each element in the triple is a set of fluents of the form $f(t_1, \ldots, t_m)$ where each t_i is either an object symbol or a variable X_j with $1 \leq j \leq n$. We say that these are the fluents that, respectively, the action *requires*, *adds* and *deletes* when it is executed. A *planning domain* $\mathcal{D}om$ is simply a set of actions and its size, denoted $|\mathcal{D}om|$, is defined as the number of symbols occurring in the description of all its actions.

An *action instance* $\alpha' = \alpha\sigma$, where σ is any substitution, corresponds to the triple of sets of fluent instances $({\alpha'}_{\text{req}}, {\alpha'}_{\text{add}}, {\alpha'}_{\text{del}})$, where ${\alpha'}_{\text{req}} = \alpha_{\text{req}}\sigma$, etc. $\blacksquare$

Example 1. We will consider as a running example for this section, a planning domain in the context of logistics. This domain has, among others, an action load-truck that takes three parameters: a package X_1 to load, a truck X_2 where to load the package, and a location X_3 where the loading takes place. The definition of such an action would probably look like:

load-truck(X_1, X_2, X_3)
  Req: at(X_1, X_3), at(X_2, X_3)
  Del: at(X_1, X_3)
  Add: in(X_1, X_2)

where at and in are binary fluent symbols. In words, the load-truck action requires both the package, represented by the variable X_1, and the truck, represented by X_2, to be at the same location, represented by X_3. The action removes the package from the location and places it, instead, in the truck. A ground instance of this action, say load-truck(pk3, w238, man), would load the particular package pk3 into the truck with license plate w238 when both items are in Manchester (man).

The size of such definition is 16 (1 action symbol + 4 fluent symbols + 11 variable occurrences). We can imagine that the planning domain also contains other actions to unload the truck and drive it from one location to another; as well as more object symbols to identify different packages, trucks and locations.

Definition 5. Let α and β be two distinct ground actions. We say that α *interferes with* β, if the action α deletes fluents that are either required or added by β (i.e. $\alpha_{\text{del}} \cap (\beta_{\text{req}} \cup \beta_{\text{add}}) \neq \emptyset$). We say that a pair of ground actions is *intefereing* if one of them interferes with the other. ■

Example 2. The ground action load-truck(pk3, w238, man) interferes with the other ground action load-truck(pk3, y659, man) since the former deletes the fluent at(pk3, man) while the later requires it. Note that this is how, implicitly, the functionality of the fluent at is preserved, i.e. no object is allowed to end up at two different places simultaneously.

Definition 6. Given a set of ground fluents $\mathbf{S}$ and a set of ground actions $\mathbf{A}$, we say that $\mathbf{A}$ *is executable in* $\mathbf{S}$ *and produces* $\mathbf{S}'$, denoted by $\mathbf{S} \xrightarrow{\mathbf{A}} \mathbf{S}'$, if:

- $\mathbf{A}$ does not contain interfering actions,
- $\mathbf{A}_{\text{req}} \subseteq \mathbf{S}$,
- $\mathbf{S}' = \mathbf{S} \setminus \mathbf{A}_{\text{del}} \cup \mathbf{A}_{\text{add}}$.

where $\mathbf{A}_{\text{req}} = \bigcup_{\alpha \in \mathbf{A}} \alpha_{\text{req}}$, etc. ■

Definition 7. A *planning problem* is given as a pair $\mathbf{I}, \mathbf{G}$ of sets of ground fluents, respectively known as the *initial* and *goal states* of the problem.

A *solution plan* for the problem is a sequence $\mathbf{A}_1, \ldots, \mathbf{A}_k$ of sets of ground actions such that the sequence $\mathbf{S}_1 \xrightarrow{\mathbf{A}_1} \mathbf{S}_2 \xrightarrow{\mathbf{A}_2} \cdots \xrightarrow{\mathbf{A}_k} \mathbf{S}_{k+1}$ holds, the set $\mathbf{I} = \mathbf{S}_1$ and $\mathbf{G} \subseteq \mathbf{S}_{k+1}$. ■

The kind of plans just defined are often known as *plans with parallel actions.* The semantics of such plans is that, at each step, the actions in a set $\mathbf{A}_i$ can be executed in any order (even simultaneously) while still reaching the same outcome. In our example one could simultaneously execute both load-truck(pk3, w238, man) and load-truck(pk4, w238, man) in order to load both packages pk3 and pk4 into the truck w238. Alternatively, a *linear plan* is a plan where each $\mathbf{A}_i$ is a singleton. Trivially, any plan with parallel actions can be converted into a linear plan just by sequencing parallel actions into an arbitrary order, e.g. first load package pk3, then load pk4.

4 Encoding of planning problems

In this section we will consider an encoding of planning problems into finite domain predicate logic. Given a planning domain and a bound k, we construct a set of constraints Γ_k whose models correspond to plans of length k. Linear plans of shorter lengths ($< k$) can also be encoded by allowing the use of a nop action that does nothing or, in plans with parallel actions, having steps where no action is executed (i.e. an empty $\mathbf{A}_i$).

Although fluents and actions were already defined as atoms in predicate logic, these predicate symbols will now play the role of constant symbols so that we can quantify over them in our encoding. For example, if $f(\overline{Y})$ is a fluent in a planning domain then the predicate $\mathsf{holds}(f, \overline{Y}, T)$ will be used to denote the fact that an instance of the fluent $f(\overline{Y})$ holds at a step T of the plan.

Note that this sort of encoding requires, however, all fluents (resp. actions) to have the same arity. This can be easily achieved by padding actions with additional variables (which will be unused in its fluents), and padding fluents in actions with some dummy constant symbol $o \in \mathcal{D}$.

We will split the encoding of a planning domain into four groups of clauses. The first group Bound_k specifies the length of the plans to be considered, the second group $\mathsf{Act}_{\mathcal{D}om}$ encodes the definitions of actions, the third $\mathsf{Prob}_{\mathbf{I},\mathbf{G}}$ encodes the initial and goal states of a particular problem instance, and the fourth and last one $\mathsf{Frame}_{\mathcal{D}om}$ encodes the frame conditions. Frame conditions are the ones responsible to state that all fluents whose status is not changed by the actions executed must remain unmodified. We will, actually, show two different encodings for frame conditions which can be used to obtain plans which are either linear or with parallel actions.

In the following we will use A, B as variables that stand for actions and F as a variable to represent a fluent. Also T is a variable that represents the *current step* in the plan, and U the *next step*. A number of constant symbols $\{s_0, \ldots, s_k\} \subset \mathcal{D}$ are used to denote the actual steps in the plan.

Definition 8. Given a positive number k, the set Bound_k is simply defined as the set containing $\mathsf{next}(s_i, s_{i+1})$ for every $i \leq 0 < k$. ■

This simple set with a size of $O(k)$ is used to define an order among steps in the plan and to determine, from each step, which is the next one. The following set encodes the actions available in the domain.

Definition 9. Given a planning domain $\mathcal{D}om$, the *domain definition* $\mathsf{Act}_{\mathcal{D}om}$ is the set that contains, for each action in the domain, the constraints:

$\mathsf{reqs}(a, \overline{X}, f, \overline{Y}\sigma)$	for each $f(\overline{Y})\sigma$ required by $a(\overline{X})$
$\mathsf{dels}(a, \overline{X}, f, \overline{Y}\sigma)$	for each $f(\overline{Y})\sigma$ deleted by $a(\overline{X})$
$\mathsf{adds}(a, \overline{X}, f, \overline{Y}\sigma)$	for each $f(\overline{Y})\sigma$ added by $a(\overline{X})$

together with the following three constraints that make actions have their corresponding preconditions and effects:

$$\mathsf{reqs}(A, \overline{X}, F, \overline{Y}) \land \neg\mathsf{holds}(F, \overline{Y}, T) \to \neg\mathsf{executes}(A, \overline{X}, T)$$
$$\mathsf{next}(T, U) \land \mathsf{adds}(A, \overline{X}, F, \overline{Y}) \land \mathsf{executes}(A, \overline{X}, T) \to \mathsf{holds}(F, \overline{Y}, U)$$
$$\mathsf{next}(T, U) \land \mathsf{dels}(A, \overline{X}, F, \overline{Y}) \land \mathsf{executes}(A, \overline{X}, T) \to \neg\mathsf{holds}(F, \overline{Y}, U)$$

Example 3. In our running example, the action load-truck would be encoded as:

$$\mathsf{reqs}(\mathsf{load\text{-}truck}, X_1, X_2, X_3, \mathsf{at}, X_1, X_3)$$
$$\mathsf{reqs}(\mathsf{load\text{-}truck}, X_1, X_2, X_3, \mathsf{at}, X_2, X_3)$$
$$\mathsf{dels}(\mathsf{load\text{-}truck}, X_1, X_2, X_3, \mathsf{at}, X_1, X_3)$$
$$\mathsf{adds}(\mathsf{load\text{-}truck}, X_1, X_2, X_3, \mathsf{in}, X_1, X_2)$$

Similar constraints are added for other actions in the domain. The last few constraints of $\mathsf{Act}_{\mathcal{D}om}$ would ensure that an action is not executed when one of its requirements does not hold or, if the action is executed, that fluents are added or deleted accordingly. It is also easy to see that $\mathsf{Act}_{\mathcal{D}om}$ has a size of $O(|\mathcal{D}om|)$.

We now move to the encoding of a problem instance using the set of constraints $\mathsf{Prob}_{\mathbf{I},\mathbf{G}}$. Typically, in propositional encodings of a planning problem, one has to completely specify the initial state **I** stating, for every ground fluent, whether $f(\bar{c})$ or $\neg f(\bar{c})$ should hold. To avoid this, we define a special action setup that adds all the ground fluents to be true at the initial state and does not require or delete anything. Quantifying over all fluents it is easy to express that "initially nothing holds" and then make the setup action execute at the step zero of the plan, the frame conditions will then ensure that everything not added by setup remains false.

Definition 10. Given a planning problem defined by an initial state **I** and goals **G**, the encoding of the problem instance $\mathsf{Prob}_{\mathbf{I},\mathbf{G}}$ is defined as the set of constraints:

$\neg\mathsf{holds}(F, \overline{Y}, s_0)$	
$\mathsf{adds}(\mathsf{setup}, \overline{X}, f, \bar{c})$	for every $f(\bar{c})$ in **I**
$\mathsf{executes}(\mathsf{setup}, \bar{o}, s_0)$	
$\mathsf{next}(T, U) \to \neg\mathsf{executes}(\mathsf{setup}, \overline{X}, U)$	
$\mathsf{holds}(f, \bar{c}, s_k)$	for every $f(\bar{c})$ in **G**

where $\bar{o}$ simply represents the sequence $o, \ldots, o$ of dummy constant symbols of the required length. ■

Example 4. Suppose that initially we have two packages in Manchester, a truck in London, and our goal is to get the packages to Edinburgh. This corresponds to $\mathbf{I} = \{\mathsf{at}(\mathsf{pk3}, \mathsf{man}), \mathsf{at}(\mathsf{pk4}, \mathsf{man}), \mathsf{at}(\mathsf{w238}, \mathsf{lon})\}$, $\mathbf{G} = \{\mathsf{at}(\mathsf{pk3}, \mathsf{edn}), \mathsf{at}(\mathsf{pk4}, \mathsf{edn})\}$ and would be encoded in the component $\mathsf{Prob}_{\mathbf{I},\mathbf{G}}$ as:

$\neg\mathsf{holds}(F, Y_1, Y_2, s_0)$
$\mathsf{adds}(\mathsf{setup}, X_1, X_2, X_3, \mathsf{at}, \mathsf{pk3}, \mathsf{man})$
$\mathsf{adds}(\mathsf{setup}, X_1, X_2, X_3, \mathsf{at}, \mathsf{pk4}, \mathsf{man})$
$\mathsf{adds}(\mathsf{setup}, X_1, X_2, X_3, \mathsf{at}, \mathsf{w238}, \mathsf{lon})$
$\mathsf{executes}(\mathsf{setup}, o, o, o, s_0)$
$\mathsf{next}(T, U) \rightarrow \neg\mathsf{executes}(\mathsf{setup}, X_1, X_2, X_3, U)$
$\mathsf{holds}(\mathsf{at}, \mathsf{pk3}, \mathsf{edn}, s_k)$
$\mathsf{holds}(\mathsf{at}, \mathsf{pk4}, \mathsf{edn}, s_k)$

The first constraint makes all fluents false at time s_0, then we have the definition of the setup action. A pair of constraints follow that make `setup` to execute at the first state, and only at that state. Finally we specify that the goals should hold at the final state s_k. Note again that we do not have to specify where packages *are not*, such as $\neg\mathsf{at}(\mathsf{pk3}, \mathsf{lon})$, or that the truck is empty (because there is nothing in it).

We finally proceed to describe the rules that actually encode the frame conditions and, at the same time, to disallow the execution of interfering actions. The following sections consider two alternatives that correspond to plans that are either linear or with parallel actions.

4.1 Linear plans

One possibility is to allow only one action to execute at a time, and the frame conditions can be directly expressed stating that the truth value of fluents not added or deleted by an action do not change. Moreover, in order to allow plans whose length is shorter than the bound k, a nop action that does nothing should be added to the definition of the planning domain.

Definition 11. Given a planning domain $\mathcal{D}om$, the *linear frame encoding* of the domain, denoted by $\mathsf{LFrame}_{\mathcal{D}om}$, is the set containing, for each action symbol $a \in \mathcal{A}$ and fluent $f \in \mathcal{F}$, the constraint

$$\mathsf{next}(T, U) \wedge \mathsf{executes}(a, \overline{X}, T) \wedge \textstyle\bigwedge_{\sigma \in \Xi_{a,f}} \overline{Y} \neq \overline{Y}\sigma \rightarrow \\ \mathsf{holds}(f, \overline{Y}, T) \leftrightarrow \mathsf{holds}(f, \overline{Y}, U)$$

and the pair of constraints

$$\exists A, \overline{X} \in \mathcal{A} \times \mathcal{O}^n.\, \mathsf{executes}(A, \overline{X}, T)$$
$$\mathsf{executes}(A, \overline{X}, T) \wedge \mathsf{executes}(B, \overline{Z}, T) \rightarrow A = B \wedge \overline{X} = \overline{Z}$$

where the set $\Xi_{a,f}$ contains all substitutions σ for which the fluent $f(\overline{Y})\sigma$ is either added or deleted by $a(\overline{X})$. ■

Example 5. In our example the linear frame conditions for the load-truck action would be expressed as follows:

$$\begin{aligned}&\mathsf{next}(T,U) \wedge \mathsf{executes}(\mathsf{load\text{-}truck}, X_1, X_2, X_3, T) \wedge \\ &\qquad \neg(Y_1 = X_1 \wedge Y_2 = X_3) \rightarrow \mathsf{holds}(\mathsf{at}, Y_1, Y_2, T) \leftrightarrow \mathsf{holds}(\mathsf{at}, Y_1, Y_2, U)\end{aligned}$$

$$\begin{aligned}&\mathsf{next}(T,U) \wedge \mathsf{executes}(\mathsf{load\text{-}truck}, X_1, X_2, X_3, T) \wedge \\ &\qquad \neg(Y_1 = X_2 \wedge Y_2 = X_3) \rightarrow \mathsf{holds}(\mathsf{in}, Y_1, Y_2, T) \leftrightarrow \mathsf{holds}(\mathsf{in}, Y_1, Y_2, U)\end{aligned}$$

In words these constraints state that, except for the package X_1 moved by the action, all other objects remain at their same locations and in their same containers. The last few constraints of $\mathsf{LFrame}_{\mathcal{D}om}$ encode the fact that one, and only one, ground action executes at any given time.

Note that this encoding requires $|\mathcal{A}|\,|\mathcal{F}|$ constraints to represent the frame conditions, where $|\mathcal{A}|$ (resp. $|\mathcal{F}|$) denotes the number of action (resp. fluent) symbols. Additionally, each fluent added or deleted by actions must appear represented as a substitution in the set $\varXi_{a,f}$ for one of such constraints. Therefore the set of constraints $\mathsf{LFrame}_{\mathcal{D}om}$ has a size of $O(|\mathcal{A}|\,|\mathcal{F}| + |\mathcal{D}om|)$.

4.2 Plans with parallel actions

Alternatively, several actions could be executed at once as long as they do not interfere with each other. We consider an *explanatory* encoding following ideas proposed by Haas [4], Schubert [8] and later applied in the propositional case by Kautz et al. [6]; where it is expressed that, if a fluent changes its value from one step to another, then one of the actions that modify it must have been executed.

Definition 12. Given a planning domain $\mathcal{D}om$, the *parallel frame encoding* of the domain, denoted by $\mathsf{PFrame}_{\mathcal{D}om}$, is the set containing, for each fluent $f \in \mathcal{F}$, the constraints:

$$\begin{aligned}&\mathsf{added}(f, \overline{Y}, T) \rightarrow \textstyle\bigvee_{(a,\sigma)\in\varDelta_f} \exists \overline{X} \in \mathcal{O}^n.(\mathsf{executes}(a, \overline{X}, T) \wedge \overline{Y} = \overline{Y}\sigma)\\ &\mathsf{deleted}(f, \overline{Y}, T) \rightarrow \textstyle\bigvee_{(a,\sigma)\in\nabla_f} \exists \overline{X} \in \mathcal{O}^n.(\mathsf{executes}(a, \overline{X}, T) \wedge \overline{Y} = \overline{Y}\sigma)\end{aligned}$$

together with the three constraints

$$\begin{aligned}&\mathsf{next}(T,U) \wedge \neg\mathsf{holds}(F, \overline{Y}, T) \wedge \quad \mathsf{holds}(F, \overline{Y}, U) \rightarrow \mathsf{added}(F, \overline{Y}, T)\\ &\mathsf{next}(T,U) \wedge \quad \mathsf{holds}(F, \overline{Y}, T) \wedge \neg\mathsf{holds}(F, \overline{Y}, U) \rightarrow \mathsf{deleted}(F, \overline{Y}, T)\\ &\mathsf{dels}(A, \overline{X}, F, \overline{Y}) \wedge \mathsf{reqs}(B, \overline{Z}, F, \overline{Y}) \wedge \\ &\qquad\qquad \mathsf{executes}(A, \overline{X}, T) \wedge \mathsf{executes}(B, \overline{Z}, T) \rightarrow A = B \wedge \overline{X} = \overline{Z}\end{aligned}$$

where the set $\varDelta_f$ (resp. ∇_f) contains the pair (a, σ) whenever the fluent $f(\overline{Y})\sigma$ is added (resp. deleted) by the action $a(\overline{X})$. ■

Example 6. In this case, the predicates added and deleted are defined for each fluent. Consider for instance the following constraint that encodes the frame conditions for the fluent $\mathsf{at}(Y_1, Y_2)$:

$$\begin{aligned}&\mathsf{added}(\mathsf{at}, Y_1, Y_2, T) \rightarrow \\ &\qquad\quad \exists \overline{X} \in \mathcal{O}^3.(\mathsf{executes}(\mathsf{unload\text{-}truck}, \overline{X}) \wedge Y_1 = X_1 \wedge Y_2 = X_3)\\ &\qquad \vee\, \exists \overline{X} \in \mathcal{O}^3.(\mathsf{executes}(\mathsf{drive\text{-}truck}, \overline{X}) \wedge Y_1 = X_1 \wedge Y_2 = X_3)\end{aligned}$$

If a fluent $\mathsf{at}(Y_1, Y_2)$ is added at some state, then it must be the case that either a package $Y_1 = X_1$ was unloaded at a location $Y_2 = X_3$ (from some truck X_2) or, similarly, a truck was driven to that location from another.

The last few constraints trigger the predicates added and deleted, whenever a change in the truth value of a fluent occurs, in order to search for an explanation of such change. The final constraint disables the execution of two actions when one deletes a requirement of the other and, therefore, they are interfering. It is also not possible to execute two actions such that one deletes the fluent added by the other, a contradiction will occur in $\mathsf{Act}_{\mathcal{D}om}$ when both actions try to assign contradictory values to the fluent.

Note that, in this case, the number of clauses in $\mathsf{PFrame}_{\mathcal{D}om}$ is linear with respect to the number of fluent symbols in $\mathcal{F}$. Moreover, the size of the clauses only depends on the number of actions that could add or delete a given fluent. Overall, $\mathsf{PFrame}_{\mathcal{D}om}$ has only a size of $O(|\mathcal{D}om|)$ and does not directly depend on the number of actions or fluents as in the previous case.

Theorem 4. *Given a planning domain $\mathcal{D}om$, a problem $\mathbf{I}, \mathbf{G}$ and a bound k, the finite domain predicate formula $\mathsf{Bound}_k \wedge \mathsf{Act}_{\mathcal{D}om} \wedge \mathsf{Prob}_{\mathbf{I},\mathbf{G}} \wedge \mathsf{Frame}_{\mathcal{D}om}$, where $\mathsf{Frame}_{\mathcal{D}om}$ is either $\mathsf{LFrame}_{\mathcal{D}om}$ or $\mathsf{PFrame}_{\mathcal{D}om}$, is satisfiable if and only if the planning problem has a solution plan, respectively linear or with parallel actions, of length $\leq k$.*

Proof. It can be shown that if an interpretation $\mathcal{I}$ is a model of the encoding, then the plan where $\mathbf{A}_i = \{a(\bar{c}) \mid \mathcal{I} \models \mathsf{executes}(a, \bar{c}, s_i)\}$, for $1 \leq i \leq k$, is a valid solution to the planning problem.

Conversely it can be shown that, if $\mathbf{A}_1, \ldots, \mathbf{A}_{k'}$ is a solution plan (linear or with parallel actions) with $k' < k$, then an interpretation $\mathcal{I}$ can be built, giving appropriate values to predicates, such that $\mathcal{I}$ is a model of the encoding. □

5 Conclusions

In this paper we have introduced the finite domain predicate logic, which corresponds to a decidable fragment of first order logic with features such as equality and finite quantification. Formulae in this logic are non-propositional, but its models can be interpreted in a finite Herbrand universe. We also show that formulae in this logic can be linearly translated to the Bernays-Schönfinkel class of formulae, which also corresponds to the category of effectively propositional problems of the CASC system competition [9].

The motivation for developing such a logic is that it enables us to succinctly and naturally encode problems from applications. In particular we show how planning problems, including their frame conditions, can be easily encoded within the proposed logic. Moreover, the size of the generated formula is linear with respect to size of a standard description, e.g. in the STRIPS language, of the original planning problem. This is in contrast with propositional encodings where the size of the resulting formula is often exponential in the size of the input.

Furthermore, we also show that reasoning with effectively propositional formulae can be exponentially more efficient than in the propositional setting. We show in particular a family of unsatisfiable formulae whose refutation proofs using first order resolution can be exponentially shorter than any propositional resolution proof. This serves to suggest that, in principle, the use of a finite domain predicate encoding can be useful both to obtain more compact representations of problems and to solve them more efficiently.

On the other hand, the ideas presented here might also turn out to be useful for propositional SAT-based approaches. Since the problem of finding optimised propositional encodings, including but not limited to planning, is reduced to finding an appropriate instantiation of the obtained finite domain formula.

We think that our work is of great value to the automated reasoning community since it can be useful to provide a new and relevant source of benchmarks for developers of first order reasoners, particularly those geared towards the effectively propositional fragment.

As an immediate future work, we will be working on the implementation of a tool able to read planning problems and domains in the STRIPS language and then translate them into effectively propositional formulae in the TPTP format. Another interesting issue left for future work, is to study how existing reasoning approaches deal with this kind of formulae and tune them to solve these problems in a more efficient way.

References

[1] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 26, pages 19–99. Elsevier Science, 2001.

[2] R.E. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.

[3] C. Green. Application of theorem proving to problem solving. In *Proceedings of IJCAI-69*, pages 219–239, Washington, D.C., 1969.

[4] A.R. Haas. The case for domain specific frame axioms. In F.M. Brown, editor, *The Frame Problem in Artificial Intelligence: Proceedings of the 1987 Workshop*, pages 343–348, San Mateo Calif., 1987. Morgan Kaufmann Publishers, Inc.

[5] Henry Kautz and Bart Selman. Planning as satisfiability. In *ECAI '92: Proceedings of the 10th European conference on Artificial intelligence*, pages 359–363. John Wiley & Sons, Inc., 1992.

[6] Henry Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. In *Proc. of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR-96)*, Boston, MA, 1996.

[7] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986. ISSN 0747-7171.

[8] L.K. Schubert. Monotonic solution of the frame problem in the situation calculus: An efficient method for worlds with fully specified actions. In H. Kyburg, R. Loui, and G. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer, Dortrecht, 1990.

[9] G. Sutcliffe and C. Suttner. The state of CASC. *AI Communications*, 19(1): 35–48, 2006.