Two Semantics for Step-Parallel Planning: Which One to Choose?

Tomáš Balyo[1], Daniel Toropila[12], and Roman Barták[1]

[1] Faculty of Mathematics and Physics, Charles University
Malostranské nám. 2/25, 118 00 Prague, Czech Republic
{tomas.balyo,daniel.toropila,roman.bartak}@mff.cuni.cz
[2] Computer Science Center, Charles University
Ovocný trh 5, 116 36 Prague, Czech Republic

Abstract. Parallel planning is a paradigm that provides interesting efficiency improvements in the field of classical AI planning and it is one of the key components of successful SAT-based planers. Popularized by the Graphplan algorithm, it provides more structure information about the plan for a plan executor when compared to the traditional sequential plan, which is one of the crucial facts of its usability in real-life scenarios and applications. Our latest research shows that different semantics can be used for parallel planning, while deciding which semantic to use for given application can have significant influence on both practical planning domain modeling and also on the computational efficiency of searching for a plan.

Keywords: automated planning, parallel semantics, SAT, domain modeling

1 Introduction

Classical AI planning deals with finding a sequence of actions that transfer the world from some initial state to a desired state. We assume a world which is fully observable (we know precisely the state of the world), deterministic (the state after performing the action is known), and static (only the entity for which we plan changes the world), with a finite (though possibly large) number of states. We also assume actions to be instantaneous so we only deal with action sequencing. Actions are usually described by a set of preconditions – features that must hold in a state to make the action applicable to that state – and a set of effects – changes that the action makes to the state. Action sequencing is naturally restricted by causal relations between the actions, i.e., the effect of certain action gives a precondition of another action.

Traditional sequential planning algorithms explore directly the sequences of actions. One of the disadvantages of this approach is liability to exploring symmetrical plans where some actions can be swapped without changing the overall effect. Hence if some sequence of actions does not lead to a goal then the algorithm may explore a similar sequence of actions where certain actions are swapped though this sequence leads to exactly same non-goal state. This is called plan-permutation symmetry [1]. It is possible to remove some of these symmetries by symmetry breaking constraints as suggested in [2] or [3]. Another way to resolve this problem is partial-order planning where the plans are kept as partially ordered sets of actions (the partial order respects the causal relations). CPT planner [4] is probably the most successful (in terms of International Planning Competition) constraint-based planner that does partial-order planning. A half way between partial-order and sequential planning is parallel planning, where the plan is represented as a sequence of sets of actions such that any ordering of actions in the sets gives a traditional sequential plan. This concept was popularised by the Graphplan algorithm [5] that introduced a so called planning graph to efficiently represent causal relations between the actions. Planning graph became a popular representation of parallel plans for approaches that translate the planning problem to other formalisms such as Boolean satisfiability or constraint satisfaction [6] [7].

Besides the elegant handling of the symmetries, another advantage of parallel planning is its strong usability for actual plan execution as the concept of parallel plan is much closer to the real-life scenarios. As an example we can imagine a transportation planning domain, such as logistics [8], where the available actions represent mainly movements of vehicles and loading/unloading of these. Obviously, it is proficient for a plan executor to know that loading of one vehicle can be realized in parallel with sending another vehicle from one location to another. Other useful examples can be devised very easily.

Even though the concept of parallel planning is straightforward, our latest research shows that multiple useful semantics can be introduced for parallel planning by modifying the constraints specifying

which actions can appear together in a single (parallel) step of a plan. In this paper we study the differences between the two main semantic concepts, proving the answers for two main questions: first we are investigating the influence of these differences on practical domain modeling, while, second, our main focus is exploring the impact on performance of searching for a plan.

The paper is organized as follows. First we provide necessary theoretical concepts together with the description of the two different parallel plan semantics we studied. After that, following sections discuss the impact of the differences between the two semantics on the practical usability for planning domain modeling and also on the computational efficiency. We then conclude by providing the experimental evaluation and final discussion of our results.

2 SAS$^+$ Planning

We use SAS$^+$ formalism to formalize the planning problem. This formalism is based on so called multi-valued *state variables*, as mentioned in [9] or [10]. For each feature of the world, there is a variable describing this feature, for example the position of a robot. World state is then specified by values of all state variables at the given state. Hence the evolution of the world can be described as a set of state-variable functions where each function specifies evolution of values of certain state variable. Actions are then the entities changing the values of state variables. Each action consists of preconditions specifying required values of certain state variables and effects of setting the values of certain state variables. We implicitly assume the frame axiom, that is, other state variables than those mentioned among the effects of the action are not changed by applying the action.

Formally, a planning task P in the SAS$^+$ formalism, alternatively referred to also as a SAS$^+$ planning problem, is defined as a tuple $P = \{X, A, s_I, s_G\}$, where

- $X = \{x_1, ..., x_n\}$ is a set of state variables, each with an associated finite domain $Dom(x_i)$;
- A is a set of actions and each action $a \in A$ is a tuple $(pre(a), \mathit{eff}(a))$, where $pre(a)$ and $\mathit{eff}(a)$ are sets of (partial) state variable assignments in the form of $x_i = v, v \in Dom(x_i)$;
- A state s is a set of assignments with all the state variables assigned. We denote s_I as the initial state, and s_G as a partial assignment that defines the goal (thus, there can be multiple states that satisfy the goal). We say state s is a goal state if $s_G \subseteq s$.

To express that a state variable x is assigned a value $v \in Dom(x)$ in a state s, we write $s(x) = v$. We say a is *applicable* in state s if all assignments in $pre(a)$ match the evaluation of variables in s. We then use $\gamma(s, a)$ to denote the state after applying a to s (according to the assignments defined in $\mathit{eff}(a)$).

Based on the formalism above we can also define the transitions that correspond to changes of variable assignments, as described in [11]. Given a state variable x, a *transition* is a change of the assignment of x from value f to g, denoted as $\delta^x_{f\to g}$, or from an unknown value to g, denoted as $\delta^x_{*\to g}$. Where unambiguous, the notation can be simplified to δ^x or δ.

For an action a, three types of transitions can be identified:

1. Transitions $\delta^x_{f\to g}$ such that $(x = f) \in pre(a)$, and $(x = g) \in \mathit{eff}(a)$. We call this type of transitions *active*. An active transition $\delta^x_{f\to g}$ is applicable to a state s, if and only if $s(x) = f$. We denote $\gamma(s, \delta)$ as the state after applying transition δ to state s, which results in a new state s' such that $s'(x) = g$.
2. Transitions $\delta^x_{f\to f}$ such that no assignment to x is in $\mathit{eff}(a)$, and $(x = f) \in pre(a)$. We call this type of transitions *prevailing*. A prevailing transition $\delta^x_{f\to f}$ is applicable to a state s, if and only if $s(x) = f$, resulting in the same state s.
3. Transitions $\delta^x_{*\to g}$ such that no assignment to x is in $pre(a)$, and $(x = g) \in \mathit{eff}(a)$. We call this type of transitions *mechanical*. A mechanical transition $\delta^x_{*\to g}$ can be applied to an arbitrary state s, with the result being a state s' where $s'(x) = g$.

For each action a, we denote its *transition set* as $M(a)$, which includes all three types of transitions above. Given a transition δ, we use $A(\delta)$ to denote the set of actions that include δ in their transition set. We call $A(\delta)$ the *supporting action set* of δ. Further, we use $R(x) = \{\delta^x_{f\to f} | \forall f, f \in Dom(x)\}$ to denote the *set of all prevailing transitions* related to x. Then we can introduce $T(x) = \{\delta^x | \exists a \in A, \delta^x \in M(a)\} \cup R(x)$, which represents the *set of all legal transitions* for the state variable x.

3 Two Semantics

When having a closer look at a sequential plan, one can very often realize that some of the actions included in the plan could be in fact swapped, or even executed together, without affecting the final

result of the plan execution. This is probably not surprising observation, as many plans typically involve almost independent sub-plans, that have to be synchronized with each other only in a few specific time steps. As an example of such we can use a plan that involves activities for two trucks, each of which can have its own separate sub-plan of moves and loading/unloading some cargo, while the only needed synchronization between the two trucks is necessary in case one of them is waiting for the cargo that other truck is to deliver first, or in case both of them need to load/unload their cargo in a location where only one truck can operate at a time. Instead of a plain action sequence $\pi = \{a_1, ..., a_k\}$ we could express a solution for a given planning problem using a richer structure – by providing a sequence of sets of actions $\pi' = \{A_1, ..., A_m\}, \forall A_i \exists j : a_j \in A_i$, where for each set A_i all of the actions contained within could be executed in parallel. Of course, one could go further and provide a full graph of causal relations of individual actions contained in the plan, obtaining thus a partial-order plan [4]. However, constructing such a structure might not be too simple, and is out of the scope of this paper.

There are two motivations behind using parallel plans. First, it is a natural way how to remove some of the plan-permutation symmetries, so the planner does not have to explore the plans where a pair of actions within a set corresponding to single time step is swapped. This fact should, ultimately, help to improve the performance of a planner. Second motivation is even more practical – the knowledge provided by a parallel plan structure is, in fact, very useful for the plan execution phase, since the executor knows which actions can be performed in parallel (if possible), and thus the whole plan can be executed using less time steps.

There are, however, two different ways to define which actions can appear together in a single (parallel) time step, providing thus two different semantics for the step-parallel planning.

3.1 Strict Semantics

Probably the most natural requirement on the actions in a single set A_i is their strict pairwise independence, as defined in [12], and also as originally described in the Graphplan planning system [5], which was the first one to introduce step-parallel plans as a technique to rapidly improve the efficiency of solving planning problems. Needless to say, the preconditions of the actions within A_i must be non-conflicting.

We say that two actions a_1, a_2 are *independent* in case a_1 does not destroy the preconditions or effects of a_2, and vice versa. The consequence of the requirement of pairwise independence between actions from A_i together with their non-conflicting preconditions is that the actions from a single set A_i can be executed in an arbitrary sequential or parallel order. Therefore the only synchronization requirement on the plan execution is that prior to executing actions from a set A_i all action from A_{i-1} must be executed.

If two actions cannot appear within the same set A_i, we say they are mutually exclusive, i.e., they are in *action mutex*. The fact of which actions are allowed to appear within the same set A_i can be also expressed using the corresponding transitions. The actions a_1, a_2 are in action mutex if there exists a pair of transitions $\delta_1 \in M(a_1)$ and $\delta_2 \in M(a_2)$ such that these two transitions are in *transition mutex*. The two different definitions of the transition mutex will help us draw the difference between the two semantics discussed in this paper.

First, let us define the transition mutex for the *strict semantics*. A pair of transitions δ_1, δ_2 is in *strict transition mutex* if there exists $x \in X$ such that $\delta_1 \in T(x)$ and also $\delta_2 \in T(x)$, meaning they are both based on the same state variable. In other words, all transitions of a given state variable are in transition mutex.

The above relation is a corollary of the following observation. Given the actions a_1, a_2, if there is no pair of transitions $\delta_1^x \in M(a_1), \delta_2^y \in M(a_2)$ such that $x = y$, i.e., both transitions are based on the same state variable, then a_1, a_2 are independent and have non-conflicting preconditions. For the sake of evidence we include a sketch of the proof. If there is no common state variable for any pair of transitions δ_1^x, δ_2^y, the actions a_1, a_2 can neither affect each other nor have conflicting preconditions, since if they did so, there would be a state variable proving the conflict together with the associated transitions – which proves the required implication.

Interestingly, it turns out that the reverse implication does not hold. Hence, that means that the definition using the transition mutex is stricter than the one using the action independence (the fact of which also provided the name for this semantics). However, as proven above, it still guarantees the possibility of an arbitrary ordering during the execution of actions within A_i.

Table 1. Example of a set of actions that can be executed only in parallel.

Action	*Preconditions*	*Effects*	*Associated Transitions*
a_1	$x = a$	$x \leftarrow b$, $z \leftarrow f$	$\delta^x_{a\to b}$, $\delta^z_{*\to f}$
a_2	$y = c$	$y \leftarrow d$, $x \leftarrow b$	$\delta^y_{c\to d}$, $\delta^x_{*\to b}$
a_3	$z = e$	$z \leftarrow f$, $y \leftarrow d$	$\delta^z_{e\to f}$, $\delta^y_{*\to d}$

3.2 Synchronized Semantics

For the previous semantics we required the universal interchangeability and executability (whether sequential or parallel) for all actions within a single set A_i. In other words, all possible orderings of the actions led to the same state. It is however possible to introduce a less strict semantics, that will in turn pose some additional requirements on the action execution phase. Let us first define the new semantics using a different specification of transition mutex.

For *synchronized semantics*, two different transitions δ_1 and δ_2 are mutually exclusive, i.e., δ_1 and δ_2 are a pair of transition mutex, if there exists a state variable $x \in X$ such that $\delta_1, \delta_2 \in T(x)$, and either of the following holds:

1. Neither δ_1 nor δ_2 is a mechanical transition.
2. Both δ_1 and δ_2 are mechanical transitions.
3. Only one of δ_1 and δ_2 is a mechanical transition and they do not transit to the same variable assignment.

To better interpret the definition above, there are only two cases when $\delta_1, \delta_2 \in T(x), \delta_1 \neq \delta_2$ are not mutex. The first case is, without loss of generality, when $\delta_1 = \delta^x_{f\to f}$ and $\delta_2 = \delta^x_{*\to f}$ for any value $f \in Dom(x)$. The second case is, when $\delta_1 = \delta^x_{e\to f}$ and $\delta_2 = \delta^x_{*\to f}$ for any values $e, f \in Dom(x), e \neq f$.

In order to illustrate the semantical difference between the two definitions of transition mutex, consider the three actions from Table 1. According to the strict semantics, no two actions of these are allowed to appear within a set A_i, since a_1, a_2, a_3 are all pairwise mutex. On the other hand, according to the synchronized semantics, none of these actions are mutually exclusive, and therefore all of them can appear together within a single parallel step as long as their preconditions are met. However, there is a following complication: no valid sequential ordering exists for these three actions! The reason for this is that anytime an action is executed, it destroys a precondition of some other action. Still, it is possible to execute these three actions in a valid way within a single step – but only with the condition of their perfectly synchronized parallel execution.

As we depicted using the above example, the synchronized semantics does not guarantee the possibility of an arbitrary sequential or parallel order of action execution. In fact, there might be no sequential execution available at all for a given parallel step A_i, in case of which the synchronized parallel execution of some of the actions might be required in order not to break the plan validity.

4 Impact on Usability

Clearly, both semantics allow different actions within the steps of a parallel plan. Before we study the performance of these two semantics, let us have a closer look at their influence on the planning domain modeling.

The main objection that can be raised against the practical use of the synchronized semantics is that the precise synchronization of the start of the actions is mostly infeasible in practice, so the planning domain designer does not want his model to be too fragile on the time constraints. Even though being a bit absurd, consider the following Bomb Terrorist planning domain. Let our modeled world consist of three terrorist t_1, t_2, t_3, each of them wearing a pack of dynamite. As usual in their community, only a terrorist that actually fires his explosives can reach the eternal fame, which is, obviously, everyone's ultimate goal. However once a bomb explodes, everything around is exterminated. Given the terrorist are located inside a room (together with a secret wooden box they are supposed to eliminate), the actions from Table 2 are available in the modeled world.

It is very unlikely that in such scenario it would be possible for all of the terrorists to fire their explosives at once. Still, it is absolutely correct to ask planner the question whether a plan of providing the fame for all three terrorists exists. For our model, the answer of the planner would depend on the

Table 2. The actions available for the silly Bomb Terrorist domain.

Action	*Preconditions*	*Effects*
$fire_1$	$alive_1 = true$	$alive_1 = false$, $alive_2 = false$, $alive_3 = false$, $fame_1 = true$
$fire_2$	$alive_2 = true$	$alive_1 = false$, $alive_2 = false$, $alive_3 = false$, $fame_2 = true$
$fire_3$	$alive_3 = true$	$alive_1 = false$, $alive_2 = false$, $alive_3 = false$, $fame_3 = true$

underlying semantics it implements. A planner that complies with the strict semantics would answer that no such plan exists, while, on the other hand, a planner compliant with the synchronized semantics would return a plan consisting of a single parallel step that would include all available fire-actions.

Using the example above we illustrated that different semantics of the step-parallel planning used for the same domain can provide completely different results regarding the existence of a plan. Moreover, other examples can be constructed where the return plans will differ in the plan length or total number of used actions. Since we used very small and simple example, it is very easy to see the potential issues, however for the greater models with tens or hundreds of actions and state variables similar issues might be very difficult to trace.

The construction of the planning domain model is purely a task of a domain designer, however a special care must be taken in order to keep the required relevance to the reality, in case the planning technology to be used returns parallel plans. An example of a planner that uses synchronized semantics is SASE planning system, as described and published in [11].

5 Impact on Performance

Regardless of the usability of the described parallel plan semantics, we were interested in their practical performance. In other words, we wanted to know the use of which semantics leads to finding plans faster. In order to find out we implemented the two versions of a SAT-based planner, which differ only in the definition of the transition mutex relation, as described in one of the earlier chapters. For both versions, the total number of the instances solved within a time limit together with the time required to solve them was measured on a large set of standard planning benchmark problems from the previous International Planning Competitions (IPC). Further details of our planner and the experiments will be described later in this paper.

Knowing which parallel plan semantics is practically faster would be useful when deciding which semantics to use at the phase of planning domain design.

5.1 Planner description

Our planner is a Java application which takes SAS$^+$ files as the input. We will refer to it as *SasPlan*. In order to translate PDDL files to SAS$^+$ formalism we used Helmert's Translator tool implemented in Python [10]. For each planning problem within our benchmark set it took at most a few seconds for the Translator to generate the SAS$^+$ input file. Since our goal was to compare the efficiency of the two parallel plan semantics, for the final evaluation we only measured the runtime of our application, not including the translation time (which was also fast compared to the time necessary for solving).

SasPlan is basically a black-box planner which encodes SAS$^+$ problems into satisfiability (SAT) problems using the SASE transition-based encoding [11]. The operation of the planner can be roughly summarized as follows. First we generate a SAT formula that is satisfiable if and only if there is a plan with timespan equal to one. If the formula is unsatisfiable, we increase the timespan by one. Ultimately, we generate a satisfiable formula, and then we can extract a valid parallel plan from its satisfying assignment.

To solve the generated SAT problems we used SAT4J – a Java library for satisfiability by Daniel Le Berre [13]. Although SAT4J is a few times slower than state of the art solvers written in C/C++, it supports incremental solving and is very easy to use within a Java application. Employing a state-of-the-art C++ solver would considerably increase the overall performance of the planner, since most of the runtime is spent on solving the formulas. MiniSat 2.2 [14] would be an appropriate choice, as it is one of the fastest SAT solvers, supporting also the incremental solving technique. Nevertheless, in order to examine the impact of the different semantics on the performance, the use of Java-based SAT solver was sufficient.

Table 3. Total number of solved instances in the time limit of 30 minutes, per domain.

Domain	*Strict*	*Synchronized*	*Difference*
Airport	30	27	+3
Depots	13	12	+1
Driverlog	15	14	+1
Elevator	46	47	-1
Freecell	4	4	0
Openstack	5	5	0
Rovers	31	32	-1
TPP	27	27	0
Zenotravel	15	14	+1

The way we use the support for incremental solving is very straightforward. When we need to generate and solve the formula for the next timespan, we first remove the clauses that require the goal conditions to hold at the end of the plan. Second, we add new variables for the next time step, together with all the clauses that ensure plan validity and the goal conditions. During the tests performed prior to the evaluation itself we observed that the SAT solver did gain some benefit from being used incrementally, compared to getting a new unknown formula at each time step. We did not however do thorough experiments on a large data set in order to compare the incremental and non-incremental approach in bigger detail, since in all tests we made the incremental version was always faster. Therefore we decided to use the incremental solving approach as a default configuration for both versions of our planner.

As stated earlier, we created two versions of SasPlan that correspond to the two transition mutex definitions. During the implementation phase we noticed that the strict semantics is much easier to encode. Since according to the strict mutex definition all transitions of a state variable are mutex, it is sufficient to add one constraint for every state variable ensuring that at most one of the corresponding transitions can be selected at given time step. In other words, a graph of transition mutex relations is a clique, which can be encoded into SAT clauses very efficiently, and which is also one of the reasons why our strict semantics is defined in a little bit stricter way compared to the definition using action independence. Actually, the interface of SAT4J provides a method for adding the *at-most-one* constraints and thus the transition mutex constraints can be added by just one method invocation for each variable. On the other hand, the mutex graph for the synchronized semantics is not a clique since not all transitions of a state variable are pairwise mutex. Therefore these relations cannot be encoded that efficiently.

5.2 Experiments

For the experiments we used a PC with 3.2GHz Intel Core i7 processor and 24GB of RAM. We limited the memory to 4GB and used a time limit of 30 minutes per instance. For the evaluation of the planner performance the following domains were selected from the available International Planning Competition (IPC) benchmark set: Airport, Depots, Driverlog, Elevator, FreeCell, Openstacks, Rovers, TPP and Zenotravel.

Before the actual experiments it was difficult to predict the results of our experiments, and therefore our expectations were not clear. The strict transition mutex definition constraints the problem more and thus the resulting SAT formula has less or equal solutions for a given makespan than for the other mutex definition. This could make us believe that the performance can be decreased. On the other hand, more constraints help unit propagation to prune the search space more efficiently. Because of these reasons it was very hard to tell in general which reasoning was stronger and how the performance of the SAT solver will be influenced.

In Table 3 we provide the total count of solved instances within the time limit, per domain. The differences are not big and for all domains except Elevator and Rovers the strict semantics solved greater or equal number of instances. Figure 1 presents the runtime increase ratio when solving the problems using the synchronized semantics compared to the strict semantics. We only considered the running times for those problems that were solved by both versions of SasPlan. Values above zero mean that the strict semantics is faster. For example, the y-value of 1 means that the runtime using the synchronized semantics was twice as long as the runtime using the strict semantics, i.e., $y =$

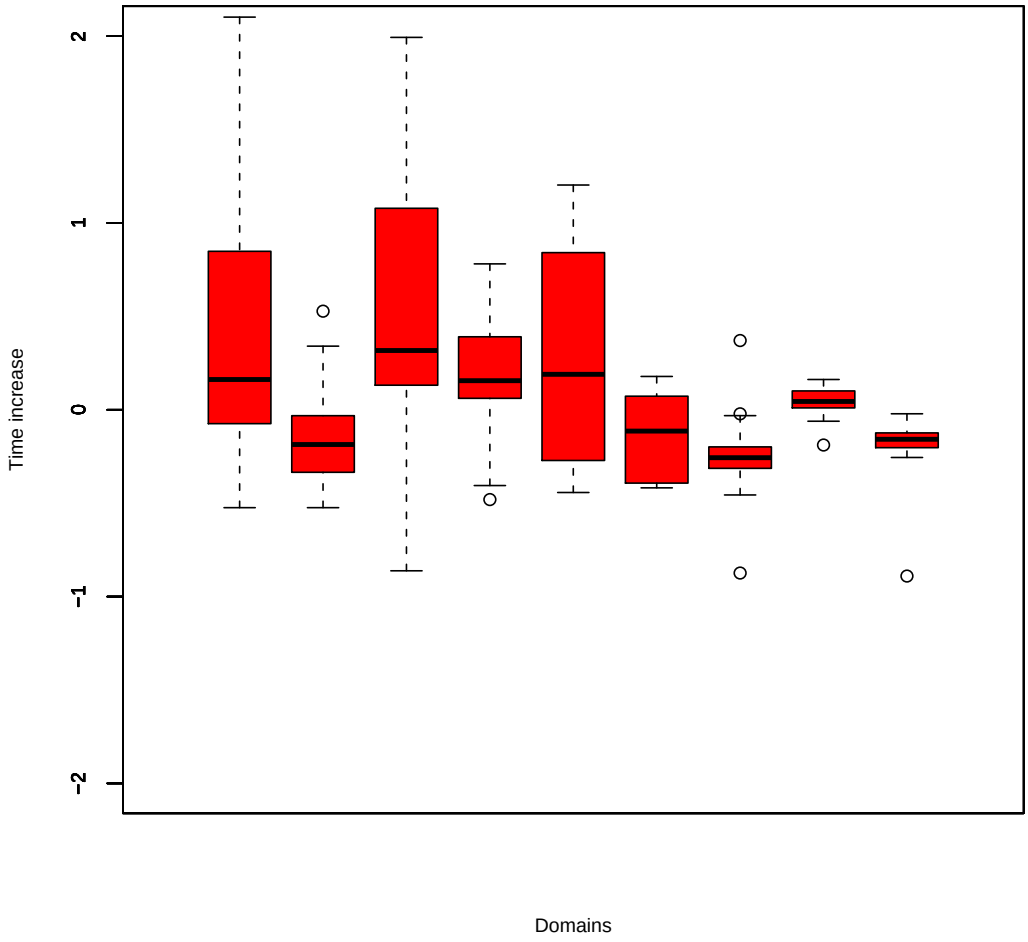

Fig. 1. Box plot representation of the runtime increase ratio when solving the problems using the synchronized semantics compared to the strict semantics. Values above zero mean that the strict semantics is faster. For example, the y-value of 1 means that the runtime using the synchronized semantics was twice as long as the runtime using the strict semantics. The box plots represent the results for domains in following (alphabetical) order: Airport, Depots, Driverlog, Elevator, FreeCell, Openstacks, Rovers, TPP and Zenotravel

$(runtime_{sync} - runtime_{strict})/runtime_{strict}$. From the box plots we can see that even though the strict semantics is not a clear winner in all domains, for some problems its runtime was over 3x faster compared to the synchronized semantics, while the experienced slowdown for other problems was less evident.

Overall, we can conclude that even though the difference between the efficiency of the presented semantics for the parallel planning is not dramatic, the strict semantics exhibited better performance compared to the synchronized one. Since the strict semantics is also more natural and practical for the real-world applications, we believe it should be preferred for majority of cases.

6 Conclusion

Step-parallel planning is a paradigm useful both for dealing with some of the plan-permutation symmetry problems and also for a practical execution of plans.

In this work we described the two available semantics for the parallel planning, for which also the implementations exist: strict semantics and synchronized semantics. After formally describing the theoretical concepts behind the two semantics, we demonstrated that the planning domain designer's awareness

of the selected parallel planning technology and its underlying semantics is a crucial component of a successful application of automated planning in the real-world scenarios.

Finally, we provided the empirical evaluation of the performance of the two presented semantics by integrating them into the implemented SAT-based planning system. The experiments, for which we used some of the traditional IPC benchmark domains, showed that the strict semantics provides better performance in terms of runtime, while it is also more natural for the practical application.

Acknowledgment

The research is supported by the Czech Science Foundation under the projects no. P103/10/1287, 201/09/H057, SVV project number 263 314 and by the Charles University Grant Agency under contracts no. 9710/2011 and 266111.

References

1. Long, D., Fox, M.: Plan Permutation Symmetries as a Source of Planner Inefficiency. In: Proceedings of UK Workshop on Planning and Scheduling. (2003)
2. Grandcolas, S., Pain-Barre, C.: Filtering, Decomposition and Search Space Reduction for Optimal Sequential Planning. In: AAAI, AAAI Press (2007) 993–998
3. Barták, R., Toropila, D.: Revisiting Constraint Models for Planning Problems. In: ISMIS '09: Proceedings of the Eighteenth International Symposium on Foundations of Intelligent Systems, Berlin, Heidelberg, Springer-Verlag (2009) 582–591
4. Vidal, V., Geffner, H.: Branching and Pruning: An Optimal Temporal POCL Planner Based on Constraint Programming. Artificial Intelligence **170**(3) (2006) 298–335
5. Blum, A., Furst, M.L.: Fast Planning Through Planning Graph Analysis. Artificial Intelligence **90**(1-2) (1997) 281–300
6. Do, M.B., Kambhampati, S.: Planning as Constraint Satisfaction: Solving the Planning Graph by Compiling It into CSP. Artificial Intelligence **132**(2) (2001) 151–182
7. Lopez, A., Bacchus, F.: Generalizing GraphPlan by Formulating Planning as a CSP. In Gottlob, G., Walsh, T., eds.: IJCAI '03: Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, Morgan Kaufmann (2003) 954–960
8. Long, D., Kautz, H.A., Selman, B., Bonet, B., Geffner, H., Koehler, J., Brenner, M., Hoffmann, J., Rittinger, F., Anderson, C.R., Weld, D.S., Smith, D.E., Fox, M.: The AIPS-98 Planning Competition. AI Magazine **21**(2) (2000) 13–33
9. Bäckström, C., Nebel, B.: Complexity Results for SAS+ Planning. Computational Intelligence **11** (1995) 625–656
10. Helmert, M.: The Fast Downward Planning System. Journal of Artificial Intelligence Research (JAIR) **26** (2006) 191–246
11. Huang, R., Chen, Y., Zhang, W.: A Novel Transition Based Encoding Scheme for Planning as Satisfiability. In Fox, M., Poole, D., eds.: AAAI, AAAI Press (2010)
12. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann Publishers (2004)
13. Berre, D.L., Parrain, A.: The Sat4j library, release 2.2. JSAT **7**(2-3) (2010) 59–6
14. Eén, N., Sörensson, N.: An Extensible SAT-solver. In Giunchiglia, E., Tacchella, A., eds.: SAT. Volume 2919 of Lecture Notes in Computer Science., Springer (2003) 502–518