Compilation-Based Approaches to Parallel Planning: An Empirical Comparison

Kristýna Pantůčková and **Roman Barták**
Charles University, Faculty of Mathematics and Physics
Prague, Czech Republic
{pantuckova, bartak}@ktiml.mff.cuni.cz

Abstract

Automated planning deals with finding a sequence of actions, a plan, to reach a goal. One of the possible approaches to automated planning is a compilation of a planning problem to a Boolean satisfiability problem or to a constraint satisfaction problem, which takes direct advantage of the advancements of satisfiability and constraint satisfaction solvers. This paper provides a comparison of three encodings proposed for the compilation of planning problems: *Transition constraints for parallel planning* (TCPP), *Relaxed relaxed ∃-Step* encoding and *Reinforced Encoding*. We implemented the encodings using the programming language Picat 2.8, we suggested certain modifications, and we compared the performance of the encodings on benchmarks from international planning competitions.

Introduction

Planning is an important aspect of deliberate reasoning of autonomous artificial agents [Ghallab, Nau, and Traverso, 2004]. The aim of planning is to find a sequence of actions (called a plan) to reach a desired state of the world. This paper focuses on solving planning tasks by compiling to a constraint satisfaction problem (CSP) or a Boolean satisfiability problem (SAT). The primary motivation for using compilation (also called reduction) techniques to problem-solving is leveraging SAT and CSP solvers advancements, including new reasoning algorithms and heuristics. When solving a planning task, the number of actions needed to reach the goal is unknown, while the SAT formula or a constraint satisfaction problem is of a fixed size. This disproportion is traditionally approached by solving a fixed-length planning problem [Kautz and Selman, 1999]. The SAT formula or a CSP encodes the problem of the existence of a plan of a given (maximum) length. If the underlying solver proves that no solution exists, i.e., no plan of a given (maximum) length exists, the length bound is increased, and the process is repeated until the plan is found. To decrease the number of calls to the solver, parallel planning is used. It allows more actions to run in parallel and hence finding plans with a smaller makespan (though possibly with more actions). We focus on finding parallel plans in this paper.

Compilation to a CSP was firstly proposed to compile a planning graph [Do and Kambhampati, 2000], later to encode planning problems directly [Lopez and Bacchus, 2003]. Barták [2011a] proposed the model PaP, where value transitions are represented by domain transition graphs and encoded by table constraints. PaP uses the ∀-step parallel semantics, where each parallel step contains more actions that are executable in any order. Afterward, PaP was reformulated as PaP-2, which uses only action variables [Barták, 2011b]. PaP-2 inspired Ghooshchi et al. [2017], who proposed the encoding Transition constraints for parallel planning (TCPP), which, in contrast, uses only state variables.

Compilation to SAT was proposed as an alternative to deduction [Kautz and Selman, 1992]. Rintanen, Heljanko, and Niemelä [2006] proposed the ∃-step parallel semantics, which allows more actions in each parallel step than the ∀-step semantics. The ∃-step semantics was further relaxed by Wehrle and Rintanen [2007] and Balyo [2013].

Our work focuses on implementation, modification and comparison of three encodings for parallel planning: TCPP [Ghooshchi et al., 2017], $R^2\exists$-Step encoding [Balyo, 2013] and Reinforced Encoding [Balyo, Barták, and Trunda, 2015]. TCPP is one of the most recent CSP encodings, and $R^2\exists$-Step encoding [Balyo, 2013] and Reinforced Encoding [Balyo, Barták, and Trunda, 2015] are among the most recent SAT-based approaches which focus purely on the encoding of problems as newer approaches use other optimization techniques such as incremental SAT solving, which is not studied in this paper. $R^2\exists$-Step encoding uses a different parallel semantics than TCPP, while Reinforced Encoding uses a similar definition of parallel step while describing the model differently than TCPP. TCPP has been compared with other SAT or CSP-based planners and with planners based on state-space search. However, to our knowledge, TCPP has not been compared with the $R^2\exists$-Step encoding or Reinforced Encoding yet. We implemented these encodings using the programming language Picat 2.8 (www.picat-lang.org), and we compare their performance on standard benchmarks from International Planning Competitions (IPC). Additionally, we propose modifications of these encodings to improve the efficiency of our implementation.

The contributions of this paper are: (1) a comparison of TCPP, $R^2\exists$-Step encoding and Reinforced Encoding, all implemented in Picat 2.8, on standard planning benchmarks

from International Planning Competitions and (2) modifications of these encodings, which are more suitable for implementation in Picat 2.8 than the original formulations.

Background

Classical (STRIPS) planning

A **classical planning problem** can be defined as a 4-tuple $P = (V, A, s_0, g)$, where V is a set of multi-valued state variables $\langle V_1, \ldots, V_n \rangle$ with finite domains ($Domain(V_i)$ is the set of values that can be assigned to the variable V_i), A is a set of actions, s_0 is the initial state of the world, and g is the goal.

Each state is described by values of state variables $\langle v_1, \ldots, v_n \rangle$ where $\forall i : 1 \leq i \leq n : v_i \in Domain(V_i)$. For example, these variables describe the locations of objects in the world. The **goal** is characterized by assignments of goal values to a subset of state variables. For example, the goal may specify desired locations of certain objects.

An **action** is defined by its preconditions and effects describing values of state variables before and after execution of the action. Sets of value assignments represent preconditions and effects. While the precondition specifies the values of certain state variables required before executing the action, the effect specifies how certain state variables change their values after the execution of the action. The other variables (not among the effects) keep their values from the previous state (the frame axiom).

A solution to a classical planning problem is a **sequential plan**, a totally ordered finite sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$, such that the first action a_1 can be executed in the initial state s_0, a_i can be executed in the state s_{i-1}, where s_i is the resulting state after execution of the action a_i ($\forall i : 2 \leq i \leq n$), and the state s_n satisfies the goal condition g.

Parallel planning

In parallel planning, it is allowed to have more actions in one time step. A sequential plan can then be obtained from the parallel plan by ordering actions in each parallel step.

The **$\forall$-step semantics** of parallel plans requires that each ordering of actions must be valid in each parallel step. Assume s to be the state before the parallel step and s' be the state after the parallel step. Then, any permutation of actions from the parallel step will transfer the state s to state s'.

Consider a planning problem with actions a_1, a_2 and a_3 where:

- $\text{preconditions}(a_1) = \{v_1 = 1\}$,
- $\text{preconditions}(a_2) = \{v_1 = 1\}$,
- $\text{preconditions}(a_3) = \{v_1 = 1,\ v_2 = 2,\ v_3 = 2\}$,
- $\text{effects}(a_1) = \{v_2 = 2\}$,
- $\text{effects}(a_2) = \{v_3 = 2\}$,
- $\text{effects}(a_3) = \{v_1 = 2\}$,
- $s_0 = \{v_1 = 1,\ v_2 = 1,\ v_3 = 1\}$,
- $g = \{v_1 = 2\}$.

Then, from the parallel plan $\langle \{a_1, a_2\}, \{a_3\} \rangle$ with two parallel steps, we obtain two sequential plans of length three, $\langle a_1, a_2, a_3 \rangle$ and $\langle a_2, a_1, a_3 \rangle$.

Formally, a **parallel plan** for a planning problem P respecting the $\forall$-step semantics [Ghooshchi et al., 2017] is a sequence of sets of actions $\Pi = \langle A_1, \ldots, A_m \rangle$ such that for each $t : 1 \leq t \leq m$ there exists a state s_t such that for each permutation A'_t of the action set A_t, s_t is the state after executing actions in the order given by the permutation A'_t in the state s_{t-1}, where s_0 is the initial state and s_m satisfies the goal condition g. We call the length m of a parallel plan a **makespan**. Some planners use the **$\exists$-step semantics**, where at least one valid action ordering must exist for each parallel step. A parallel plan is also called a **layered plan**, where each action set A_t is called a layer [Barták, 2011a]. While sequential planners search for a plan with the minimal sequential plan length, parallel planners search for a plan with the minimal makespan. As a result, parallel plans often contain redundant actions, which are not necessary for reaching the goal [Ghooshchi et al., 2017].

Encoding of a planning problem

To translate a planning task to SAT or CSP, we encode the problem with a fixed plan length since the problem must be described by a fixed number of variables and constraints. There will be one copy of each variable and each constraint for each step of the plan. If the solver proves that a plan of this length does not exist, we encode the problem again with an increased plan length until the solver finds a solution [Kautz and Selman, 1999].

Transition Constraints for Parallel Planning

The encoding Transition constraints for parallel planning (TCPP) was proposed by Ghooshchi et al. [2017] as a constraint satisfaction problem. TCPP models domains of state variables by **domain transition graphs** (DTG) of variables. Vertices of DTG represent values in its domain and edges represent value transitions. For each action a where $\{v = i\} \in \text{preconditions}(a)$ and $\{v = j\} \in \text{effects}(a)$, there is an edge $i \rightarrow j$ in DTG of the variable v. For each action a where $\{v = j\} \in \text{effects}(a)$ but the value of v is not defined in $\text{preconditions}(v)$, there is an edge $* \rightarrow j$ in DTG of v, where $*$ denotes the don't care vertex. Actions that do not change a value of v, i.e., actions that have v only in preconditions and actions that have v neither in preconditions nor in effects, are called **no-op actions**. No-op actions can be represented by loops on vertices of DTG. Figure 1 shows an example of DTG.

A parallel plan corresponds to synchronized transitions in DTGs of all variables. Any set of non-interfering actions can be used in one parallel step. Two actions are **non-interfering** if they can be executed in any order with the same result, i.e., effects of one action do not destroy preconditions and effects of the other action. Note that preconditions must be preserved during the execution of an action. Transitions in DTG are encoded by table constraints. Table constraints, which enumerate possible values of a tuple of variables, are provided by Picat [www.picat-lang.org, 2019] as well as by

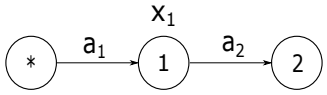

Figure 1: An example domain transition graph (DTG). Consider a variable x_1, whose values can be changed by two actions, a_1 and a_2, where $\text{preconditions}(a_1) = \{\}$, $\text{effects}(a_1) = \{x_1 = 1\}$, $\text{preconditions}(a_2) = \{x_1 = 1\}$ and $\text{effects}(a_2) = \{x_1 = 2\}$. Then this picture represents DTG of x_1. The vertex labelled by $*$ is the don't care vertex. No-op actions are omitted in the picture.

other CSP solvers. The model contains two types of variables for each time point, state variables and parallelism variables. For each state variable, there is one parallelism variable. Values of parallelism variables prevent a solver from selecting interfering actions in the same layer. For instance, when two actions a_1 and a_2, which change a value of v, are interfering, we can set the values of the parallelism variable p_v of the variable v such that the value of p_v, when a_1 is selected, is different from the value of p_v, when a_2 is selected.

Table 1: Example transition table of the model TCPP

$V_i[T]$	$V_i[T+1]$	$V_j[T]$	$V_j[T+1]$	$P_i[T]$	$P_j[T]$
1	2	3	4	1	3
*	2	*	4	0	3
1	1	*	*	0	*
2	2	*	*	0	*

Let us assume a variable V_i whose transition table for the transition between the time points T and $T+1$ is shown in table 1. The first two rows correspond to two actions that can modify the value of V_i. The other rows represent no-op actions. The symbol $*$ denotes undefined values (don't care values). The value of the parallelism variable P_i must be different in the first two rows. Otherwise, these two actions could be executed in one layer as values of the other variables are compatible.

Relaxing the Relaxed Exist-Step Parallel Planning Semantics

The $R^2\exists$-Step encoding proposed by Balyo [2013] as a SAT model implements the relaxed relaxed $\exists$-step parallel semantics, where actions from one layer must be executable in at least one order. The order of execution of actions is fixed and decided in advance. The authors propose to find the action ordering by sorting vertices in an enabling graph by modified topological sort, which ignores cycles in the graph. Vertices of an **enabling graph** represent actions. For each precondition $V_i = x$ of an action A_j, there is a directed edge $U_k \rightarrow U_j$ for each action A_k having $V_i = x$ in effects. The encoding uses Boolean action variables, state variables, and auxiliary variables, which are used in constraints enforcing the $R^2\exists$-Step parallel semantics. We implemented the encoding using multi-valued state variables.

Formulation in Picat

The original formulation containing a large number of variables and constraints turned out to be very inefficient for the Picat SAT solver. Therefore, we decided to shorten the description of a model by describing the evolution of a value of each variable during each parallel step by a constraint simulating a regular automaton, which is provided by Picat [www.picat-lang.org, 2019].

In the automaton for the variable V_i for the time step T, there is an initial state q_0, one accepting state q_f and $|Domain(V_i)|$ additional states $q_1, \ldots, q_{|Domain(V_i)|}$, where each state corresponds to one value from the domain of V_i. Edges can be labelled by actions. However, the size of the input alphabet can be reduced if we consider value transitions instead of actions. The input sequence of an automaton for time T starts with a letter representing $V_i[T]$, then there follows the sequence of symbols, which represent action variables $A_1[T], \ldots, A_k[T]$. The last symbol in the input sequence represents $V_i[T+1]$.

If an action variable $A_i[T]$ is equal to 0, which means that the action a_i it not executed during the parallel step T, the automaton reads the universal no-op symbol and stays in its current state. Otherwise, the automaton transits from the state representing the precondition value of the action to the state representing the effect value. If the value of the variable is defined only in preconditions, the automaton will stay in the state representing the precondition value. If the current state of the automaton is not compatible with preconditions, the automaton transits to the error state.

The input alphabet contains one symbol for each possible value transition of the variable (including no-op transitions and transitions from a don't care value) and a symbol representing the universal no-op transition. Figure 2 shows an example of such automaton.

The transition function is defined as follows:

- $\delta(q_0, input\_value_j) = q_j$ for each value j from the domain of V_i ($input\_value_j$ is the symbol identifying an input value and there is an edge from the initial state to each internal vertex to transfer the automaton to the state corresponding to the initial value of V_i)
- $\delta(q_j, t) = q_k$ for each transition identifier t such that $value_k$ is the goal value of the transition and either $value_j$ is the initial value or the initial value is the don't care value
- $\delta(q_j, t) = q_j$ for each transition identifier t, which represents the no-op transition on the value $value_j$; these transitions correspond to actions where V_i is in preconditions but not in effects
- $\delta(q_j, noop) = q_j$, where $noop$ is the identifier used for all transitions (actions) which are not executed (the universal no-op action)
- $\delta(q_j, output\_value_j) = q_f$ for each value j from the domain of V_i ($output\_value_j$ is the symbol identifying an output value and the automaton accepts only if the last input is equal to the last state, i.e., the computation must end in the final value of V_i)

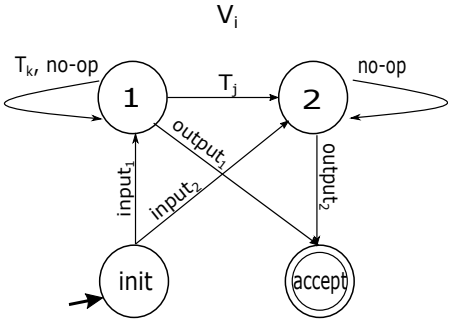

Figure 2: An example automaton describing evolution of values of the variable V_i during the time step T. States 1 and 2 represent values in the domain of V_i. The letter no-op represents the universal no-op action. Some actions, represented by the transition T_k, require $V_i = 1$ in preconditions. Other actions, represented by T_j, change the value of V_i from 1 to 2. The other edges correspond to input and output values of the variable.

- $\delta(q, x) = error\_state$ for each transition not defined above

Ranking

The efficiency of a model is affected by the ranking of actions. A better ordering of actions allows solving a planning problem in fewer parallel steps. We modified the ranking by adding disabling edges to describe relations between actions that are not yet connected in the enabling graph. For a pair of actions a_i, a_j where $v_k = v$ is an effect of a_i and $v_k = w$ is a precondition of a_j and $v \neq w$ (a_i destroys a precondition of a_j), disabling edge is an edge directed from the vertex representing a_j to the vertex representing a_i. By adding disabling edges to an enabling graph, we create an **enabling-disabling** graph. Additionally, when searching for the topological sorting in an enabling-disabling graph using the depth-first search, we always start in the vertex with the lowest indegree (number of incoming edges). According to our experiments, adding disabling edges improved the performance of the encoding.

Reinforced Encoding

Reinforced Encoding was proposed by Balyo, Barták, and Trunda [2015] for compilation to SAT. This model combines action and state variables with transition variables, representing how the value of a state variable changes between two layers. Constraints describe relations between the execution of actions, value transitions, and values of state variables. Our implementation uses multi-valued state variables.

Although both Reinforced Encoding and TCPP use the $\forall$-step parallel semantics, in Reinforced Encoding, two actions cannot be executed during the same time step if they are not independent. Two actions are **independent** if neither preconditions nor effects of the first action contain any variable occurring in preconditions or effects of the second action. By this definition, two actions cannot be executed together, for instance, if they have the same precondition. However, suppose this precondition variable does not occur in effects of any of these actions. In that case, these actions could be selected in one parallel step because they can be executed in any order.

We modified the encoding to use the same definition of parallel step as TCPP, i.e., to allow selection of actions that are not interfering. During each time step, at most one action that causes any transition $value_1 \rightarrow value_2$ of the variable v_i where $value_1 \neq value_2$ can be executed. We used the Picat predicate $sum/1$, where the sum of the involved action variables must not exceed 1 to ensure that at most one of the actions is executed.

Let us assume a variable V_i, which is in both preconditions and effects of actions a_j, a_k and a_l. We can describe the conflict of these actions during the time step T with the predicate sum as follows: $sum(\langle A_j[T], A_k[T], A_l[T]\rangle) \leq 1$.

Experimental evaluation

To translate planning instances from International Planning Competitions from PDDL [Ghallab et al., 1998] to the SAS+ formalism [Bäckström and Nebel, 1995], which uses multi-valued state variables, we used the translation component of the Fast Downward planning system [Helmert, 2006]. We also extended all encodings by mutex constraints produced by the translator component of Fast Downward, as suggested by Ghooshchi et al. [2017]. These constraints enumerate mutually exclusive values of pairs of state variables. We encoded mutex constraints by negative table constraints provided by Picat.

We performed experiments to compare our modifications with the original versions and to compare different encodings. All experiments were run on a computer with the Intel Core i7-8550U CPU @ 1.80GHz processor and 16 GB of RAM. For solving planning problems, we used the SAT solver provided by Picat as it outperformed other solvers available in Picat. In most experiments, we measure, for each domain, the total number of instances solved within the time limit of 60 minutes per instance. Besides the solving time, the total time also includes the translation time and the time used for all processing steps. We measure the total time from translation from PDDL to finding a plan. For evaluation of encoding modifications, we used planning problems from the domains *airport*, *grid*, *miconic*, *pegsol*, *storage*, and *zenotravel*. For comparison of different models, we used in addition the domains *tpp*, *parcprinter*, and *woodworking*.

We have chosen the domain *miconic* since it contains a wide range of problems from simple problems that were solved with all encodings, including the most literal implementations, to difficult problems that were not solved with any of the implemented models. The domain *grid* contains difficult problems where the number of solved problems was low with all encodings and with some encodings, we could not solve any problem. The other domains were selected from the domains used for experimental evaluation in the three original papers. The domains *storage* and *zenotravel* were used by Ghooshchi et al. [2017] and

Balyo [2013]. The domains *pegsol*, *parcprinter* and *woodworking* were used by Balyo [2013] and Balyo, Barták, and Trunda [2015]. The domains *tpp* and *airport* were used by Ghooshchi et al. [2017].

$R^2\exists$-Step encoding

Table 2 shows the number of problems solved with $R^2\exists$-Step encoding within one hour. The table demonstrates the necessity of modification of the encoding as we could solve only the simplest problems with the original encoding in Picat (the column labeled as *orig*). With regular automata, the total number of solved problems is higher with disabling edges (*disab*) than with the original enabling graph (*enab*) even though the results slightly worsened in some domains. Addition of mutexes produced by the Fast Downward translator slightly improved the performance (*mut*). In the rows where the translator did not produce any mutexes, the numbers are in parentheses.

Table 2: The number of solved problems for modifications of $R^2\exists$-Step encoding

domain	total	orig	enab	disab	mut
airport	50	0	**15**	14	**15**
grid	5	0	**1**	0	0
miconic	150	15	48	**58**	(**58**)
pegsol	20	0	**19**	18	(18)
storage	30	1	16	**17**	**17**
zenotravel	20	0	**15**	**15**	(**15**)
total	275	16	114	122	**123**

Reinforced Encoding

Table 3 compares results of modifications of Reinforced Encoding. The results improved significantly after changing the definition of parallel step to work with non-interfering actions (*non-interfering*) instead of independent actions (*orig*). Mutexes of the Fast Downward translator (*mut*) slightly improved the performance.

Table 3: The number of solved problems for modifications of Reinforced Encoding

domain	total	orig	non-interfering	mut
airport	50	11	**20**	**20**
grid	5	0	0	**1**
miconic	150	40	**53**	(**53**)
pegsol	20	**14**	**14**	(**14**)
storage	30	12	**16**	**16**
zenotravel	20	7	**16**	(**16**)
total	275	84	119	**120**

Comparison of encodings

Table 4 contains the number of problems from nine domains solved within the limit of one hour for TCPP, $R^2\exists$-Step encoding and Reinforced Encoding (RE). Figure 3 shows how the number of solved problems increases with increasing time limit. Based on the aggregate results, TCPP solved the highest total number of problems, the $R^2\exists$-Step encoding was in the second place and the Reinforced Encoding was in the last place.

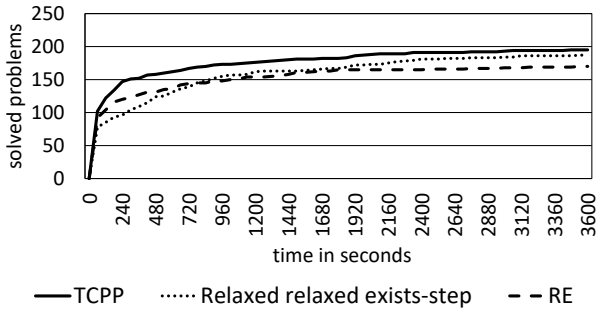

Figure 3: This graph shows the number of problems solved with TCPP, $R^2\exists$-Step encoding and Reinforced Encoding in time given in seconds.

Table 4: Comparison of the number of solved problems for TCPP, $R^2\exists$-Step encoding and Reinforced Encoding

domain	total	TCPP	$R^2\exists$-Step	RE
airport	50	**23**	15	20
grid	5	**1**	0	**1**
miconic	150	**58**	**58**	53
parcprinter	20	**20**	**20**	**20**
pegsol	20	15	**18**	14
storage	30	16	**17**	16
tpp	30	**28**	24	20
woodworking	20	18	**20**	10
zenotravel	20	**16**	15	**16**
total	345	**195**	187	170

Each encoding attempts to achieve a better performance in a different way. The advantage of TCPP is that this encoding requires the smallest number of variables and constraints. For instance, for problems in the domain *zenotravel*, TCPP adds on average about 33 variables for each layer, while our final versions of both the $R^2\exists$-Step encoding and Reinforced Encoding add on average more than 7000 variables. However, as we used a SAT solver, Picat internally translated the models to SAT; as a result, variables and constraints do not correspond to those used by the solver. Nevertheless, the results suggest that even in that case, the formulation of models affects the performance significantly. Picat can translate some constraints more efficiently than the others, and a high number of variables and constraints in the initial model seems to decrease the performance of the final SAT model.

On the other hand, the R$^2\exists$-Step encoding requires the lowest number of SAT solver calls due to the shortest makespan of parallel plans. For instance, in the domain *pegsol*, where this encoding outperformed both TCPP and Reinforced Encoding, the parallel plans produced by these two encodings were more than twice as long as the plans of the R$^2\exists$-Step encoding (the average makespan was 9 for R$^2\exists$-Step encoding and 24 for TCPP or Reinforced Encoding).

TCPP and the final version of Reinforced Encoding describe the same model. While Reinforced Encoding defines a model using a high number of constraints that describe relations between state, action, and transition variables, TCPP requires only state and parallelism variables, whose relations can be concisely described by table constraints. Judging by the results, decreasing the number of variables and constraints can significantly improve the performance. Since TCPP also solved the highest total number of problems while having a similar number of constraints as our final version of the R$^2\exists$-Step encoding, we conclude that reducing the number of variables and using suitable types of constraints may be more beneficial than reducing the number of solver calls. Our implementation of TCPP also solved more problems from most of the common benchmarks (*airport*, *miconic*, *storage*, *tpp*, and *zenotravel*) in the time limit of 60 minutes than the original version of Ghooshchi et al. [2017]. However, Ghooshchi et al. [2017] limited memory to 4 GB during their experiments.

Conclusions

We implemented three compilation-based planning approaches, TCPP, R$^2\exists$-Step encoding and Reinforced Encoding, in Picat 2.8. The source codes are available for download at https://bitbucket.org/krpant/compilation-based-approaches-to-parallel-planning. Firstly, we showed how these encodings could be implemented more efficiently. In particular, we suggested a more concise implementation of the R$^2\exists$-Step encoding, and we showed how the ranking of actions influences the performance. Next, we compared two different definitions of a parallel step in Reinforced Encoding. Secondly, we compared the encodings on benchmarks from International Planning Competitions. As TCPP outperformed the R$^2\exists$-Step encoding, we suppose that the formulation of the planning problem, including types of constraints and number of variables, may affect the efficiency more than the number of SAT solver calls.

Since all these encodings are makespan-optimal, they produce plans with redundant actions. Future work could focus on decreasing plan lengths by eliminating redundant actions. In the future, we could also deal with incremental solving as it is employed in the most recent papers related to SAT-based planning [Gocht and Balyo, 2017].

Acknowledgments

Research is supported by Czech Science Foundation under the project 18-07252S, by SVV project number 260 575 and by the Charles University, project GA UK number 156121.

References

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS+ planning. *Computational Intelligence* 11(4):625–655.

Balyo, T.; Barták, R.; and Trunda, O. 2015. Reinforced encoding for planning as SAT. *Acta Polytechnica CTU Proceedings* 2(2):1–7.

Balyo, T. 2013. Relaxing the relaxed exist-step parallel planning semantics. In *Proceedings of the 25th IEEE International Conference on Tools with Artificial Intelligence*, 865–871.

Barták, R. 2011a. A novel constraint model for parallel planning. In *Proceedings of the 24th International Florida Artificial Intelligence Research Society Conference*, 9–14.

Barták, R. 2011b. On constraint models for parallel planning: The novel transition scheme. In *Proceedings of the 11th Scandinavian Conference on Artificial Intelligence*, volume 227, 50–59.

Do, M. B., and Kambhampati, S. 2000. Solving planning-graph by compiling it into CSP. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling*, 82–91.

Ghallab, M.; Knoblock, C.; Wilkins, D.; Barrett, A.; Christianson, D.; Friedman, M.; Kwok, C.; Golden, K.; Penberthy, S.; Smith, D.; Sun, Y.; and Weld, D. 1998. PDDL – the planning domain definition language. AIPS-98 Planning Competition Committee.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Elsevier.

Ghooshchi, N. G.; Namazi, M.; Newton, M. H.; and Sattar, A. 2017. Encoding domain transitions for constraint-based planning. *J. of Artificial Intelligence Research* 58:905–966.

Gocht, S., and Balyo, T. 2017. Accelerating SAT based planning with incremental SAT solving. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling*, 135–139.

Helmert, M. 2006. The Fast Downward planning system. *J. of Artificial Intelligence Research* 26:191–246.

Kautz, H. A., and Selman, B. 1992. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, 359–363.

Kautz, H., and Selman, B. 1999. Unifying SAT-based and graph-based planning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, 318–325.

Lopez, A., and Bacchus, F. 2003. Generalizing GraphPlan by formulating planning as a CSP. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, volume 3, 954–960.

Rintanen, J.; Heljanko, K.; and Niemelä, I. 2006. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence* 170(12-13):1031–1080.

Wehrle, M., and Rintanen, J. 2007. Planning as satisfiability with relaxed $\exists$-step plans. In *Proceedings of the 20th Australasian Joint Conference on Artificial Intelligence*, 244–253. Springer.

www.picat-lang.org. 2019. Picat 2.8. Computer software.