Artificial Intelligence Research and Development
M. Villaret et al. (Eds.)

doi:10.3233/FAIA210117

Exploring Lifted Planning Encodings in Essence Prime

Joan ESPASA, [a] Jordi COLL, [b] Ian MIGUEL, [a] and Mateu VILLARET [c]
[a] *School of Computer Science, University of St Andrews, St Andrews KY16 9SX, UK*
e-mail: {jea20,ijm}@st-andrews.ac.uk
[b] *Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France*
e-mail: jordi.coll@lis-lab.fr
[c] *Departament d'Informàtica, Matemàtica Aplicada i Estadística*
Universitat de Girona, E-17003 Girona, Spain
e-mail: mateu.villaret@udg.edu

Abstract. State-space planning is the *de-facto* search method of the automated planning community. Planning problems are typically expressed in the Planning Domain Definition Language (PDDL), where action and variable templates describe the sets of actions and variables that occur in the problem. Typically, a planner begins by generating the full set of instantiations of these templates, which in turn are used to derive useful heuristics that guide the search. Thanks to this success, there has been limited research in other directions.

We explore a different approach, keeping the compact representation by directly reformulating the problem in PDDL into ESSENCE PRIME, a Constraint Programming language with support for distinct solving technologies including SAT and SMT. In particular, we explore two different encodings from PDDL to ESSENCE PRIME, how they represent action parameters, and their performance. The encodings are able to maintain the compactness of the PDDL representation, and while they differ slightly, they perform quite differently on various instances from the International Planning Competition.

Keywords. Automated Planning, Constraint Programming, Modelling, Reformulation

1. Introduction

Given a model of the environment, a planning problem asks us to find a sequence of actions that leads from an initial state to a given goal state. These models are typically expressed in the Planning Domain Definition Language [1] (PDDL). The user describes the problem in terms of predicates, actions and functions with parameters. In turn, these parameters can be instantiated with a set of defined objects.

A simple example of a planning problem is a logistics problem, expressed in PDDL in Figure 1, where we must transport Bob and Alice from the city of Barcelona to the airport, so they are finally able to embark on a plane that will take them home. The initial state is that both Alice and Bob are at Barcelona, the goal is having them embarked in a plane at the Airport, and the possible actions are `move` (change person of location) and `embark` (person into plane at a location).

In spite of having a compact model representation, having read the model, a planner typically begins by producing a totally *grounded* representation. The action of grounding (or instantiation) will replace all variables that represent parameters in actions by their possible values, creating all the possible instantiations of the actions. After grounding, no variables are left free and all valid instantiations of predicates and functions in the actions are computed. The size of the fully grounded planning problem is exponential in the maximum number of arguments of all the actions.

Depending on the original problem and how the task is grounded, this growth can result in an instance that cannot be efficiently handled. There have been approaches that try to alleviate this grounding problem in various ways. For example, one could ground only relevant parts of the problem [2,3], make clever representations of actions [4] or simplify the input problem [5]. The opposite of grounded planning would be *lifted* planning, where grounding is fully avoided. Grounding is normally seen as a necessary step, and there are very few approaches to lifted planning that skip grounding entirely [6,7,8]. These approaches are not that popular mainly due to the efficiency of computing informative heuristics on a grounded representation, which are difficult to compute at the lifted level. Also, reasoning at a more abstract level is typically more difficult.

Herein we try to avoid grounding as much as possible by using the expressivity of ESSENCE PRIME [9], a declarative constraint modelling language. Moreover, we take advantage of SAVILE ROW [10], a sophisticated constraint reformulation tool supporting ESSENCE PRIME that is able to generate SAT and SMT [11], or CSP [12] instances.

Our contributions in this paper are two different lifted encodings from PDDL to ESSENCE PRIME, which differ in how they represent action parameters. The encodings maintain the compactness of the PDDL representation, and while they differ slightly, they perform quite differently. We also report results with backend solvers for SAT, SMT and for a lazy clause generation [13] (LCG) constraint solver. The rest of the paper proceeds as follows. In Section 2 we recall the theoretical framework. In Section 3 we propose the encodings from PDDL to ESSENCE PRIME. Section 4 is devoted to experimental evaluation of the encodings. Finally, Section 5 discusses future work and concludes.

2. Preliminaries on Automated Planning

This paper considers numeric planning problems, which extend propositional planning with numeric state variables. We formally define the numeric planning problem only in terms of the grounded representation of the problem.

Definition 1 (Numeric Planning Problem)**.** A planning problem can be defined as a tuple $\prod = \langle B, O, F, X, A, I, G \rangle$, where: B is a set of names for all the objects, O is a set of object state variables, F is a set of propositional state variables, X is a set of numeric state variables, A is a set of actions, I is the initial state and G is the goal.

An action $a \in A$ is defined as a tuple $a = \langle Pre_a, Eff_a \rangle$, where Pre_a refers to the precondition and Eff_a to the effects of the action.

Definition 2 (State)**.** Given a planning problem $\prod$, a *state* is a variable-assignment (or valuation) function over state variables $O \cup F \cup X$, which maps each $o \in O$ to an object in B, each $f \in F$ into a truth value, and each $x \in X$ to an integer. A state is represented a set of ordered pairs $\{(v_1, z_1), (v_2, z_2), \ldots, (v_n, z_n)\}$, where each v_i is the variable and z_i the value mapped to it.

An *object condition* has the form $\zeta \otimes b$, where ζ is an expression over O, $\otimes \in \{=, \neq\}$ and b is an object in B. A *numeric condition* has the form $\zeta \otimes k$, where ζ is a linear integer arithmetic expression over X, $\otimes \in \{\leq, <, =, >, \geq\}$ and k is an integer constant.

Preconditions (*Pre*) and the goal G are sets (conjunctions) of object conditions, numeric conditions and propositions. Action effects (*Eff*) are sets of assignments to propositional variables, assignments to object variables and increase/decrease/assign a numeric variable by a numeric expression. A *conditional effect* is a pair $\langle c, e \rangle$ where c is a set of object, numeric and propositional conditions; and e is an effect. e is applied only if c is satisfied in the state where the action is applied.

An action a is *applicable* in a state s only if its preconditions are satisfied in s ($s \models Pre_a$) and the applied numeric, object and propositional effects do not induce conflicting assignments. The outcome after the application of an action a will be the state where variables that are assigned in Eff_a take their new value, and variables not referenced in Eff_a keep their current values.

A sequence of actions $\langle a_0, \ldots, a_{n-1} \rangle$ is called a *plan*. We say that the application of a plan starting from the initial state I brings the system to a state s_n. If each action is applicable in the state resulting from the application of the previous action and the final state satisfies the goal (i.e., $s_n \models G$), the sequence of actions is a a *valid plan*. A planning problem has a solution if a valid plan can be found for the problem.

3. Encodings

In this section we propose various encodings for a numeric planning problem. First we will explain how the planning as satisfiability approach works, then in what kind of input we will receive the planning problem and finally the proposed encodings.

3.1. Planning as Satisfiability

As is typical in the planning as SAT or as CSP approaches [14,15,16], we will solve the planning problem by considering a sequence of CSPs ϕ_0, ϕ_1, ϕ_2, ..., where ϕ_i encodes the existence of a plan that allows to reach a goal state from the initial state in i steps. The solving procedure will test the satisfiability of ϕ_0, ϕ_1, ϕ_2, and so on, until a satisfiable formula ϕ_n is found, proving the existence of a valid plan of n steps.

Each ϕ formula will need variables to represent the state for each step and need to define the values of the variables in the initial step. Then, it will also need some variables to represent which action is executed at each step. We will need to make sure that if an action is executed, its precondition holds with respect to the problem variables. We will need to make sure that the goal conditions are met and we will do it by adding some constraints on the variables representing the state of the final step. Finally, we will need to make *frame axioms* explicit, i.e. constraints that specify that if no action has modified a variable, it keeps its value between steps. Semantics such as the $\forall$ or $\exists$-step [17] allow parallel actions, but here just one action will be executed per time step.

3.2. Planning Domain Definition Language (PDDL)

In contrast to the formal definition of a planning problem given in Section 2, PDDL allows the specification of problems in a lifted manner. Although being normally represented this way, most solving approaches ground the problems.

```
(define (domain transport)
 (:types person aircraft - locatable
         location - object)
 (:predicates (at ?p - locatable ?l - location)
              (in ?p - person ?a - aircraft))
 (:functions (seats ?p - aircraft) - number)
 (:action move
    :parameters (?p - person ?from ?to - location)
    :precondition (at ?p ?from)
    :effect (and (not (at ?p ?from)) (at ?p ?to)))
 (:action embark
    :parameters (?p - person ?l - location ?a - aircraft)
    :precondition (and (at ?p ?l) (at ?a ?l) (> (seats ?a) 0))
    :effect (and (not (at ?p ?l)) (in ?p ?a) (decrease (seats ?a) 1))))

(define (problem example)
 (:domain transport)
 (:objects plane - aircraft
           Bob Alice - person
           Barcelona Airport - location)
 (:init (at Bob Barcelona) (at Alice Barcelona)
        (at plane Airport) (= (seats plane) 2))
 (:goal (and (in Bob plane) (in Alice plane))))
```

Figure 1. Domain and problem file in PDDL, representing the problem of moving Bob and Alice from Barcelona to a plane in the airport. A valid plan for the problem would be: `(move Bob Barcelona Airport)`, `(move Alice Barcelona Airport)`, `(embark Bob Airport plane)` and `(embark Alice Airport plane)`.

A *fluent*, in the area of automated planning, refers to a variable that represents some attribute of the problem and changes over time. Roughly speaking, our framework will be numeric planning. More concretely, our formalism will derive from PDDL 2.1 [18], without temporal semantics or metric optimizations. We also consider functional strips semantics [19], incorporated in the recent revisions of the PDDL. This means that, apart from reasoning with integer fluents, we will be able to have actions that work with objects and refer to attributes of these objects. Therefore, a fluent declared as `(location ?p - object) - place` will be able to express where objects are, and expressions like `(= (location plane) (location person))` or `(> (fuel plane) 10)` will be valid.

Even though planning formalisms do not consider templates, they are widely used in PDDL to make the representation compact. Types are also used in PDDL to make the problem more readable and to give more information to the planners. It can be seen in Figure 1 how types, templates for actions, predicates and functions are expressed. As our input will be a problem defined in the PDDL language, we will need to directly consider them. In fact, the instantiations of the predicate templates will correspond to the predicate state variables of the planning problem at hand, and the instantiations of the function templates will correspond to the object and numeric state variables of the planning problem, depending on its return type.

Templates can be *state variable templates* or *action templates*. These are comprised of a name and a sequence of typed parameters, or "ordinary" variables. For example, consider `(location ?p - object) - place`, being an object state variable template. Its name is `location` and its parameters, the sequence [`?p`], where the only parameter

`?p` has the name `p` and the `object` type. The domain of this object state variable is the set of objects with type `place` in the problem.

For instance, in the PDDL specification, expressions such as preconditions and effects can also contain variables, belonging to the action template parameters. For example, the effect `(and (not (at ?p ?from)) (at ?p ?to))` belonging to the `move` action template in Figure 1 contains three variables: `p`, `from` and `to`.

3.3. Basic Encoding

In this section we describe formulas ϕ_h, that is, the existence of a valid plan with h actions. Again, our purpose in this work is to encode PDDL instances into ESSENCE PRIME in a lifted manner. Roughly, a grounded representation would have a Boolean variable stating whether action $move\_alice\_Barcelona\_airport^t$ is performed in a given time step t. Instead, for each time step t, we will have an integer variable stating which action template is applied and an integer variable per parameter of each action template stating what particular object is used as parameter of that action. Moreover, and for each time step t, we will also have a Boolean or Integer ESSENCE PRIME variable (CP variable) for each concrete instantiation of each state variable template.

To express the encoding, we will need some auxiliary definitions. Let E be the set of types specified in the PDDL model. Each object $b \in B$ has a type associated with it. Also, each type $e \in E$ has a domain associated to it, being $Domain_e \subseteq B$. Let A_T be the set of action templates in the PDDL problem. Similarly, O_T, F_T and X_T will be the sets of object, propositional and numeric state variable templates, respectively.

Let V be the set $O \cup F \cup X$, representing the set of all state variables, without taking their type into account. V_T will represent the set of all state variable templates. For $x \in A_T \cup V_T$, let $Parameters_x$ be the sequence $[z_1, \ldots, z_n]$, representing the parameters of the template. For each parameter z_i, let $Type_{z_i}$ be the type associated to z_i, and $Name_{z_i}$ its name. Let $l(k) \to \mathbb{Z}$ be an injective function defined for all $k \in B \cup A$. It serves as a labelling function, that maps an object or action to a unique integer. This will be useful to later encode objects and object state variables as integers and integer state variables respectively. We will start by introducing the following CP variables:

$$state_v^t \qquad \forall v \in V, \forall t \in 0..h \tag{1}$$

$$action^t \qquad \forall t \in 1..h \tag{2}$$

$$param_{a,i}^t \qquad \forall a \in A_T, \forall i \in Parameters_a, \forall t \in 1..h \tag{3}$$

Variables $state_v^t$ hold the value of state variable v in step t. This representation corresponds to a new CP variable for each grounded state variable. Variables $action^t$ express which action is scheduled at time step t. The domain of these $action^t$ variables is $\{l(a) \mid a \in A_T\}$, being the set of integers the labelling function l assigns to the problem action templates. Finally, variables $param_{a,i}^t$ denote the value of i-th parameter in action template a at each step t. Each of these variables will have a domain of the parameter type. Note that variables introduced in (2) and (3) correspond to the action templates. With this representation there is no need to ground all the possible instantiations of the actions, and the solver will be responsible for choosing which action template is executed and with which parameters. We state initial and goal states:

$$state_v^0 = z \qquad \forall (v,z) \in I \tag{4}$$

$$g^h \qquad \forall g \in G \tag{5}$$

where G is a conjunction of conditions on state variables, and g^h is the ESSENCE PRIME translation of these conditions on CP variables $state_v^h$ for all variables v in the conditions of G. Note that the initial state must be fully specified.

Frame axioms express that, if a given state variable has changed from one time step to the next, it is because an action that is able to change it has been executed.

$$state_v^{t-1} \neq state_v^t \rightarrow \bigvee_{\substack{\forall a \in A_T, \\ \forall m \in modify(a,v)}} \left(action^t = l(a) \wedge \bigwedge_{\forall (j,o) \in m} param_{a,j}^t = l(o) \right) \begin{array}{l} \forall t \in 1..h, \\ \forall v \in V \end{array} \tag{6}$$

Given an action template a and a state variable v, the function $modify(a,v)$ returns the set of all combinations of parameter assignments (expressed as a pair (j,o)) that make action a modify variable v. For instance, the state variable `at(Bob,Barcelona)` is modified by action template `move`, with the following set of parameter assignments:

$$\{\{(\texttt{p},\texttt{Bob}),(\texttt{from},\texttt{Barcelona}),(\texttt{to},\texttt{airport})\},\ \{(\texttt{p},\texttt{Bob}),(\texttt{from},\texttt{airport}),(\texttt{to},\texttt{Barcelona})\},...\}$$

Finally, actions are expressed

$$action^t = l(a) \rightarrow Pre_a^t \wedge Eff_a^t \qquad \forall a \in A_T, \forall t \in 1..h \tag{7}$$

Preconditions are sets of conditions and effects are sets of assignments. When translating Pre_a^t and Eff_a^t into ESSENCE PRIME, we use the *element* global constraint to access the corresponding state variables according to the values given to the action parameters. The translation of conditions and state variable assignments to ESSENCE PRIME is straightforward. However, conditions and right hand sides of assignments will consult the state variables of time $t-1$, and left hand side of the assignments will update state variables of time t. For instance, when considering the effect on the number of free seats in the `embark` action: `seats[embark_a[k],k] = seats[embark_a[k],k-1]-1`.

3.4. Encoding Compaction

Approximations such as the $\forall$-step or $\exists$-step semantics [17] allow parallel actions as long as they are not interfering. For now our encoding assumes that one action will be executed per time step. With one action executed per time step, we can see that most of the variables from (3) are rarely used. That is, only the parameters belonging to the selected action are used, and the others are ignored. Here we introduce two variants of the encoding with the aim of reducing the total number of variables: *Type sharing* and *Max Parameters*. They differ in how parameters are treated, as shown in Figure 2.

Before explaining the encodings, we introduce the concept of a *substitution* (or *renaming*) σ: a partial mapping from variables to variables. It can be represented as a function by a set of bindings of variables to variables. That is, if $\sigma = \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$, then $\sigma(x_i) = y_i$ for all i in $1..n$, and $\sigma(x) = x$ for every other variable. Using an infix

Original PDDL representation

`fly(?p - plane ?from ?to - loc)` `unload(?p - plane ?x - package)`
`load(?p - plane ?x - package)`

Standard encoding	**Type Sharing**	**Max parameters**
$\texttt{fly}(p_{fly,1}, p_{fly,2}, p_{fly,3})$	$\texttt{fly}(p_{plane,1}, p_{loc,1}, p_{loc,2})$	$\texttt{fly}(p_1, p_2, p_3)$
$\texttt{unload}(p_{unload,1}, p_{unload,2})$	$\texttt{unload}(p_{plane,1}, p_{package,1})$	$\texttt{unload}(p_1, p_2)$
$\texttt{load}(p_{load,1}, p_{load,2})$	$\texttt{load}(p_{plane,1}, p_{package,1})$	$\texttt{load}(p_1, p_2)$
Set of parameters	**Set of parameters**	**Set of parameters**
$\{p_{fly,1}, p_{fly,2}, p_{fly,3}, p_{unload,1}, p_{unload,2}, p_{load,1}, p_{load,2}\}$	$\{p_{plane,1}, p_{package,1}, p_{loc,1}, p_{loc,2}\}$	$\{p_1, p_2, p_3\}$

Figure 2. Example of how parameters are shared in the various encodings for the planes domain. For each encoding (standard, type sharing, max parameters), the corresponding set of parameters is shown below.

notation and given any expression τ containing variables, $\tau\sigma$ is τ with all the contained variables replaced, as specified by σ. For example, given a substitution $\sigma = \{p \mapsto q\}$ and the term representing an effect $\tau =$ `(and (not (at ?p ?from)) (at ?p ?to))`, the result of $\tau\sigma$ would be `(and (not (at ?q ?from)) (at ?q ?to))`.

3.4.1. Type Sharing

Although actions can have many parameters, they typically have few parameters of the same type. Therefore, in this encoding each action parameter of a given type is replaced by a new parameter that is shared by all the actions that need a parameter of that type.

Let C_e for each $e \in E$ be the maximum number of parameters on all actions that share type e. Then, variables introduced in (3) are substituted with

$$param^t_{e,i} \qquad \forall e \in E, \forall i \in 1..C_e, \forall t \in 1..h \tag{8}$$

Example 1. If the PDDL action that has most parameters with the `place` type is an action such as `move(?p - person, ?from - place, ?to - place)`, then $C_{place} = 2$. Then, the previous Equation will introduce parameter variables $param^t_{person,1}$, $param^t_{place,1}$ and $param^t_{place,2}$ for each time step.

Given an action template $a \in A_T$, a parameter $q \in Parameters_a$ and its type $Type_q \in E$, let $pos(q,a) = [z \mid z \in Parameters_a, Type_z = Type_q]$. That is, the subsequence of parameters of a that have the same type as q.

Then, we can define a substitution σ_a for every action $a \in A_T$, such that

$$\sigma_a = \{param_{a,q} \mapsto param_{e,i} \mid q \in Parameters_a, e = Type_q, i \in 1..|pos(q,a)|, pos(q,a)[i] = q\} \tag{9}$$

Finally, to Equation (7) is modified to use these new parameter variables

$$action^t = l(a) \rightarrow Pre^t_a\sigma_a \wedge Eff^t_a\sigma_a \qquad \forall a \in A_T, \forall t \in 1..h \tag{10}$$

Following Example 1, this will substitute all appearances on the *Pre* and *Eff* of $?p$ by $param_{person,1}$ and so on.

	depots(3)		driverlog(10)		planes(8)		zenotravel(9)		total
RanTanPlan-SMT(LIA)	1	(4809.4)	7	(3174.8)	3	(4600.2)	8	(945.4)	19
SR-SAT T. Sharing	3	(1354.3)	**10**	**(53.6)**	8	(758.9)	4	(4049.6)	25
SR-SAT Max. Par	2	(2614.1)	10	(814.1)	0	(7200.0)	7	(2143.5)	19
SR-LCG T. Sharing	2	(3293.8)	7	(2631.1)	3	(4708.8)	7	(1633.2)	19
SR-LCG Max. Par	1	(5763.0)	0	(*)	0	(*)	4	(4087.1)	5
SR-SMT(BV) T. Sharing	**3**	**(889.9)**	10	(205.0)	**8**	**(462.2)**	**9**	**(142.8)**	30
SR-SMT(BV) Max. Par	0	**(*)**	0	(*)	0	(*)	0	(*)	0

Table 1. For each domain and configuration: left, number of solved instances; right, mean solving time in seconds, counting timeouts as 7200 seconds. We only consider the subset of instances solved by some setting, and the subset sizes are next to domain name. All instances for cells with (*) have run out of memory.

3.4.2. Max parameters

An alternative approach is to share parameters independently of their types. That is, instead of dedicated parameter variables for each action, we will only declare n parameters, where n is equal to the number of parameters of the action with most parameters. Formally, $n = max(\{|Parameters_a| \mid a \in A_T\})$. These parameters will be representing different types depending on which action is executed. Therefore, the domain of each one will be the union of all possible objects. We will again substitute variables in (3) by

$$param^t_q \qquad \forall q \in 1..n, \forall t \in 1..h \tag{11}$$

Now, let σ_a be a substitution for every action $a \in A_T$, such that $\sigma_a = \{param_{a,q} \mapsto param_q | q \in Parameters_a\}$. This substitution will replace the mentioned parameters in the action by the new declared parameters in (11). Finally, Equation (7) is also modified to use these new variables

$$action^t = l(a) \rightarrow Pre^t_a \sigma_a \wedge Eff^t{}_a \sigma_a \qquad \forall a \in A_T, \forall t \in 1..h \tag{12}$$

To improve the encoding, if using a CSP solver as a backend, a *table* constraint can be added to the ESSENCE PRIME model to limit the possible values of the parameters, depending on the action chosen. Hence, once an action has been decided, the domains of the parameters are restricted to its declared types.

4. Experimental Evaluation

In this section we evaluate the performance of the presented encodings by solving a set of numeric planning problems coming from the third IPC [20]. These domains contain integer numeric fluents without quantified preconditions, as the rest of the domains contain features that we still do not support. These domains are: *Zenotravel*, *Driverlog*, *Depots*. The *Planes* domain from [21] is also considered since it has an interesting numerical component. Although some domains give various optimization criteria, we only consider the problem of finding a valid plan minimizing the total makespan, i.e. number of steps. As noted, our approach reformulates the PDDL description to the ESSENCE PRIME language. In turn, this ESSENCE PRIME model is given as input to SAVILE ROW [10] to

generate a SAT, SMT or CSP model. Finally we use Glucose [22], Chuffed[1] and Boolector [23] as the backend solvers. We validated the usefulness of the SAVILE ROW preprocessing steps suchs as common subexpression elimination [24,25] or symmetry breaking capabilities by turning them off and determining that solving times were significantly increased, at least by a factor of two. To compare the presented encodings with a fully grounded approach, we use the linear planning as SMT encoding provided by RanTanPlan [21]. The experiments were run on a AMD Opteron® 6272 Processor. Each process was given a limit of 4GB of memory and 1 hour timeout.

We do not consider the basic encoding without compacting action parameters, as it behaves worse than the two proposed improvements. Table 1 shows the number of solved instances and average solving time for each domain and each considered approach. We can observe that there are differences in the performance of the different approaches in the different domains, but the SR-SMT approach is generally better than the other ones.

The *Depots* domain seems too big, as all the approaches are only able to return a solution for a very few instances. If we look at the *Driverlog*, *Zenotravel* and *Planes* domains, the different approaches differ between them. The lifted approaches are generally better than RanTanPlan which uses grounding, but *Zenotravel* is harder for the lifted approaches except for *SR-SMT(BV)*. The Type Sharing encoding is better in general for all solving approaches and all considered problems. Even though the Max. Parameters encoding generates fewer parameters, it uses the maximum possible domain size for each parameter. This could imply that parameters with small domain are encoded using unnecessarily large domains, and could be the reason why we have many memory outs with Max. Par. We have observed that the action parameters of *Zenotravel* instances have relatively balanced domain sizes in comparison with the other domains, explaining why *SR-SAT Max. Par* does not behave worse than *SR-SAT T. Sharing* in *Zenotravel*.

Summing up, we observe that lifted encodings with T. Sharing are generally better than the grounded encoding, and in particular *SR-SMT(BV) T. Sharing* is the overall best approach. However, further experiments are needed to identify to what extent this is due to using a lifted encoding. Other aspects such as the used SMT theory or the reformulations performed by SAVILE ROW could also play an important role.

5. Conclusions and Future Work

We have presented two lifted approaches to encoding planning problems as CSP, and experimented with different solving back ends via SAVILE ROW. One configuration outperformed the fully grounded linear encoding of RanTanPlan, which also solves numeric planning as SMT. The relative performance of the two encodings depends on the number of action parameters and their domain sizes. In future work, a preprocess could select an encoding based on problem structure. The encodings could also be improved by considering the symmetries between successive application of different actions, or by incorporating the application of various actions in the same step.

Acknowledgements This work is supported by UK EPSRC EP/P015638/1 and EP/V027182/1, by the MICINN/FEDER, UE (RTI2018-095609-B-I00), by the French Agence Nationale de la Recherche, reference ANR-19-CHIA-0013-01, and by Archimedes institute, Aix-Marseille University.

[1] https://github.com/chuffed/chuffed

References

[1] Haslum P, Lipovetzky N, Magazzeni D, Muise C. An Introduction to the Planning Domain Definition Language. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers; 2019.

[2] Helmert M. Concise finite-domain representations for PDDL planning tasks. Artificial Intelligence. 2009;173(5-6):503–535.

[3] Gnad D, Torralba A, Domınguez M, Areces C, Bustos F. Learning How to Ground a Plan - Partial Grounding in Classical Planning. In: AAAI; 2019. p. 7602–7609.

[4] Areces C, Bustos F, Dominguez MA, Hoffmann J. Optimizing Planning Domains by Automatic Action Schema Splitting. In: ICAPS; 2014. p. 11–19.

[5] Masoumi A, Antoniazzi M, Soutchanski M. Modeling Organic Chemistry and Planning Organic Synthesis. In: GCAI; 2015. p. 176–195.

[6] Penberthy JS, Weld DS. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In: KR'92; 1992. p. 103–114.

[7] Bofill M, Espasa J, Villaret M. Efficient SMT Encodings for the Petrobras Domain. In: ModRef. Lyon, France; 2014. p. 68–84.

[8] Correa AB, Pommerening F, Helmert M, Francès G. Lifted Successor Generation Using Query Optimization Techniques. In: ICAPS; 2020. p. 80–89.

[9] Nightingale P, Rendl A. Essence' Description. 2016;ArXiv:1601.02865 [cs.AI].

[10] Nightingale P, Akgün Ö, Gent IP, Jefferson C, Miguel I, Spracklen P. Automatically improving constraint models in Savile Row. Artificial Intelligence. 2017;251:35–61.

[11] Biere A, Heule M, van Maaren H, Walsh T. Handbook of Satisfiability. vol. 326. IOS press; 2021.

[12] Rossi F, Van Beek P, Walsh T. Handbook of constraint programming. Elsevier; 2006.

[13] Ohrimenko O, Stuckey PJ, Codish M. Propagation via lazy clause generation. Constraints An Int J. 2009;14(3):357–391.

[14] Kautz HA, Selman B. Planning as Satisfiability. In: ECAI; 1992. p. 359–363.

[15] van Beek P, Chen X. CPlan: A Constraint Programming Approach to Planning. In: Sixteenth National Conference on AI and Eleventh Conference on Innovative Applications of AI; 1999. p. 585–590.

[16] Miguel I, Jarvis P, Shen Q. Flexible graphplan. In: ECAI; 2000. p. 506–510.

[17] Rintanen J, Heljanko K, Niemelä I. Planning as Satisfiability: Parallel Plans and Algorithms for Plan Search. Artificial Intelligence. 2006;170(12-13):1031–1080.

[18] Fox M, Long D. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. Journal of Artificial Intelligence Research (JAIR). 2003;20:61–124.

[19] Geffner H. Functional STRIPS: a more flexible language for planning and problem solving. In: Logic-based artificial intelligence. Springer; 2000. p. 187–209.

[20] Long D, Fox M. The 3rd International Planning Competition: Results and Analysis. Journal of Artificial Intelligence Research (JAIR). 2003;20:1–59.

[21] Bofill M, Espasa J, Villaret M. The RANTANPLAN planner: system description. The Knowledge Engineering Review (KER). 2016;31(5):452–464.

[22] Audemard G, Simon L. Predicting Learnt Clauses Quality in Modern SAT Solvers. In: IJCAI; 2009. p. 399–404.

[23] Brummayer R, Biere A. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In: TACAS; 2009. p. 174–177.

[24] Nightingale P, Akgün Ö, Gent IP, Jefferson C, Miguel I. Automatically improving constraint models in Savile Row through associative-commutative common subexpression elimination. In: CP; 2014. p. 590–605.

[25] Nightingale P, Spracklen P, Miguel I. Automatically improving SAT encoding of constraint problems through common subexpression elimination in Savile Row. In: CP; 2015. p. 330–340.