Relaxing the Relaxed Exist-Step Parallel Planning Semantics

Tomáš Balyo
Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics, Charles University
Prague, Czech Republic
biotomas@gmail.com

***Abstract*—Solving planning problems via translation to satisfiability (SAT) is one of the most successful approaches to automated planning. We propose a new encoding scheme which encodes a planning problem represented in the SAS+ formalism using a relaxed Exist-Step semantics of parallel actions. The encoding by design allows more actions to be put inside one parallel step than other encodings and thus a planning problem can be solved with fewer SAT solver calls. The experiments confirm this property. In several non-trivial cases the entire plans fit inside only one parallel step. In our experiments we also compared our encoding with other state-of-the-art encodings such as SASE and Rintanen's Exist-Step encoding using standard IPC benchmark domains. Our encoding can outperform both these encodings in the number of solved problems within a given limit as well as in the number of SAT solver calls needed to find a plan.**

***Keywords*-planning; satisfiability; encoding; SAS; exist-step;**

I. Introduction

Planning is the problem of finding a sequence of actions – a plan, that transforms the world from some initial state to a goal state. The world is fully-observable, deterministic and static (only the agent we make the plan for changes the world). The number of the possible states of the world as well as the number of possible actions is finite, though possibly very large. We will assume that the actions are instantaneous (take a constant time) and therefore we only need to deal with their sequencing. Actions have preconditions, which specify in which states of the world they can be applied as well as effects, which dictate how the world will be changed after the action is executed.

One of the most successful approaches to planning is encoding the planning problem into a series of satisfiability (SAT) formulas and then using a SAT solver to solve them. The method was first introduced by Kautz and Selman [1] and is still very popular and competitive. This is partly due to the power of SAT solvers, which are getting more efficient year by year. Since then many new improvements have been made to the method, such as new compact and efficient encodings [2], [3], [4], better ways of scheduling the SAT solvers [3] or modifying the SAT solver's heuristics to be more suitable for solving planning problems [5].

In this paper we present a new encoding scheme which is particularly designed to be able to solve planning problems using a smaller number of SAT solver calls. In other words we are trying to pack more actions into one step of the solving process. This issue is also addressed by the $\exists$-Step planning semantics [3] and its relaxed version the Relaxed $\exists$-Step semantics [6]. Our encoding, which we call the Relaxed Relaxed $\exists$-Step (or $R^2\exists$-Step) semantics, is even more relaxed than these two and therefore our encoding can produce parallel plans with even smaller makespans. In several cases entire plans fit inside a single parallel step. We compare our new encoding with state-of-the-art encodings [2], [3] and show that for several International Planning Competition (IPC) [7] benchmark problem sets (called domains) we can solve significantly more problem instances.

II. Background

In this section we give the basic definitions of satisfiability, planning, and planning with parallel plans. We will also discuss the differences between the known parallel step semantics and our new semantics.

A. Satisfiability

A *Boolean variable* is a variable with two possible values *True* and *False*. A *literal* of a Boolean variable x is either x or $\neg x$ (*positive* or *negative literal*). A *clause* is a disjunction (OR) of literals. A clause with only one literal is called a *unit clause* and with two literals a *binary clause*. An implication of the form $x \Rightarrow (y_1 \vee \cdots \vee y_k)$ is equivalent to the clause $(\neg x \vee y_1 \vee \cdots \vee y_k)$. A *conjunctive normal form (*CNF*) formula* is a conjunction (AND) of clauses. A truth assignment ϕ of a formula F assigns a truth value to its variables. The assignment ϕ satisfies a positive (negative) literal if it assigns the value True (False) to its variable and ϕ satisfies a clause if it satisfies any of its literals. Finally, ϕ satisfies a CNF formula if it satisfies all of its clauses. A formula F is said to be satisfiable if there is a truth assignment ϕ that satisfies F. Such an assignment is called a *satisfying assignment*. The satisfiability problem (SAT) is to find a satisfying assignment of a given CNF formula or determine that it is unsatisfiable.

B. Planning

In the introduction we briefly described what planning is, in this section we give the formal definitions. We will use

the multivalued SAS+ formalism [8] instead of the classical STRIPS formalism [9] based on propositional logic.

A planning task Π in the SAS+ formalism is defined as a tuple $\Pi = \{X, O, s_I, s_G\}$ where

- $X = \{x_1, \ldots, x_n\}$ is a set of multivalued variables with finite domains $\mathrm{dom}(x_i)$.
- O is a set of actions (or operators). Each action $a \in O$ is a tuple $(\mathrm{pre}(a), \mathrm{eff}(a))$ where $\mathrm{pre}(a)$ is the set of preconditions of a and $\mathrm{eff}(a)$ is the set of effects of a. Both preconditions and effects are of the form $x_i = v$ where $v \in \mathrm{dom}(x_i)$.
- A state is a set of assignments to the state variables. Each state variable has exactly one value assigned from its respective domain. We denote by S the set of all states. $s_I \in S$ is the initial state. s_G is a partial assignment of the state variables (not all variables have assigned values) and a state $s \in S$ is a goal state if $s_G \subseteq s$.

An action a is *applicable* in the given state s if $\mathrm{pre}(a) \subseteq s$. By $s' = \mathrm{apply}(a, s)$ we denote the state after executing the action a in the state s, where a is applicable in s. All the assignments in s' are the same as in s except for the assignments in $\mathrm{eff}(a)$ which replace the corresponding (same variable) assignments in s.

A *sequential plan* P of *length* k for a given planning task Π is a sequence of actions $P = \{a_1, \ldots, a_k\}$ such that $s_G \subseteq \mathrm{apply}(a_k, \mathrm{apply}(a_{k-1} \ldots \mathrm{apply}(a_2, \mathrm{apply}(a_1, s_I)) \ldots))$.

C. Parallel Plans

A *parallel plan* P with *makespan* k for a given planning task Π is a sequence of sets of actions (called parallel steps) $P = \{A_1, \ldots, A_k\}$ such that $\mathrm{ord}(A_1) \oplus \cdots \oplus \mathrm{ord}(A_k)$ is a sequential plan for Π, where $\mathrm{ord}(A_i)$ is an ordering function, which transforms the set A_i into a sequence and $\oplus$ denotes the concatenation of sequences.

Let us denote by s_j the world state in between the parallel steps A_j and A_{j+1}, which is obtained by applying the sequence $\mathrm{ord}(A_j)$ on s_{j-1} (except for $s_0 = s_I$). Various parallel plan semantics define what actions can be together inside a parallel step. We list them in the order of decreasing strictness.

- The $\forall$-Step semantics [10] requires that each action $a \in A_j$ is applicable in the state s_j, the effects of all actions are applied in s_{j+1} and no two actions $a, a' \in A_j$ are interfering, e.g., a does not destroy any precondition of a' and vice versa. All orderings of such sets make valid sequential plans, hence the name $\forall$-Step semantics. The actions are independent so they can be executed in parallel (at the same time), hence the name parallel plans and parallel steps.
- The $\exists$-Step semantics [3] weakens the requirement on the independence of the actions. It is sufficient if there exists at least one ordering of the actions in each parallel step to construct a valid sequential plan. The actions cannot be executed in parallel anymore but the terms parallel plan and parallel step are used anyway.
- The Relaxed $\exists$-Step semantics [6] removes the requirement that each action in A_j has to be applicable in s_j. The effects of all actions are still applied.
- The Relaxed Relaxed $\exists$-Step ($R^2\exists$-Step) semantics, which we are introducing in this paper, removes the requirement of applied action effects. Therefore the only thing we require is that the actions in a parallel step can be ordered to form a valid sequential plan.

Let us demonstrate the differences of the semantics using the following simple example.

Example 1. *A truck is moving between 3 locations L_1, L_2, and L_3 starting at L_1. Two packages are located at L_1 and L_2, which can be picked up by the truck. The goal is that the truck is at L_3 having both packages. We model the problem using three state variables: the location of the truck x^T, $\mathrm{dom}(x^T) = (L_1, L_2, L_3)$, and the location of the packages x^{P1} and x^{P2}, $\mathrm{dom}(x^{P1}) = (L_1, T)$, $\mathrm{dom}(x^{P2}) = (L_2, T)$. We have two kinds of actions – move(mv) and pickUp(pu).*

- $\mathrm{mv}(L_1, L_2) = (\{x^T = L_1\}, \{x^T = L_2\})$
- $\mathrm{mv}(L_2, L_3) = (\{x^T = L_2\}, \{x^T = L_3\})$
- $\mathrm{pu}(P1, L_1) = (\{x^T = L_1, x^{P1} = L_1\}, \{x^{P1} = T\})$
- $\mathrm{pu}(P2, L_2) = (\{x^T = L_2, x^{P2} = L_2\}, \{x^{P2} = T\})$

The $R^2\exists$-Step semantics allows the single step plan $P = \{\{\mathrm{pu}(P1, L_1), \mathrm{mv}(L_1, L_2), \mathrm{pu}(P2, L_2), \mathrm{mv}(L_2, L_3)\}\}$. The Relaxed $\exists$-Step semantics forbids having both move operations inside one step, since the effect of the first move action is not applied after the step. Therefore the shortest possible Relaxed $\exists$-Step plan is $P = \{\{\mathrm{pu}(P1, L_1), \mathrm{mv}(L_1, L_2), \mathrm{pu}(P2, L_2)\}, \{\mathrm{mv}(L_2, L_3)\}\}$ with a makepan 2. The $\exists$-Step semantics does not allow the second pickUp action to be inside the first step, since its precondition is not satisfied in the initial state. Nevertheless the following 2 step parallel plan is still possible $P = \{\{\mathrm{pu}(P1, L_1), \mathrm{mv}(L_1, L_2)\}, \{\mathrm{pu}(P2, L_2), \mathrm{mv}(L_2, L_3)\}\}$ As of the $\forall$-Step semantics, all 4 actions must be in separate parallel steps.

III. Methodology

The basic idea of solving planning as SAT is the following [1]. We construct (by encoding the planning task) a series of SAT formulas $F_1, F_2, \ldots$ such that F_i is satisfiable if there is a parallel plan of makespan i. Then we solve them one by one starting from F_1 until we reach the first satisfiable formula F_k. From the satisfying assignment of F_k we can extract a plan of makespan k.

This basic idea can be improved in several ways. One of them is improved solver scheduling, i.e., better scheduling the tasks of solving the F_is. We can for example proceed by more than one step or impose time limits on the solving of the formulas. As shown in [3] clever ways of solver

scheduling can significantly improve the performance of the planning algorithm at the cost of possibly longer makespan plans. Nevertheless, in our experiments we will use the basic one-by-one scheduling since we are interested only in comparing the properties of encodings, i.e., the construction of the formulas F_i.

IV. The $R^2\exists$-Step Encoding

In this section we present a detailed description of our $R^2\exists$-Step encoding of planning into SAT. Concretely we will construct a formula F_k such that if F_k is satisfiable and ϕ is its satisfying assignment, then we can construct a $R^2\exists$-Step parallel plan from ϕ for the task $\Pi = \{X, O, s_I, s_G\}$. In the encoding we will explicitly represent which actions are in which step of the parallel plan as well as the world states in between steps. The first such state will represent the initial state, the second will represent the world state after applying the actions of the first step (in a proper order) and so on.

A. Variables, Plan Extraction and Action Ranking

In our encoding we have three kinds of Boolean variables for each time step $t \in \{1 \dots k\}$.

- Action variables z_t^a for each action $a \in O$. We will require that $z_t^a = True \Leftrightarrow a \in A_t$, i.e., a is in the parallel plan in step t.
- Assignment variables $y_t^{x=v}$ for each SAS+ variable $x \in X$, and each $v \in \mathrm{dom}(x)$. We will require that $y_t^{x=v} = \mathrm{True} \Leftrightarrow s_t(x) = v$, i.e., in the state s_t at the beginning of time step t the variable x is assigned to the value v.
- Auxiliary variables $c_t^{i,j}$. More details about these variables will be provided later.

By checking the truth values of all the action variables in a satisfying assignment of the formula we are about to define, we can easily construct a parallel plan. To prove that such a plan is a valid $R^2\exists$-Step parallel plan we will also need to provide an ordering function for the action sets.

We define the ordering function based on action ranks (inspired by [3]). Each action $a \in O$ gets a unique integer rank. We will denote by $r(a)$ the rank of a. The idea of ranking is that actions with lower ranks can be before actions with higher ranks. The ordering function will order the actions in the parallel steps according to their ranks.

The ranking can be arbitrary, however some rankings are better than others, since they may allow parallel plans with smaller makespans. In our implementation we assign the action ranks using modified topological sorting on the enabling graph of the problem. The *enabling graph* G is a directed graph with vertices representing actions defined as $G = (O, \{a \to a' | a, a' \in O; \mathrm{eff}(a) \cap \mathrm{pre}(a') \neq \emptyset\})$. The enabling graph may contain cycles and therefore topological ordering is not defined. We break the cycles by removing edges and use the topological ordering of the reduced enabling graph to assign action ranks. This is implemented by modifying the depth-first-search topological sorting algorithm [11] to ignore cycles. Surely there are better ways of assigning ranks to the actions. Finding better orderings is a subject of future work.

The $\exists$-Step [3] and the Relaxed $\exists$-Step [6] encodings use similar graphs called disabling and disabling-enabling graphs.

B. Basic Clauses

Here we start defining the clauses that the formula F_k contains. First we encode the initial state by a series of unit clauses for time 0 (before the first action set A_1).

$$y_0^{x=v}; \forall (x = v) \in s_I \tag{1}$$

Similarly we encode the goal conditions for time k.

$$y_k^{x=v}; \forall (x = v) \in s_G \tag{2}$$

To ensure the consistency of the world states we add the following binary clauses which enforce, that each state variable has at most one value.

$$\begin{gathered}(\neg y_t^{x=v_i} \vee \neg y_t^{x=v_j}) \\ \forall x \in X, v_i \neq v_j, \{v_i, v_j\} \subseteq \mathrm{dom}(x), \forall t \in \{1 \dots k\}\end{gathered} \tag{3}$$

So far we have only used the assignment variables. Next we will add the clauses connecting the action variables to assignments. The standard way of connecting actions to their precondition assignments is to add clauses that require that actions imply their preconditions

$$\begin{gathered}(\neg z_t^a \vee y_t^{x=v}) \\ \forall a \in O, \forall x = v \in \mathrm{pre}(a), \forall t \in \{1 \dots k\}\end{gathered}$$

This is used in the $\forall$-Step and $\exists$-Step encodings [3]. In our (and also in the Relaxed-$\exists$-Step [6]) encoding we require that the preconditions of actions are either satisfied by the proper assignment or by some other action in the same time step. If we encoded this idea in a straightforward way it would cause trouble since the actions could support each other in a cyclic manner with disregard to the initial state of the time step. The solution is to use the action ranks ($r(a)$). Then we can encode our relaxed condition by clauses requiring that each precondition of an action is either satisfied by the assignment from the previous world state or by an action with a lower rank in the same time step. The action-precondition clauses will be the following.

$$\begin{gathered}(\neg z_t^a \vee y_t^{x=v} \bigvee \{z_{a',t} | r(a') < r(a), (x = v) \in \mathrm{eff}(a')\}) \\ \forall a \in O, \forall (x = v) \in \mathrm{pre}(a), \forall t \in \{1 \dots k\}\end{gathered} \tag{4}$$

The ranking solves the problem of cyclic supporting, however there still remains one problem. It may happen that an action is relying on another action (with a lower rank) which sets it precondition but there may be another action between them which destroys the precondition. Dealing efficiently

with this problem is a bit more complicated and we will address it separately in the next subsection.

Next we describe the clauses that distinguish our $R^2\exists$-Step encoding from the Relaxed $\exists$-Step encoding and ensure that the effects of the actions are properly propagated. The idea is similar as in the case of preconditions. We will add clauses that ensure, that each effect $x = v$ of an action is either projected to the next world state or there is an action with a higher rank that will take the responsibility of setting the value of the variable x in the next world state (or find another substitute action).

$$\begin{aligned}&(\neg z_t^a \vee y_{t+1}^{x=v} \bigvee \{z_t^{a'} | r(a') > r(a), x \in \mathrm{scope}(a')\}) \\ &\forall a \in O, \forall (x = v) \in \mathrm{effprev}(a), \forall t \in \{1 \dots k-1\}\end{aligned} \tag{5}$$

By $\mathrm{effprev}(a)$ we mean the set of effects and prevailing assignments of a. *Prevailing assignments* are such precondition assignments that have no matching effect assignment, e.g., an assignment to the same variable. For example the action $(\{x_1 = 1, x_2 = 1\}, \{x_2 = 3\})$ has a prevailing assignment $x_1 = 1$. By $\mathrm{scope}(a)$ we mean the set of SAS+ variables that appear in $\mathrm{effprev}(a)$.

The next class of clauses will ensure consistency between two adjacent world states. The clauses will enforce that if an assignment holds in the state s_{t+1} then either the assignment was already valid in the state s_t or there was an action which had it in its effects.

$$\begin{aligned}&(\neg y_{t+1}^{x_i=v} \vee y_t^{x_i=v} \bigvee \{z_t^a | x_i = v \in \mathrm{eff}(a)\}) \\ &\forall x_i \in X, \forall v \in \mathrm{dom}(x_i), \forall t \in \{1 \dots k-1\}\end{aligned} \tag{6}$$

In other words the assignments cannot change between states without an action changing them.

C. Action Interference Clauses

The last and the most complicated set of clauses we need to add to our formula are the clauses solving the above mentioned problem of lower ranking actions destroying the preconditions of higher ranking actions in the same parallel step. We start by some notation regarding the relationships between actions and assignments that will simplify the following descriptions.

The relationship status of an action a and an assignment $x = v$ is one or more of the following

- a is *supporting* $x = v$ if $x = v \in \mathrm{eff}(a)$
- a is *opposing* $x = v$ if $x = v' \in \mathrm{eff}(a); v' \neq v$
- a is *requiring* $x = v$ if $x = v \in \mathrm{pre}(a)$
- a is *unrelated* to $x = v$ if $x = v \notin \mathrm{pre}(a) \cup \mathrm{eff}(a)$

Note that an action can be both requiring and opposing an assignment (that precondition would be non-prevailing).

We want to ensure that if an opposing action a_1 destroys an assignment then there is no action with a higher rank a_2 requiring it unless there is some supporting action a_3 between a_1 and a_2 that sets it up again. But then again there might be an action between between a_3 and a_2 that destroys the assignment again. The situation seems to be quite intricate but fortunately there is an elegant solution. The solution is inspired by the $\exists$-Step encoding [3], particularly their encoding of action interference constraints (also used in the Relaxed $\exists$-Step encoding [6]).

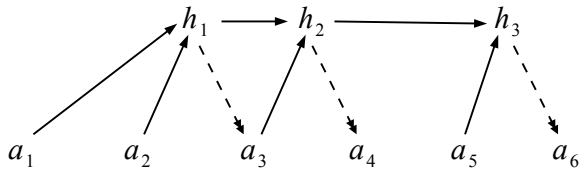

Figure 1. A graphic representation of the implications chain. In the example on the picture a_1, a_2, a_3, and a_5 are opponents of the given assignment and a_3, a_4, and a_6 require it. Notice that a_3 both requires and opposes the assignment. The solid lines represent the implications and the dashed lines negative implications.

In [3] and [6] the authors want to ensure that if an action destroys an assignment then no other action with a higher rank requiring that assignment can be in the same parallel step. Using auxiliary variables they build a chain of implications for every possible assignment over the opposing and requiring actions of that assignment. A formal description of such a chain for one assignment follows.

Let $a_1, a_2, \dots, a_k$ be a list of all n opposing and m requiring actions of a given assignment sorted by the ranks of the actions. Let $o_1, \dots o_n$ be the indices of the opposing actions and $r_1, \dots, r_m$ the indices of requiring actions ($k \leq m + n$). Let $\mathrm{next}(i) > i$ be the index of the closest requiring action after the i-th action in the sequence $a_1, a_2, \dots, a_k$. Let $h_{r_1}, h_{r_2}, \dots, h_{r_m}$ be auxiliary variables. The chain is composed of the following implications.

- $h_j \Rightarrow h_{j+1}; \forall j \in \{r_1, \dots, r_{m-1}\}$
- $h_j \Rightarrow \neg a_j; \forall j \in \{r_1, \dots, r_m\}$
- $a_j \Rightarrow h_{\mathrm{next}(j)}; \forall j \in \{o_1, \dots, o_n\}$

See Figure 1 for a graphic representation of an example. It is easy to see that such a chain of implications will ensure the required constraints. Whenever an opposing action gets selected into the plan it blocks all the actions with a higher rank that require the given assignment.

We will extend the chain definition to accommodate the situation where an intermediate action can restore the destroyed assignment. First we need to add the supporting actions into the chain and then relax the corresponding $h_j \Rightarrow h_{j+1}$ implications to allow the supporting actions to break the chain. A more formal description follows.

Let $a_1, a_2, \dots, a_k$ be a list of all n opposing, m requiring, and l supporting actions of a given assignment sorted by the ranks of the actions. Let $o_1, \dots o_n$ be the indices of the opposing actions, $r_1, \dots, r_m$ the indices of requiring actions, and $s_1, \dots, s_l$ the indices of supporting actions. Let $h_1, h_2, \dots, h_{k-1}$ be auxiliary variables. The chain is composed of the following implications.

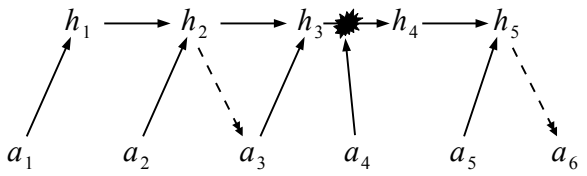

Figure 2. A graphic representation of the extended implications chain. In the example on the picture a_1, a_2, a_3, and a_5 are opponents of the given assignment, a_4 is a supporter and a_3, and a_6 require it. Being a supporter, a_4 can break the chain between h_3 and h_4. The solid lines represent the implications, the dashed lines negative implications and the explosion denotes the chain breaking.

- $h_{j-1} \Rightarrow h_j; \forall j \in \{2, \ldots, k-1\} \setminus \{s_1, \ldots, s_l\}$
- $h_{j-1} \Rightarrow \neg a_j; \forall j > 1; j \in \{r_1, \ldots, r_m\}$
- $a_j \Rightarrow h_j; \forall j \in \{o_1, \ldots, o_n\}$
- $h_{j-1} \Rightarrow (h_j \vee a_j); \forall j > 1; j \in \{s_1, \ldots, s_l\}$

An example with a graphic representation of the extended implication chain is given in Figure 2. The extended chain is very similar to the original one and it is easy to verify from the definition of the implications that a supporting action can indeed break the chain of implications which would prevent an action whose precondition has been restored. Next we give the clauses describing the extended chain.

$$\begin{aligned}
&\neg c_t^{x=v,\mathrm{prev}(a,N_{x=v})} \vee c_t^{x=v,a}; a \in (R_{x=v} \cup O_{x=v}) \\
&\neg c_t^{x=v,\mathrm{prev}(a,N_{x=v})} \vee \neg z_t^a; a \in R_{x=v} \\
&\neg z_t^a \vee c_t^{x=v,a}; a \in O_{x=v} \\
&\neg c_t^{x=v,\mathrm{prev}(a,N_{x=v})} \vee c_t^{x=v,a} \vee z_t^a; a \in S_{x=v} \\
&\forall x \in X, \forall v \in \mathrm{dom}(x), \forall t \in \{1 \ldots k\}
\end{aligned} \tag{7}$$

Where

- $c_t^{x=v,a}$ are new auxiliary Boolean variables
- $R_{x=v}$ is the set of actions that require $x = v$
- $O_{x=v}$ is the set of actions that oppose $x = v$
- $S_{x=v}$ is the set of actions that support $x = v$
- $N_{x=v}$ is the sorted list (by action ranks) of all supporting, opposing and requiring actions of $x = v$
- $\mathrm{prev}(a, N_{x=v})$ is the action preceding a in $N_{x=v}$

The encoding of the chain can be improved by removing some of the auxiliary variables and changing the related implications accordingly.

This concludes the description of our encoding. The final formula F_k is a conjunction of all the clauses defined in formulas (1),(2),(3),(4),(5),(6), and (7).

V. Experiments

To compare our new encoding to other state-of-the-art encodings we did experiments using eight International Planning Competition (IPC) [7] domains. We measured the time required to solve the instances, the number of SAT solver calls (makespans), and the number of problems solved within a given time limit. We also investigated the size and composition of the encoded formulas.

We compared our encoding with the state-of-the-art ∃-Step encoding [3] and the SASE encoding [2], which, similarly to our encoding, uses the SAS+ formalism as an input. We were unable to do experiments with the Relaxed ∃-Step encoding [6], (which is the most similar to our encoding) since we could not find any available implementation.

A. Experimental Setting

To compare the performance of the encodings we created a simple script, which iteratively constructed and solved the formulas for time steps $1, 2, \ldots$ until a satisfiable formula was reached. For each encoding we used the same SAT solver – Lingeling[12] (version ala).

The time limit was 30 minutes for each translation and SAT solving, e.g., the iterative construction and solving of formulas continued until either the solver or the translator took more than 30 minutes (or a satisfiable formula was reached and solved). Hence the overall planning time could exceed 30 minutes for a problem instance.

The experiments were run on a computer with Intel i7 920 CPU @ 2.67 GHz processor and 6 GB of memory. Our new R^2∃-Step encoding and the SASE encoding procedures were implemented in Java. To obtain the ∃-Step formulas we used Rintanen's planner Madagascar[13] (version 0.9999 10/05/2012 20:35:55 x86-32 1-core).

The following IPC domains were used (each contains 20 problems): Barman, Elevators, Parcprinter, Pegsol, Visitall, Storage, Woodworking, and Zenotravel. The benchmark problems are provided in the PDDL format which is accepted by Madagascar, however our encoding and SASE require input in the SAS+ format. We used Helmert's translation tool, which is a part of the Fast Downward planning system [14], to obtain the SAS+ files from the PDDL files. The translation is very fast requiring only a few seconds for entire domains.

B. Performance Results

Table I shows the number of problems (out of 20) that were solved withing the time limit. Our new encoding performed very well particularly in the Barman, Pegsol, and Visitall domains, where it could solve significantly more problems than SASE and ∃-Step. The only domain, where our encoding was dominated is the Zenotravel domain, but the difference is only one problem instance. As of the comparison of SASE and ∃-Step, ∃-Step solved more or equal number of problem instances in each domain.

Table I also contains information about the number of SAT solver calls. The column "calls" contains the average number of SAT solver calls while solving a problem instance in the given domain. If the problem was not solved, then the number of calls within the time limit is considered. Therefore the value "calls" denotes a lower bound on the

Table III
THIS TABLE CONTAINS THE AVERAGE NUMBER OF VARIABLES, CLAUSES, LITERALS, AND THE PERCENTAGE OF BINARY CLAUSES IN THE FORMULAS GENERATED FOR MAKESPAN=5 FOR ALL DOMAINS AND ENCODINGS.

Domain	SASE				∃-Step				R^2∃-Step			
	vars	clauses	literals	binary	vars	clauses	literals	binary	vars	clauses	literals	binary
barman	8 211	168 951	355 161	97%	5 853	35 862	83 266	96%	45 200	104 781	2 433 815	57%
elevators	6 592	79 451	173 920	95%	3 617	30 758	71 832	96%	64 059	139 819	907 931	83%
parcprinter	6 882	40 628	87 426	92%	2 690	23 681	51 115	93%	35 817	75 344	248 721	84%
pegsol	3 558	100 914	214 716	97%	1 476	12 312	28 378	93%	44 322	95 075	317 140	89%
storage	10 517	429 702	908 835	98%	5 190	46 072	112 891	96%	167 154	351 021	5 335 298	85%
visitall	4 203	255 634	517 948	99%	1 480	14 753	31 989	93%	88 605	186 991	506 840	97%
woodwork.	9 076	63 022	147 196	95%	3 815	27 213	67 310	95%	73 473	149 890	1 846 921	67%
zenotravel	55 380	1 387 458	2 963 258	99%	36 552	207 313	539 024	98%	169 740	358 900	11 467 913	83%

Table I
THE NUMBER OF PROBLEMS IN EACH DOMAIN THAT THE ENCODING SOLVED WITHIN THE GIVEN TIME LIMIT (30 MINUTES PER TRANSLATION/SOLVING) AND THE AVERAGE NUMBER OF SAT SOLVER CALLS. EACH DOMAIN CONTAINS 20 PROBLEMS.

Domain	SASE		∃-Step		R^2∃-Step	
	solved	calls	solved	calls	solved	calls
barman	8	46.3	8	36.6	14	15.8
elevators	20	9.5	20	6.5	20	5.3
parcprinter	20	13.5	20	13.5	20	2.5
pegsol	7	22.8	13	24.0	19	9.6
storage	15	9.2	19	7.9	19	5.3
visitall	9	27.0	11	31.35	20	2.7
woodwork.	20	3.4	20	3.3	20	2.7
zenotravel	16	5.9	16	4.45	15	3.7

Table II
TIME IN SECONDS REQUIRED TO SOLVE ALL THE PROBLEMS WHICH WERE SUCCESSFULLY SOLVED, AND THE TOTAL TIME SPENT SOLVING THE DOMAINS. THE DISPLAYED TIMES ARE SUMS OF TRANSLATION AND SAT SOLVING TIMES.

Domain	SASE		∃-Step		R^2∃-Step	
	solved	total	solved	total	solved	total
barman	14256	87729	7268	64400	21089	52795
elevators	586	586	6	6	446	446
parcprinter	512	512	8	8	63	63
pegsol	4563	58162	7558	32158	832	3466
storage	829	15357	1408	3562	1453	4244
visitall	574	64323	2426	52241	85	85
woodwork.	181	181	5	5	164	164
zenotravel	633	10212	79	8296	3709	11818

average makespan of the plans that could be found. In case that all the problems in a domain were solved, then it is exactly the average makespan of found plans. Note, that the R^2∃-Step encoding requires to solve $k+1$ formulas to find a parallel plan of makespan k. Therefore the average makespan of the plans in this case is one less than the number of calls presented in the Table I.

From the data in Table I it is apparent, that our new encoding requires the smallest number of SAT solver calls and produces parallel plans with the smallest makespans for all domains. The second is the ∃-Step encoding and the third is SASE. The results are not unexpected, since SASE represents a ∀-Step semantics encoding. The gap between the number of calls is particularly big for the domains Parcprinter and Visitall. The makespans of plans found using our encoding were 1 for 10 instances of Parcprinter and 6 instances of Visitall and 2 for the rest of the instances (10 and 14 respectively). For example the problem "problem06-full.pddl" of the Visitall domain was solved (in 2 seconds) with a single parallel step and the ∃-Step encoding failed to solve that problem (with a total time 4662 seconds) at makespan 34. In these two domains our relaxation has a particularly strong effect.

Table II shows the times required to solve the problems. The columns labeled "solved" contain the total time spent solving those instances, which were successfully solved. This includes the time required for both formula construction (encoding) and solving. The columns "total" contain the total time spent solving all instances in the given domain.

From the data we can see that on the "easy" domains (Elevators, Parcprinter, and Woodworking) the ∃-Step encoding is very fast. The reason for this might be the fact, that the ∃-Step translator is highly optimized and implemented in C++, while our encoding and SASE are implemented in Java and also the formulas generated by the ∃-Step translator are much smaller (see the next subsection for more details). On the other hand, for the harder domains the time differences are not that favorable for the ∃-Step encoding. The time difference is very significant in the Pegsol and Visitall domains, where our encoding is several times faster than the ∃-Step or SASE encoding. The worst results are obtained by the SASE encoding, which is dominated by each opponent in each domain except from the Zenotravel domain, where SASE is faster than the R^2∃-Step encoding.

C. Formula Sizes

We also investigated the formulas that are generated by the encodings. We measured the number of variables, clauses, and the total number of literals (sum of lengths of all clauses). We also computed the percentage of binary clauses (implications) in the formulas. The formulas were generated for makespan=5 and we averaged the results of the instances in each domain. Table III contains the measured data.

The $\exists$-Step formulas are the smallest, followed by the SASE formulas and $R^2\exists$-Step formulas, which are much bigger than the previous two. The first two also have a consistently high percentage of binary clauses, while the $R^2\exists$-Step has between 57% and 97%. It is interesting that the percentage is the lowest for the Barman domain and highest for the Visitall domain, since these two domains are the ones, where $R^2\exists$-Step significantly outperformed SASE and $\exists$-Step (see Table I). This suggest a probably surprising conclusion, that the percentage of binary clauses in the formula is not very significant for the performance of the planning algorithm.

Also the bigger size of the $R^2\exists$-Step formulas does not seem to cause performance problems. The first explanation that comes to mind is that a planner using this encoding does not need to solve formulas for large makespans, which is true (see Table I). But on the other hand if we look at the Pegsol and Barman domains, the average number of solver calls is high for those domains, but the results are still much better than for the other two encodings.

VI. Conclusion

In this paper we have introduced a new way of encoding planning into satisfiability called the $R^2\exists$-Step encoding. It is designed to allow much more actions to be packed inside one parallel step than previous encodings and therefore a SAT based planning algorithm using this encoding requires much less SAT solver calls in order to find a plan.

The encoding is build upon the multivalued SAS+ formalism[8], which is an alternative to the STRIPS planning formalism[9]. We compared our encoding using 8 standard IPC benchmark sets against the state-of-the-art $\exists$-Step encoding[3] and the SASE encoding[2], which is also based on the SAS+ formalism. The experiments showed that our new encoding scheme, although produces much bigger SAT formulas, is very competitive and can solve significantly more benchmark problems than the other two encodings. The experiments also confirmed that much less calls of a SAT solver are required to solve the planning problems.

A. Future Work

As of future work we believe that the encoding can be further improved by reducing the size of the formulas. Both the number of variables and clauses could be potentially reduced by deeper analysis of the planning problem instances and refining the encoding, particularly the encoding of the implication chain.

Also the ranking of actions could be possibly improved by a more advanced analysis of the graph of action relations.

Acknowledgment

The research is supported by the Czech Science Foundation under the contract P103/10/1287 and by the Grant Agency of Charles University under contracts no. 266111 and 600112. This research was also supported by the SVV project number 267 314.

References

[1] H. A. Kautz and B. Selman, "Planning as satisfiability," in *ECAI*, 1992, pp. 359–363.

[2] R. Huang, Y. Chen, and W. Zhang, "A novel transition based encoding scheme for planning as satisfiability," in *AAAI*, M. Fox and D. Poole, Eds. AAAI Press, 2010.

[3] J. Rintanen, K. Heljanko, and I. Niemelä, "Planning as satisfiability: parallel plans and algorithms for plan search," *Artif. Intell.*, vol. 170, no. 12-13, pp. 1031–1080, 2006.

[4] N. Robinson, C. Gretton, D. N. Pham, and A. Sattar, "Sat-based parallel planning using a split representation of actions," in *ICAPS*, A. Gerevini, A. E. Howe, A. Cesta, and I. Refanidis, Eds. AAAI, 2009.

[5] J. Rintanen, "Planning as satisfiability: Heuristics," *Artif. Intell.*, vol. 193, pp. 45–86, 2012.

[6] M. Wehrle and J. Rintanen, "Planning as satisfiability with relaxed exist-step plans," in *Australian Conference on Artificial Intelligence*, ser. Lecture Notes in Computer Science, M. A. Orgun and J. Thornton, Eds., vol. 4830. Springer, 2007, pp. 244–253.

[7] T. Stegun and S. Fratini, "The fourth international competition on knowledge engineering for planning and scheduling, 2012," http://ipc.icaps-conference.org/, July 2013.

[8] C. Bäckström and B. Nebel, "Complexity results for sas+ planning," *Computational Intelligence*, vol. 11, pp. 625–656, 1995.

[9] R. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artif. Intell.*, vol. 2, no. 3/4, pp. 189–208, 1971.

[10] A. Blum and M. L. Furst, "Fast planning through planning graph analysis," *Artificial Intelligence*, vol. 90, no. 1-2, pp. 281–300, 1997.

[11] R. E. Tarjan, "Edge-disjoint spanning trees and depth-first search," *Acta Inf.*, vol. 6, pp. 171–185, 1976.

[12] A. Biere, "Lingeling and plingeling home page," http://fmv.jku.at/lingeling/, July 2013.

[13] J. Rintanen, "Planning as satisfiability: state of the art," http://users.cecs.anu.edu.au/~jussi/satplan.html, July 2013.

[14] M. Helmert, "The fast downward planning system," *Journal of Artificial Intelligence Research (JAIR)*, vol. 26, pp. 191–246, 2006.