Transforming and Encoding FTS for SAT Solving: What Helps, What Hurts

João Filipe[1] , **Álvaro Torralba**[2] , **Gregor Behnke**[1]
[1]University of Amsterdam, Institute for Logic Language and Computation, The Netherlands
[2]Aalborg University, Denmark
{j.sa, g.behnke}@uva.nl, alto@cs.aau.dk

Abstract

Factored tasks are a classical planning representation that extends SAS+ with limited forms of disjunctive preconditions, conditional effects, and angelic nondeterminism. This allows for a more compact representation of tasks than traditional formalisms such as STRIPS or SAS+, and supports a wide range of task transformations. However, existing planning approaches for factored tasks have been limited to heuristic search methods.

In this work, we investigate how to encode factored tasks in SAT. We propose several ways to encode the tasks, focusing on different strategies for translating the factored transition relation into propositional logic. We also analyze how to exploit parallelism at various levels in this setting and study the impact of common task transformations on the performance of SAT-based planners.

1 Introduction

Classical planning algorithms aim to exploit a model of the environment to find a plan. Different representations can be used to model the environment, e.g., STRIPS [Fikes and Nilsson, 1971] or SAS$^+$ [Bäckström and Nebel, 1995], offering a trade-off between expressiveness and ease to exploit for planning algorithms. We consider planning tasks in a factored transition system (FTS) representation, where states are described in terms of finite-domain variables, and each variable (also called factor) is described as a transition system. This provides a lot of flexibility for expressing the possible behaviors (preconditions and effects) of an action within one variable, provided independence of this behavior across variables. The FTS representation was originally considered in the context of merge-and-shrink abstractions [Helmert *et al.*, 2014], but was later adopted as a planning task representation due to having several advantages over previous representations. Compared to SAS$^+$, for example, we can easily express for each variable disjunctive preconditions, conditional effects, and even angelic non-determinism (where we can choose multiple outcomes for a given action). Expressing the same transition semantics in SAS$^+$ requires either exponential-size compilation or introducing auxiliary state variables and operators that increase plan length [Nebel, 2000; Büchner *et al.*, 2024]. On the one hand, it allows to naturally encode domains that are not easy to express in SAS$^+$, such as finding algorithms for matrix multiplication [Speck *et al.*, 2023], or Rubik's cube [Korf, 1997; Büchner *et al.*, 2024]. On the other hand, it provides a very flexible representation for task transformation [Torralba and Sievers, 2019; Sievers and Helmert, 2021].

Thus, it is desirable to have planning methods that support this representation natively. So far, existing planning approaches for factored tasks have focused more on optimal planning, using heuristic search methods with admissible heuristics [Torralba and Sievers, 2019; Büchner *et al.*, 2024], or symbolic search [Torralba *et al.*, 2017; Torralba *et al.*, 2023]. For agile planning, only heuristic search algorithms exist [Torralba and Sievers, 2019; Torralba *et al.*, 2023]. These are not always suitable for the FTS representation. For example, an FTS can encode tasks with exponential branching factor in the size of the task, where straightforward heuristic search approaches fail immediately.

SAT-based planning has been a prominent approach since the early work of Kautz and Selman [1992]. The general idea is to encode the existence of a plan of length N as a propositional formula ϕ_N, which is satisfiable if and only if such a plan exists. This formula is then passed to a SAT solver, either incrementally increasing N for optimal planning, or following a schedule for satisficing planning [Rintanen, 2012; Rintanen, 2011]. Over the years, several SAT-based planning systems have been developed, such as SATPLAN04 [Kautz *et al.*, 2006], Madagascar [Rintanen, 2012], action splitting [Robinson *et al.*, 2009], abstract CNF encodings [Domshlak *et al.*, 2009] PASAR [Froleyks *et al.*, 2019], AxSAT [Behnke *et al.*, 2025], and SASE [Huang *et al.*, 2012], each contributing different encoding strategies or solver techniques. However, these systems typically target classical representations and are not directly applicable to FTS tasks. Adapting SAT planning to this setting enables new encoding strategies that make better use of the structure inherent in factored models.

In this work, we consider several alternative ways to encode FTS planning tasks as a SAT formula. While a naive encoding is straightforward, we propose several encodings for the transition relation, as well as exploiting parallelism. Our experiments show that these optimizations are fundamental to achieve a good performance, that SAT-based planning can be

competitive as well for solving factored tasks, and have good synergy with task reformulation methods.

2 Preliminaries

A *transition system* (TS) is a tuple $\Theta = \langle S, L, T, s^I, S^G \rangle$ where S is a finite set of *states*, L is a finite set of *labels*, $T \subseteq S \times L \times S$ is a set of *transitions*, $s^I \in S$ is the *initial state*, and $S^G \subseteq S$ is the set of *goal states*. A *path* $s \xrightarrow{l_1} \dots \xrightarrow{l_k} t$ from s to t is a sequence of k transitions $(s_{i-1}, l_i, s_i) \in T$ for $1 \leq i \leq k$, starting in $s = s_0$ and ending in $t = s_k$. A *plan* for s is a path from s to any $s' \in S^G$.

2.1 Factored Tasks

We consider planning tasks in a factored transition system representation, where states are described in terms of finite-domain variables, and each variable (also called factor) is described as a TS [Helmert *et al.*, 2014; Torralba and Sievers, 2019; Sievers and Helmert, 2021; Büchner *et al.*, 2024]. A factored task $\mathcal{T} = \langle \Theta_1, \dots, \Theta_n \rangle$ is a tuple of TSs with a common set of labels, L, such that $\Theta_i = \langle S_i, L, T_i, s_i^I, S_i^G \rangle$ for $1 \leq i \leq n$. We assume that all S_i are disjoint, i.e., $S_i \cap S_j = \emptyset$. We characterize the size of task $\mathcal{T}$ in terms of the number of factors $|\mathcal{T}| = n$, the number of labels $|L|$, and the maximum number of states in each factor $D = \max_{\Theta_i \in \mathcal{T}} |S_i|$.

This is a compact representation of a state space, which is the synchronized product $\Theta_\otimes = \bigotimes_{i=1}^{n} \Theta_i$ of all factors. A state in $\Theta_\otimes$ is a tuple of states from the individual factors $s = \langle s_1, \dots, s_n \rangle$, one for each factor, $s_i \in S_i$. The state space of a factored task is the synchronized product of all its factors. This is another transition system $\Theta_\mathcal{T} = \Theta_i \otimes \dots \otimes \Theta_n = \langle S_\mathcal{T}, L, T_\mathcal{T}, s_\mathcal{T}^I, S_\mathcal{T}^G \rangle$, where $S_\mathcal{T} = S_1 \times \dots \times S_n$ is the Cartesian product of each factor's states, $T_\mathcal{T} = \{(\langle s_1, \dots, s_n \rangle, l, \langle t_1, \dots, t_n \rangle) \mid (s_i, l, t_i) \in T_i \text{ for } 1 \leq i \leq n\}$, $s_\mathcal{T}^I = \langle s_1^I, \dots, s_n^I \rangle$, and $S_\mathcal{T}^G = S_1^G \times \dots \times S_n^G$.

2.2 Task Transformation

Apart from being able to easily express some planning tasks more naturally than other formalisms, an advantage of using the factored representation is that a large number of *transformations* are available [Sievers and Helmert, 2021]. Though many of these transformations are tailored towards creating abstractions of the task, some of them can also be used to simplify the task before solving it, if they have the property that a plan for the original task can be reconstructed from any plan for the transformed task [Torralba and Sievers, 2019]. Some of the transformations available include:

- Label reduction [Sievers *et al.*, 2014] reduces the set of labels while keeping the set of transitions of the product TS intact (up to label renaming).
- Weak-bisimulation shrinking [Hoffmann *et al.*, 2014] reduces the number of states in a factor. This may change the goal distance of states in $\Theta_\otimes$, but it preserves whether states were solvable or not so that a solution for the original task can be reconstructed.
- Pruning unreachable and dead-end states in the factors [Sievers and Helmert, 2021].
- Merging two or more factors into one [Fan *et al.*, 2014; Sievers *et al.*, 2016; Sievers *et al.*, 2024]. This reduces the number of factors, at the cost of increasing their size exponentially in the number of combined factors.

These transformations can be combined, having synergies [Sievers and Helmert, 2021]. For example, pruning typically is not applicable on the input task, but merging factors may lead to discovering that many of the combined states are unreachable or dead-ends. Notably, the transformations make use of the expressiveness of the representation. A factored task obtained from compiling a SAS$^+$ task will never have non-deterministic labels or disjunctive preconditions. However, label reduction or shrinking may introduce them.

2.3 Boolean Satisfiability

Boolean Satisfiability (SAT) is the task of determining whether a propositional formula in conjunctive normal form (CNF) has a truth assignment of its variables that makes the formula true. A CNF formula consists of a conjunction ($\wedge$) of clauses, where each clause is a disjunction ($\vee$) of literals. A literal is a propositional variable, or variable for short, x or its negation $\neg x$. SAT solvers, which are readily available, determine whether a formula is satisfiable, and if so, provide a satisfying truth assignment. For ease of notation, we will not present the SAT formulae in this paper in CNF, but all formulae can be translated easily to CNF.

3 SAT Planning

The general idea in SAT planning is to, given a planning task and a bound on the number of timesteps N, construct a SAT formula that is satisfiable if and only if there exists a plan with a makespan of N. Our encodings use the following variables, for each timestep $t \in \{0, \dots, N\}$:

- s_i^t: indicates if we are at state $s_i \in \Theta_k$ at timestep t.
- l^t: indicates if a label $l \in L$ is selected at timestep t.

Then, we need to encode constraints that ensure that the values of these variables correspond to an executable and goal-reaching path. In each factor, we have to be exactly in one state at every timestep. To enforce this, we assert that at least one state in each factor has to be selected via a disjunction ($\vee$) across all states of that factor. We also assert that at most one state in the factor is selected. Depending on the number of variables, its encoding varies. For up to 256 variables, a naive quadratic encoding is used, whereas a binary counter encoding is applied otherwise [Sinz, 2005].

$$atLeastOne(\{s^t \mid s \in S_k\}) \qquad \forall \Theta_k \in \mathcal{T} \qquad (1)$$

$$atMostOne(\{s^t \mid s \in S_k\}) \qquad \forall \Theta_k \in \mathcal{T} \qquad (2)$$

Additionally, the initial and goal states are enforced at the first and last timesteps respectively.

$$s_i^0 \qquad \forall s_i \in s^I \qquad (3)$$

$$\bigvee\nolimits_{s_i \in S_k^G} s_i^N \qquad \forall \Theta_k \in \mathcal{T} \qquad (4)$$

Finally, we must ensure that consecutive layers are consistent with the transition relation, i.e., for each t, s^{t+1} can be reached from s^t by applying the selected labels l^t. Next, we

focus on encoding the transition relation under the restriction that a single label can be selected at each timestamp.

$$atLeastOne(\{l^t \mid l \in L\}) \quad (5)$$

$$atMostOne(\{l^t \mid l \in L\}) \quad (6)$$

We will relax these constraints when we discuss parallelism.

4 Encoding of the Transition Relation

We present several ways to encode FTS tasks into SAT, which leverage the structure of the FTS representation. In an FTS task, labels capture all the dependencies across factors. So, it suffices to encode each Θ_k separately, restricting which pairs of states $s_i^t, s_j^{t+1} \in S_k$ are consistent with the label l^t selected at time t. We describe the encoding for a single factor Θ_k and timestep t. The final SAT formula is the conjunction of the encoding for every factor and time step, in addition to constraints 1 to 6.

The transition relation T_k can be seen as a 3D structure (with dimensions: source, target and label) indicating whether a transition is valid for a given triple of values. In a SAT encoding, the constraints do not "build solutions" directly but rule out those assignments that do not correspond to possible transitions. The simplest encoding of T_k is to explicitly list every impossible transition. Since at every timestep a state and a label must be chosen (constraints 1, 2, 5 and 6), the chosen transition must be one allowed in T_k. We call these constraints forbidding transitions *primal constraints*.

$$\neg(l^t \wedge s_i^t \wedge s_j^{t+1}) \qquad \forall (s_i, l, s_j) \notin T_k \quad (7)$$

Together constraints (1) through (7) already constitute a complete encoding of the FTS task as a SAT formula – which will be our baseline encoding. This encoding can be quite inefficient in terms of the number of clauses, as constraint 7 requires $O(|L| \cdot |\mathcal{T}| \cdot D^2)$ clauses. It is also fairly inflexible as it only asserts that transitions that are not present in the transition system cannot be used and does not leverage in any way the information of valid transitions. In the rest of the section we develop alternative ways of encoding equivalent formulas with the same satisfying assignments.

4.1 Constraint-per-line Encoding

Constraint 7 excludes transitions that do not exist in T_k. Each such constraint can be written as an implication stating that if two of the values are selected, then the third one cannot be selected: e.g. if we focus on the target state: $l^t \wedge s_i^t \implies \neg s_j^{t+1}$. We can now aggregate all such implications for label l and source state s_i into a single formula.

$$l^t \wedge s_i^t \implies \bigwedge\nolimits_{(s_i,l,s_j) \notin T_k} \neg s_j^{t+1} \quad \forall l \in L, \forall s_i \in \Theta_k \quad (8)$$

When focusing on a single label l, the transitions for l in T_k describe an adjacency matrix between the source and target states. Constraint 8 then encodes each row of l's adjacency matrix by forbidding targets which cannot be reached from the source corresponding to that row. Transforming constraint 8 to CNF yields the same formula as constraint 7.

Since at any time at most one state is selected per factor, we can use a positive version of the implication instead of the primal constraint. This *dual* constraint expresses the possible transitions of T_k. Since at most one possible transition can be chosen, no impossible transition can be executed.

$$l^t \wedge s_i^t \implies \bigvee\nolimits_{(s_i,l,s_j) \in T_k} s_j^{t+1} \quad \forall l \in L, \forall s_i \in \Theta_k \quad (9)$$

For every row of the adjacency matrix of a label l we can now either use the primal constraint constraint 7 or the dual constraint 9. The key difference lies in the number and size of generated clauses. Clauses of the primal constraint always have three literals, but require one clause per impossible target. The dual constraint can always be expressed as one clause, but contains two plus the number of possible targets many literals. I.e. using dual constraints for encoding yields fewer, but larger clauses. Dual constraints typically have weaker propagation as the clauses are larger and unit-propagation triggers only when the value of all variables except one is known. In order to avoid clauses with weaker propagation, we use the primal constraint 8 as a default. Only when $|\{s_j \mid (s_i, l, s_j) \in T_k\}| = 1$ both primal and dual constraints have the same size (three). Then we use the dual constraint to replace the primal constraints.

Example 1. *Consider the adjacency matrices for l_1 and l_2.*

l_1	A'	B'	C'
A	0	1	1
B	0	0	1
C	1	0	0

Table 1: l_1

l_2	A'	B'	C'
A	0	1	0
B	1	0	0
C	0	0	0

Table 2: l_2

In the adjacency matrix for label l_1, it can be seen that there is no self-loop in A. In order to encode this, the following constraint would be added $\neg(l_1 \wedge A \wedge A')$, which is equivalent to $l_1 \wedge A \implies \neg A'$. Alternatively, one could also encode this constraint as $l_1 \wedge A \implies B' \vee C'$, however this constraint would have less propagation power.

4.2 Projection Over 2 Dimensions

Constraints 8 and 9 each express an entire row of the adjacency matrix for a label l using either a conjunction of negated values or a disjunction of positive ones on the right-hand-side of the implication. But what happens if the right-hand side contains no literals? For constraint 8, we may transition from s_i with label l to any target – rendering the constraint useless as it will always evaluate to true. On the other hand, constraint 9 becomes more interesting. Here it is impossible to transition from s_i with label l to any target and the constraint 9 simplifies to $\neg s_i^t \vee \neg l^t$. This clause allows us to represent an entire row of the adjacency matrix – containing only zeros – with a single binary clause. This encoding is substantially more compact than constraint 8 using both fewer (only one) and smaller (size two) clauses. In fact, this is a new type of primal constraint stating that any transition involving s_i as the source and l as the label is impossible.

Previously, we arbitrarily selected the target dimension for aggregation to obtain constraint 8. Symmetrically, we can consider the aggregation on sources and labels. To create the dual constraints 9, aggregating on targets is beneficial in our evaluation benchmark, as labels typically have few possible

targets given a source. If aggregation however leads to a new primal constraint with only two variables, any aggregation is beneficial as it allows to remove some of the original primal constraints 7. We consider all three possible aggregations of the transition relation – which we term the *projection optimization* ("project away" one dimension).

$$T_k^{\mathtt{s}}[l][s_j] = \{s_i \mid (s_i, l, s_j) \in T_k\} \tag{10}$$

$$T_k^{\mathtt{t}}[s_i][l] = \{s_j \mid (s_i, l, s_j) \in T_k\} \tag{11}$$

$$T_k^{\mathtt{l}}[s_i][s_j] = \{l \mid (s_i, l, s_j) \in T_k\} \tag{12}$$

For any of the three (**s**ource, **t**arget, **l**abel), we then generate the following new primal constraints:

$$\neg x^{t(+1)} \vee \neg y^{t(+1)} \;\; \forall x, y : T_k^{\sigma}[x][y] = \emptyset \;\; \forall \sigma \in \{\mathtt{s}, \mathtt{t}, \mathtt{l}\} \tag{13}$$

We can now omit all primal constraints 7 that are already covered by these new binary primal constraints. That is, we generate $\neg(l^t \wedge s_i^t \wedge s_j^{t+1})$ only if $T_k^s[l][s_j]$, $T_k^t[s_i][l]$, and $T_k^l[s_i][s_j]$ are non-empty. Otherwise a constraint 13 covering (i.e. implying) $\neg(l^t \wedge s_i^t \wedge s_j^{t+1})$ will be generated.

As for the primal constraint 7, we can aggregate the primal constraint 13 along one of the two remaining dimensions of the constraints to a constraint, e.g., $s_i \rightarrow \bigwedge_{T_k^t[s_i][l]=\emptyset} \neg l^t$. We can again dualise these constraints to obtain constraints of the form $s_i \rightarrow \bigvee_{T_k^t[s_i][l]\neq\emptyset} l^t$ – listing in this case the labels that are applicable in the state s_i. These constraints have the same properties as the dual constraints involving all three dimensions (fewer but larger clauses). In total, there are six types of such dual constraints, depending on which dimension is aggregated first and second. We again generate dualised clauses only if there is only one literal on the right-hand side.

In theory, it is possible that the right-hand side of these dualised constraints is empty, allowing to further project these 2D projections into a single dimension. This would reveal information about individual states (that they are dead or unreachable), or labels (being inapplicable). However any such state or label (except init or goal) are pruned automatically in preprocessing, making this further projection useless.

Example 2. *A simple way to visualise how these binary constraints are created is to consider once again the 3D structure T_k and projecting its entries onto planes defined by pairs of dimensions. Each projected entry summarizes the corresponding line of T_k parallel to the remaining axis. This entry is zero exactly when that line contains no valid transitions. Consider the adjacency matrices for l_1 and l_2 and the projection (P) to the (source, target) plane. Here, we use different symbols to represent transitions forbidden by binary constraints from different planes: $\square$ for (label, source), $+$ for (label, target), and $\times$ for (source, target).*

l_1	A'	B'	C'
A	×	1	1
B	0	×	1
C	1	×	×

Table 3: l_1

l_2	A'	B'	C'
A	×	1	+
B	1	×	+
C	□	⊠	⊠

Table 4: l_2

P	A'	B'	C'
A	0	1	1
B	1	0	1
C	1	0	0

Table 5: P

When looking at table 4, we know that row "C" (a line parallel to the "target" axis) and column "C'" (a line parallel to the "source" axis) contain only 0's. As such, they can be encoded with constraints $l_2 \implies \neg C$ and $l_2 \implies \neg C'$.

When looking at table 5, we add the following constraints: $A \implies \neg A'$, $B \implies \neg B'$, $C \implies \neg B'$ and $C \implies \neg C'$.

After adding these constraints, only one 0 remains unblocked in the initial tables, which can be blocked with the following constraint: $l_1 \wedge B \implies \neg A'$. So with this method we were able to encode the information using 6 binary constraints and 1 ternary constraint as opposed to 12 (the total number of 0's in the initial tables) ternary constraints.

4.3 Self-loops

In each factor, many labels may have no effect at all. A label l is *irrelevant* for factor Θ_k iff it has a self-loop transition in every state in Θ_k and no other transitions. All other labels are relevant. Among the relevant labels there are further labels that only have self-loop transitions, but not necessarily in every state. Encoding these labels as outlined so far, creates (with projection optimization) a clause $l^t \wedge s_i^t \implies s_i^{i+1}$ for each state s_i in which l has a self-loop. These constraints are both repetitive and memory-inefficient, given that typically the majority of all labels is irrelevant for an individual factor.

To capture the semantics of self-loop transitions more compactly, we introduce an auxiliary variable SL_k^t:

- SL_k^t: Θ_k performed a self-loop transition at time t.

We assert this semantic with the following constraint:

$$s_i^t \implies (SL_k^t \iff s_i^{t+1}) \qquad \forall s_i \in S_k \tag{14}$$

Let's now consider $L_k^{\rightarrow} = \{l \in L \mid \exists (s, l, t) \in T_k \text{ s.t. } s \neq t\}$, i.e., the set of labels that include non-self-loop transitions in Θ_k, and $L_k^{\circ} = L \setminus L_k^{\rightarrow}$ the labels that are only self-loops. If SL_k^t is false, i.e., the transition was not a self-loop, some label from $L_k^{\rightarrow}$ was selected at timestep t (Eq. 15).

$$\neg SL_k^t \implies \bigvee\nolimits_{l \in L_k^{\rightarrow}} l^t \tag{15}$$

We cannot require that SL_k^t implies a disjunction of $l \in L_k^{\circ}$ as a label in $L_k^{\rightarrow}$ can have both self-loop and non-self-loop transitions. If a self-loop label $l \in L_k^{\circ}$ is executed, constraint 6 ensures that no label in $L_k^{\rightarrow}$ is executed and thus SL_k^t is true. Constraint 14 then enforces the self-loop. For relevant only-self-loop labels (self-loops in some but not all states), we additionally need to ensure that they are executed in a state in which they have a self-loop via (both for time t and $t+1$):

$$l^t \implies \bigvee\nolimits_{(s_i, l, s_i) \in T_k} s_i^{t(+1)} \qquad \forall l \in L_k^{\circ} \tag{16}$$

The transition semantics of labels $l \in L_k^{\rightarrow}$ is enforced as before. Since the semantics of labels $l \in L_k^{\circ}$ is encoded via constraint 16, we omit all constraints otherwise restricting their execution. We remove all primal containing l^t and all dual constraints with l^t on the left-hand side. If an $l \in L_k^{\circ}$ appears on the right-hand side of a projected dual clause, the clause specifies *possible* labels, e.g., in constraints of the type $s_i \implies \bigvee l$. To replace these occurrences, we introduce a new SAT variable.

- NS_k^t: some label in $L_k^{\rightarrow}$ was executed.

We define its semantics using the constraints:

$$NS_k^t \Rightarrow \bigvee_{l \in L_k^{\rightarrow}} l^t \qquad \neg NS_k^t \Rightarrow \bigwedge_{l \in L_k^{\rightarrow}} \neg l^t \tag{17}$$

Note that $\neg NS_k^t$ implies SL_k^t, but not vice versa – as labels may have both self-loop and non-self-loop transitions.

We replace all occurrences of $l \in L_k^{\circ}$ with $\neg NS_k^t$ (and remove duplicates) stating that *some* self-loop can be performed while constraint 16 ensures that the selected self-loop is possible. For dual constraints where the right-hand-side are labels (i.e., source or target implies labels), there are almost always at least two possible labels as almost every state has a self-loop with an irrelevant label. To still enable the utilization of dual clauses in these cases, we generate them only if there is at most one *relevant* label on the right-hand-side. I.e. we may generate clauses of the form $s_i^t \Rightarrow l^t \vee \neg NS_k^t$.

This encoding of self-loops introduces only two extra variables and $2|S_k| + |L_k^{\circ}| + |L_k^{\rightarrow}| + 2$ constraints per transition system instead of $|\{L \backslash L_k^{\rightarrow}\}| \times |S_k|$. Additionally, it lets us eliminate constraint 5. If no label is selected, SL_k^t must be true (constraint 15) and thus the states does not change (constraint 14). This ability is crucial when using more advanced methods for selecting horizon length (Algorithm C, see [Rintanen, 2011] and Sec. 6) for satisficing planning.

Example 3. *Consider the following adjacency matrices for labels l_1 to l_4 where l_1 and l_3 are irrelevant labels, l_2 is composed only of self-loops but is relevant and l_4 contains transitions that are not self-loops.*

l_1,l_3	A'	B'	C'
A	1	0	0
B	0	1	0
C	0	0	1

Table 6: l_1 and l_3

l_2	A'	B'	C'
A	1	0	0
B	0	0	0
C	0	0	1

Table 7: l_2

l_4	A'	B'	C'
A	0	1	0
B	0	0	1
C	0	0	0

Table 8: l_4

Without the self-loop optimisation, we create constraints of the type $l_1 \wedge A \implies A'$ for every state, for labels l_1 and l_3, and for label l_2 we create the same type of constraint for every state where there is a self-loop plus constraints like $l_2 \implies \neg B$ (if using the projection optimisation) for every state where there is not a self-loop. Therefore, the number of clauses to encode self-loop transitions scales with $O(|S_k||L^{\circ}|)$.

With the self-loop optimisation we start by encoding the constraint $A \implies (SL \iff A')$ for every state, for a total of $2|S_k|$ clauses. Then, we assert the preconditions of the relevant labels comprised only of self-loops, which in this case would be the constraint $l_2 \implies A' \vee C'$ adding an extra $|L^{\circ}|$ clauses. Then, we add the constraint enforcing that if SL is false, then the transition executed was not a self-loop accounting for one constraint $\neg SL \implies l_4$. Lastly, we include the constraints using the NS variable. This corresponds to $NS \implies l_4$ for an additional clause and to $\neg NS \implies \neg l_4$ for $|L_k^{\rightarrow}|$ additional clauses. So, we get $O(|S_k| + |L|)$, which is often smaller than the encoding without the optimisation.

4.4 Label Groups

Similar to the optimization for self-loops, we can reduce redundancy in the encoding by exploiting the structure of label groups. Label groups are an optimisation for representing more compactly the transitions of each factor [Sievers, 2018]. This is based on the observation that oftentimes there are many labels that share the same set of transitions within a factor. For example, all irrelevant labels have a self-loop in every state. Therefore, a label group $LG \subseteq L$ is a set of labels such that, for any $l, l' \in LG$, $(s_i, l, s_j) \in T_k \iff (s_i, l', s_j) \in T_k$. The same idea can be translated into the SAT encoding. In each factor, all labels within a given label group share the exact same set of transitions. As a result, encoding identical constraints for each label individually is inefficient. To avoid this duplication, we introduce one variable LG^t per label group for each factor. Then the transition constraints can be encoded using label groups instead of labels. To maintain correctness across factors, we add an additional constraint that ensures a label can only be selected if its corresponding label group is also selected. Let $\mathcal{LG}$ be the set of all label groups in Θ_k. We then add for all $LG \in \mathcal{LG}$ the constraint $LG^t \Leftrightarrow \bigvee_{l \in LG} l^t$.

Example 4. *Consider the adjacency matrices of labels l_1 and l_2 which execute exactly the same transitions under a certain transition system.*

l_1	A'	B'	C'
A	0	1	1
B	0	0	1
C	1	0	0

Table 9: l_1

l_2	A'	B'	C'
A	0	1	1
B	0	0	1
C	1	0	0

Table 10: l_2

Without the label group optimisation, the encoding of the constraints for the transitions of these two labels would be nearly identical, with the only difference in the constraints being the variable of the label, e.g., $l_1 \wedge A \implies \neg A'$ and $l_2 \wedge A \implies \neg A'$.

With the use of label groups, we can reduce all nearly identical constraints by introducing a new variable (LG) and a new constraint for each label group. With this new variable we can just encode the transitions using it, e.g., $LG \wedge A \implies \neg A$ and assert that if a label group is being used, then at least one of its labels was selected: $LG \Leftrightarrow l_1 \vee l_2$.

5 Parallelism

So far, our encoding has been restricted to generating sequential plans due to constraint 6, restricting us to at most one label per timestep. While this ensures correctness, it is highly limiting in practice, as it significantly increases the number of steps required to reach a solution (see e.g. [Kautz and Selman, 1996; Rintanen *et al.*, 2006]). We show how to extend our encoding to support parallelism, i.e., selecting multiple labels simultaneously. Removing constraint 6 without additional restrictions could result in incorrect plans. There would be nothing to prevent labels from, e.g., executing transitions (s_i, l_1, s_j) and (s_i, l_2, s_j) concurrently in one factor, which may be invalid, e.g., if s_j has no outgoing transitions.

We aim for a form of parallelism akin to the ∀-step parallelism for STRIPS and SAS$^+$ tasks [Kautz and Selman, 1996; Rintanen *et al.*, 2006]. In ∀-step parallelism, a set of actions can be applied in parallel if *every* linearisation is executable and leads to the same state. That is, if a_1 and a_2 are ∀-step parallel, then for any state s in which both are applicable, both $\langle a_1, a_2\rangle$ and $\langle a_2, a_1\rangle$ are valid and reach the same state.

We consider two different definitions, one of which is simpler to encode, and one that allows for more parallelism.

5.1 Self-loop Parallelism

Consider transitioning in some factor from state s_i to s_j with a label l. Any label l' that is a self-loop in both s_i and s_j can be safely executed together with l in any order. We allow this parallelism only if l' comprises *only* of self-loop transitions. That is, self-loop parallelism allows to execute at most *one* actual transition in each factor together with a set of self-loop only transitions. To enforce this, we replace constraint 6 with a new set of constraints 18, one per factor. These constraints ensure that at most one label from $L_k^{\rightarrow}$ is selected at each timestep.

$$atMostOne(\{l^t \mid l \in L_k^{\rightarrow}\}) \tag{18}$$

However in the base encoding of the transition relation T_k, selecting two labels to be true requires both to be executed. This makes it impossible to execute a self-loop label l° together with a label $\vec{l}$ performing an actual transition: l° requires source and target state to be the same, while $\vec{l}$ requires them to be different. I.e. parallelism between two labels is only possible if both of them are self-loops in all factors.

Here we leverage the self-loop optimization. It enforces for each executed self-loop label l° only that both the source and the target state have a self-loop with l° (constraint 16). Source and target state must only be identical if no label from $L_k^{\rightarrow}$ is executed. Executing self-loops in parallel with one actual transition is still possible if SL_k^t is false.

Lastly, dualised constraints where the right-hand-side contains labels become critical, as they relied on at most one label being executed at each time. This will be a projected constraint of the from $s_i^{t(+1)} \Rightarrow \bigvee l^t$ with s_i either being a source or target state. Suppose a label $l \in L_k^{\rightarrow}$ would be executed that is impossible, i.e., not mentioned on the right-hand-side of the dualised constraint. Then no other $l' \in L_k^{\rightarrow}$ can be selected for execution due to constraint 18. To satisfy the dualised constraint, the right-hand-side must contain $\neg NS_k^t$ and NS_k^t must be false. But this forces via constraint 17 all non-self-loop-only labels to also be false, i.e. l must be false.

5.2 Chain Parallelism

Self-loop parallelism is by construction a subset of ∀-step parallelism: every set of actions A executable under self-loop parallelism is executable in parallel under ∀-step semantics.

As previously mentioned, labels that contain transitions of the form (s_i, l_1, s_j) and (s_i, l_2, s_j), should not be applied simultaneously. However, if both labels also allow for self-loop transitions at the target s_j (i.e. (s_j, l_1, s_j) and (s_j, l_2, s_j)), then any ordering of the l_1 and l_2 is safe and leads to the same resulting state s_j – with the idea that the label that is executed first transitions to s_j and the other loops there.

This motivates a further notion of parallelism. We say that a set of labels $\mathcal{L}$ is executable under *chains parallelism* when transitioning from state s_i to state s_j, if when we consider the non-self-loop labels $\mathcal{L} \cap L_k^{\rightarrow}$, either $|\mathcal{L} \cap L_k^{\rightarrow}| = 1$ or for all $l \in \mathcal{L} \cap L_k^{\rightarrow}$ the transition (s_j, l, s_j) exists.

We cannot simply allow or forbid a specific set of labels to be executed in parallel, since the validity of their parallel execution is dependent on s_j. Let $\mathcal{L}_j$ be the set of labels which have a transition to state s_j from another state. We need to encode that if some $l \in \mathcal{L}_j$ is executed all other $l' \in \mathcal{L}_j \setminus \{l\}$ have a self-loop at s_j. And thus if at least two are executed, all have a self-loop at s_j. This exclusion is exactly the restriction that Rintanen *et al.*'s [2006] *chains* constraints describe – with $\mathcal{L}_j$ being the erasers and $\{l \mid l \in \mathcal{L}_j \text{ and } (s_j, l, s_j) \notin T_k\}$ being the requirers. For every factor Θ_k and target state s_j we introduce $2|\mathcal{L}_j|$ "helper" variables ($h^i_{k,j}$ and ${h'}^i_{k,j}$) along with constraints 19 to 22.

$$(l_i \wedge s_j) \implies h^i_{k,j} \wedge {h'}^{i-1}_{k,j} \qquad \forall l_i \in \mathcal{L}_j \tag{19}$$

$$(h^i_{k,j} \implies h^{i+1}_{k,j}) \qquad \forall i{:}\,1 \leq i \leq |\mathcal{L}_j| - 2 \tag{20}$$

$$({h'}^i_{k,j} \implies {h'}^{i-1}_{k,j}) \qquad \forall i{:}\,2 \leq i \leq |\mathcal{L}_j| - 1 \tag{21}$$

$$(h^{i-1}_{k,j} \vee {h'}^i_{k,j}) \Longrightarrow \neg l_i \quad \forall l_i \in \mathcal{L}_j \text{ with } (s_j, l_i, s_j) \notin T_k \tag{22}$$

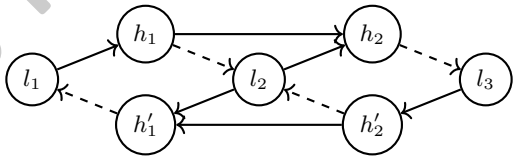

Figure 1: Visual representation of constraints 19 to 21

The core idea is the following: if a label l is selected and leads to state s_j, then every other label that can transition to s_j but lacks a self-loop at s_j must be disabled. Figure 1 gives a visual idea behind the encoding. A normal edge represents an implication from the source to the target and a dashed edge represents an implication from the source to the negated target. Note that the dashed edges should only be present if the target does not contain a self-loop.

In contrast to self-loop parallelism, chains parallelism allows for executing two non-self-loop labels at the same time step. Thus moving from primal to dual constraints (see Sec. 4.1) is not sound any more, i.e., the dual constraints do not forbid all impossible transitions, (see counterexample in the appendix [Filipe *et al.*, 2026b]) but only force that one of the selected labels is a possible transition. We therefore cannot omit primal constraints whenever we generate a dual constraint. As dual constraints provide useful information for unit-propagation, we nevertheless keep them. I.e. for chains parallelism we include both primal and dual constraints and only omit primal constraints 8, if there is a primal constraint 13 covering it.

Example 5. *Consider labels l_1 and l_2 where they both have the transitions $A- > B'$, $A- > C'$ and $C- > C'$, but then*

only l_2 has the transition $B- > B'$.

l_1	A'	B'	C'
A	0	1	1
B	0	0	0
C	0	0	1

Table 11: l_1

l_2	A'	B'	C'
A	0	1	1
B	0	1	0
C	0	0	1

Table 12: l_2

From our encoding, these two labels can only be executed together under ∀-step parallelism if they are moving to state C'. Let's see why: For starters, for both labels to even have the possibility of being executed at the same time, their preconditions have to be satisfied, which means we are either starting at A or at C since l_1 is not executable in B. If we start at A and try to move to B', then the chains for this state would generate the constraints: $l_1 \wedge B' \implies h_1$, $l_2 \wedge B' \implies h'_1$ and $h'_1 \implies \neg l_1$, thus making it impossible to select both l_1 and l_2 while moving to B', which is the intended behaviour since not all linearisations are possible, e.g., it is not possible to execute the labels in the order l_2, l_1 since l_1 does not have a self-loop at B'. Then if we were to move to state C', we would only require the constraints $l_1 \wedge C' \implies h_1$ and $l_2 \wedge C' \implies h'_1$ from the chains corresponding to this state, as the constraint $h'_1 \implies \neg l_1$ would no longer be generated since l_1 has a self-loop at C', thus making it possible to execute both labels in parallel when moving from A to C'. If we were to try to move from C to C' exactly the same constraints would be generated since once again both labels have a self-loop at the target.

This means that in total there are only 4 ways in which these labels can be executed in parallel: $(A, l_1, C'), (C, l_2, C')$, $(A, l_2, C'), (C, l_1, C')$, $(C, l_1, C'), (C, l_2, C')$ and $(C, l_2, C'), (C, l_1, C')$.

Theorem 1. *If a set of labels L is executable in parallel under chains parallelism then they are ∀-step parallel.*

Proof. Suppose L is executable in parallel under chains parallelism. Let $\vec{L}$ be an arbitrary ordering of L and we have to prove that $\vec{L}$ is executable and all linearisiations of L yield the same state. $\vec{L}$ is executable if it is executable for every individual factor. Consider an arbitrary one. Any label $l \in L \cap L^\circ$ does not change the state and can be executed at any position and thus be ignored. Observe that the remaining actions perform overall a single state transition from state s_i to state s_j. In the execution of $\vec{L}$, the first label $l \in \vec{L} \setminus L^\circ$ can (as enforced by the primal and dual constraints) and will perform the transition. Given constraints 19 to 22, either $|L \setminus L^\circ| = 1$ or all other $l' \in \vec{L} \setminus L^\circ$ have a self-loop at s_j allowing them to be executed and that execution leading to s_j. □

5.3 FTS Parallelism vs ∀-step Parallelism

Next, we study the relation of self-loop and chains parallelism with classical ∀-step parallelism on FTS and SAS$^+$ tasks.

Consider how SAS$^+$ tasks are compiled into FTS tasks. In SAS$^+$ tasks actions are specified via preconditions and effects, which are (partial) variable assignments. When translated into FTS, each variable corresponds to a factor (the atomic projection over that variable) and each action corresponds to a label. Thus, the transitions for label l in Θ_k belong to one of four cases, depending of whether the value of the variable k is defined for the preconditions and effects:

- $v_k \notin pre(l), v_k \notin \mathit{eff}(l)$: there is a self-loop on all states in Θ_k
- $v_k \in pre(l)$, $v_k \notin \mathit{eff}(l)$: there is a single self-loop transition $(pre(l)[k], l, pre(l)[k])$
- $v_k \notin pre(l)$, $v_k \in \mathit{eff}(l)$: there are transitions $(s, l, \mathit{eff}(l)[k])$ from each $s \in \Theta_k$
- $v_k \in pre(l)$, $v_k \in \mathit{eff}(l)$: there is a single transition $(pre(l)[k], l, \mathit{eff}(l)[k])$

Self-loop parallelism is provably weaker than regular ∀-step parallelism. Consider an SAS$^+$ problem with two actions a_1 and a_2 so that a_1 has the effects $v_1 = 1$ and $v_2 = 2$, while a_2 has $v_2 = 2$ and $v_3 = 3$. None of the actions have preconditions. a_1 and a_2 can clearly be executed in any order and all such orders lead to the same state. However w.r.t. the variable v_2 none of the two actions is a self-loop and thus only one of them can be executed in parallel in our encoding.

However, chains parallelism is a generalisation of the ∀-step parallelism on SAS$^+$ tasks.

Theorem 2. *Consider a FTS task whose factors are atomic projections of a SAS$^+$ task. Then, if L is executable in parallel under the classical ∀-step parallelism, it is also executable under chains parallelism.*

Proof. Let L be a set of labels executable under ∀-step parallelism. We can again ignore all irrelevant labels and focus on a single factor. We consider transitioning from state s_i to state s_j (if $s_i = s_j$ all executed transitions for that factor are self-loops, trivialising the conditions). Let $l \in L$ be an executed relevant label. There are linearisations of L in which l is the first label and one in which l is the last label. From the latter case, we know that there must be a transition for l ending at s_j (either from s_i or a self-loop). From the former, we know that there is a transition for l that starts at s_i (either a self-loop at s_i or transitioning to s_j). If l is a self-loop in both s_i and s_j, then l is a self-loop in all states if this factor is an atomic projection from a SAS$^+$ problem. Otherwise l must have a transition (s_i, l, s_j). Suppose it does not have the transition (s_j, l, s_j). If this is the only non-self-loop label for the factor, no problem arises. Otherwise, there must be another label l' for which the same reasoning holds. Then both ll' and $l'l$ must be executable and reach the state s_j. Note that it is impossible for l to have the transition (s_i, l, s_i) – as in the SAS$^+$ sense it has an effect moving it to s_j. The same holds for l', thus if executed, l' forces a transition to s_j – as it cannot loop at s_i. Since $l'l$ must be executable due to ∀-step parallelism, l must be able to loop at s_j, thus forcing the transition (s_j, l, s_j), which satisfies the chains-parallelism condition. □

6 Experiments

We implemented our encodings on FTSPlan [Torralba *et al.*, 2023], and ran experiments on all International Planning Competition domains, containing in total 2026 instances. We also ran experiments on the FTS benchmark which contains a

	One-by-One			AlgC		
	Seq	S-L	Chains	Seq	S-L	Chains
Baseline	603	–	–	764	–	–
SL	659	985	1096	1097	1270	1271
SL+LG	668	1001	1116	1014	1221	1271
SL+P	708	**1080**	**1196**	**1219**	**1418**	**1431**
SL+P+LG	**712**	1065	1184	1113	1336	1393

Table 13: Coverage for multiple configurations of formula generation and parallelism (columns) and transition relation optimizations (rows). S-L and Chains require the SL optimisation.

	SAT			FF		MpC	
	Seq	S-L	Chains	–	p.o.	∀	∃
None	979	1236	1288	1318	1615	1368	1431
LR	1095	1338	1344	1324	1596	–	–
LR+S	**1219**	**1418**	**1431**	**1430**	**1627**	–	–
LR+S+M	1039	1241	1210	1314	1537	–	–

Table 14: Coverage using different task transformations.

total of 431 instances. A snapshot of the data and code used is archived at [Filipe *et al.*, 2026a]. The experiments were run using Lab [Seipp *et al.*, 2017] on an AMD EPYC 9654 with a memory limit of 1.75GB and a time limit of 1800 seconds. Our encodings use the Kissat SAT solver [Biere *et al.*, 2024]. Madagascar comes with its own solver specifically tailored to its encoding and inextricably linked to it, as such it was ran with it.

We conducted two sets of experiments using a modified version of Fast Downward [Helmert, 2006] which supports the FTS representation (except for Madagascar) to evaluate our strategies. The first set (Table 13) uses the transformations label reduction (LR) and shrinking (S) and presents the results of the baseline encoding (which encodes transitions only using constraint 7) as well as the results of encodings using projections (P), self-loop (SL) and label grouping (LG) optimisations with different parallelism strategies (Seq, for no parallelism; S-L for Self-loop parallelism; Chains for Chains parallelism) as well as different strategies for formula generation (One-by-One for generating a new formula with a length increased by 1 regarding the previous formula when that one is proved unsatisfiable; AlgC, similar to what is proposed by [Rintanen, 2014]). The second set (Table 14) explores the effect of applying various task transformations on top of the best-performing configuration, allowing us to better understand their impact. For reference, we also include the performance of the state-of-the-art SAT-based planner Madagascar (MpC), which was ran with configurations for ∀ and ∃-step parallelism combined with AlgC formula generations, as well as an adapted version of the FF heuristics [Hoffmann and Nebel, 2001] [Torralba and Sievers, 2019] with lazy-greedy search with and without preferred operators (p.o.). This latter configuration is equivalent to what is commonly known as FTSPlan [Torralba *et al.*, 2023]. Due to the large number of domains, we report only the total number of solved instances. An appendix is available [Filipe *et al.*, 2026b] with detailed results per domain as well as a more in depth analysis of the impact of each individual projection.

Parallelism and Formula Generation Table 13 highlights the performance differences across various encoding strategies. Sequential encodings significantly underperform encodings that support parallelism, solving around 300 fewer instances, since they require more timesteps to find a satisfying assignment. As a new formula is generated for each timestep, this results in more formulas being attempted and, consequently, an increase in required solving time. When comparing the two parallelism strategies, chain-based parallelism outperforms the self-loop-based approach, despite requiring additional variables and clauses. This indicates that the benefit of reducing the number of timesteps, and thus the number of formulas that must be solved, outweighs the cost introduced by the more complex encoding. Regarding formula generation strategies, encodings using the One-by-One approach significantly underperform, regardless of optimisations, compared to AlgC, solving 200–400 fewer instances. This is due to the requirement to prove unsatisfiability for each formula length before proceeding to the next, which becomes increasingly costly for longer plans.

Optimisations The projection optimisation, which reduces the number of constraints and their arity, shows a consistent improvement across configurations, solving 50 to 200 more instances. On the other hand, the label groups optimisation seems to have a negative impact. Aside from one result in One-by-One, the use of this optimisation always reduced the number of instances solved indicating that the introduction of the additional variable as well as the new overhead in the formula ends up being more harmful than beneficial. Regarding the Self-loop optimisation, we only present the baseline results without it because it would not be possible to allow any sort of relevant parallelism without this optimisation as otherwise the base constraint encoding the transitions would be too restrictive.

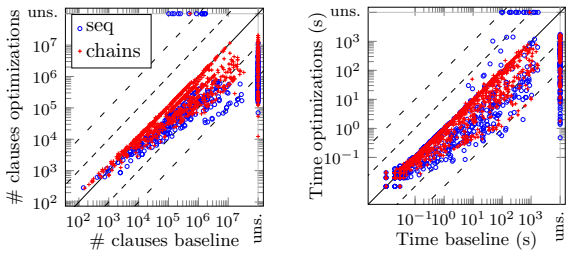

Figure 2: Number of clauses in first satisfiable formula and runtime of one-by-one with LR+S with/without projection optimizations.

Figure 2 shows the difference in formula size (in terms of the number of clauses) and total time with and without using the projection optimisation. Using chains, with the optimisation enabled, the encoding produces less clauses for a total of 533 tasks while only exceeding the number of clauses for 299. Additionally, it can also be seen that the difference in the number of clauses in the cases where it produces less clauses is much bigger than in the cases where it produces more. We do not provide an analysis for the self-loop optimisation because it has to be enabled in order for parallelism to be allowed. We also do not provide a plot to analyse how the

number of variables changes with each optimisation since the number of variables introduced by them is negligible.

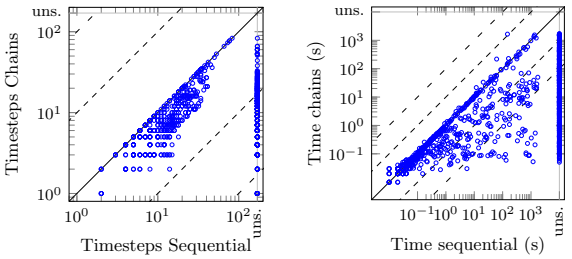

Figure 3: Minimum number of timesteps to find a plan and total time with chains parallelism (y-axis) versus without parallelism (x-axis).

Figure 3 shows the impact of parallelism in more detail. In this case, the main difference is in reducing the number of time steps necessary to find a plan. This translates into a very significant difference in terms of time, sometimes of several orders of magnitude. This is not surprising because the SAT formula grows linearly with the number of time steps, but solving the larger formulas can be exponentially harder.

Transformations When analysing the impact of transformations, it is clear that they can influence performance, both positively and negatively. While our default set of transformations consistently achieves the best results, merge (M) transformations consistently decrease performance, sometimes even falling below the encoding without any transformations. We evaluated the merge strategies available in FTSPlan, namely DFP and MIASM, setting a maximum factor size of 100. MIASM obtained slightly better results than DFP (around 5% higher coverage) and is the configuration reported as M in the paper. Coverage was lower or equal in every single domain compared to encodings not using merging. This drop can be attributed to two key factors.

The first is the loss of parallelism. Merging transition systems can force previously independent labels to be treated as interdependent. For example, imagine two transition systems and two labels: one label is only relevant in the first system, and the other only in the second. After merging, both labels become relevant in the combined system and may even collapse into a single label during label reduction. As a result, they can no longer be executed in parallel, leading to longer plans with more timesteps.

The second issue is the increase in state space. When multiple systems are merged, the combined system may contain many more states than each system individually. This expansion leads to a corresponding increase in the number of possible transitions that must be encoded, thus increasing the overall size and complexity of the formula.

Notably, these merge strategies were originally devised for computing heuristics rather than for problem reformulation. In the former setting, merging typically continues until all variables have been merged, while for the latter setting it is arguably more important to decide if merging is useful at all. Other merge strategies from the literature are likely to run into the same issues for this reason. While setting a size limit, as done in FTSPlan and here, provides a practical safeguard, devising merge strategies specifically tailored to task reformulation remains an open problem and is beyond the scope of this paper.

	Σ	FF –	FF (p.o.)	SAT Chains	SAT Seq	SAT S-L
burnt-pancakes	100	60	63	48	54	55
matrix-mult	11	11	11	11	11	11
cavediving-adl14	20	7	7	8	7	8
pancakes	100	58	66	53	58	63
rubiks-cube	100	9	10	45	47	47
topspin	100	22	22	29	30	28
Σ	431	167	179	194	207	212

Table 15: FF and SAT on FTS Benchmark per Domain. Our encodings use AlgC and SL, and P.

FTS Benchmark Table 15 presents the results of running FF (with and without preferred operators) and our encodings on the FTS Benchmark using label reduction and shrinking. It can be seen that just like in the International Planning Competition Benchmark (IPC), the inclusion of the projections always has a positive impact. However, there are two key differences to the results of the IPC. The first is that now our encodings outperform FF both with and without preferred operators. The second is that the encoding using chains parallelism, achieves the worst results among our encodings. This is due to the fact that in this benchmark there is almost no parallelism allowed, and as such we cannot find significantly shorter plans using the encodings that support parallelism. So, in this scenario the chains encoding mostly only introduces the additional overhead of the extra variables and constraints, without any benefits. Note, however, that the lack of parallelism in this benchmark is not directly related to whether the input task is in SAS^+ or FTS representation, so one should not conclude that paralellism is irrelevant in this setting. This is simply due to it being a relatively small benchmark set compared to the IPC and consisting mostly of permutation puzzles who typically lack parallelism compared to planning tasks where multiple agents/components can perform actions simultaneously.

7 Conclusions

We presented several SAT encodings for Factored Transition Systems (FTS), including techniques leveraging projections, self-loops, label groups, and different forms of parallelism. Empirically, we have shown that our encoding matches the state-of-the-art SAT-based planner Madagascar in total instances solved in the IPC benchmark even though we do not support ∃-step parallelism. In the IPC benchmark, we also outperform FF without preferred operators while in the FTS benchmark, we outperform FF both with and without preferred operators.

For future work, we aim to further relax the constraints on parallelism, to support ∃-step parallelism. Additionally, exploring new merge strategies tailored for SAT encodings could help avoid the problems of existing merging strategies.

Acknowledgements

This publication is part of the project "Exploiting Problem Structures in SAT-based Planning" with file number OCENW.M.22.050 of the research programme Open Competition which is financed by the Dutch Research Council (NWO).

References

[Bäckström and Nebel, 1995] Christer Bäckström and Bernhard Nebel. Complexity results for SAS^+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

[Behnke *et al.*, 2025] Gregor Behnke, David Speck, and Daniel Gnad. Axsat–bringing axioms to sat planning. In *European Conference on Logics in Artificial Intelligence*, pages 77–93. Springer, 2025.

[Bernardini and Muise, 2024] Sara Bernardini and Christian Muise, editors. *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2024)*. AAAI Press, 2024.

[Biere *et al.*, 2024] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL, Gimsatul, IsaSAT and Kissat entering the SAT Competition 2024. In Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions*, volume B-2024-1 of *Department of Computer Science Report Series B*, pages 8–10. University of Helsinki, 2024.

[Büchner *et al.*, 2024] Clemens Büchner, Patrick Ferber, Jendrik Seipp, and Malte Helmert. Abstraction heuristics for factored tasks. In Bernardini and Muise [2024], pages 40–49.

[Domshlak *et al.*, 2009] Carmel Domshlak, Jörg Hoffmann, and Ashish Sabharwal. Friends or foes? On planning as satisfiability and abstract CNF encodings. *Journal of Artificial Intelligence Research*, 36:415–469, 2009.

[Fan *et al.*, 2014] Gaojian Fan, Martin Müller, and Robert Holte. Non-linear merging strategies for merge-and-shrink based on variable interactions. In Stefan Edelkamp and Roman Barták, editors, *Proceedings of the Seventh Annual Symposium on Combinatorial Search (SoCS 2014)*, pages 53–61. AAAI Press, 2014.

[Fikes and Nilsson, 1971] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Filipe *et al.*, 2026a] João Filipe, Álvaro Torralba, and Gregor Behnke. (data and code) transforming and encoding fts for sat solving: What helps, what hurts. https://doi.org/10.5281/zenodo.20444380, 2026.

[Filipe *et al.*, 2026b] João Filipe, Álvaro Torralba, and Gregor Behnke. (extended version) transforming and encoding fts for sat solving: What helps, what hurts. https://arxiv.org/abs/, 2026.

[Froleyks *et al.*, 2019] Nils Christian Froleyks, Tomás Balyo, and Dominik Schreiber. PASAR - planning as satisfiability with abstraction refinement. In Pavel Surynek and William Yeoh, editors, *Proceedings of the 12th Annual Symposium on Combinatorial Search (SoCS 2019)*, pages 70–78. AAAI Press, 2019.

[Helmert *et al.*, 2014] Malte Helmert, Patrik Haslum, Jörg Hoffmann, and Raz Nissim. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the ACM*, 61(3):16:1–63, 2014.

[Helmert, 2006] Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[Hoffmann and Nebel, 2001] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[Hoffmann *et al.*, 2014] Jörg Hoffmann, Peter Kissmann, and Álvaro Torralba. "Distance"? Who cares? Tailoring merge-and-shrink heuristics to detect unsolvability. In Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan, editors, *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014)*, pages 441–446. IOS Press, 2014.

[Huang *et al.*, 2012] Ruoyun Huang, Yixin Chen, and Weixiong Zhang. SAS^+ planning as satisfiability. *Journal of Artificial Intelligence Research*, 43:293–328, 2012.

[Kautz and Selman, 1992] Henry Kautz and Bart Selman. Planning as satisfiability. In Bernd Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 1992)*, pages 359–363. John Wiley and Sons, 1992.

[Kautz and Selman, 1996] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI 1996)*, pages 1194–1201. AAAI Press, 1996.

[Kautz *et al.*, 2006] Henry Kautz, Bart Selman, and Jörg Hoffmann. SatPlan: Planning as satisfiability. In *Fifth International Planning Competition (IPC-5): Planner Abstracts*, 2006.

[Korf, 1997] Richard E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI 1997)*, pages 700–705. AAAI Press, 1997.

[Nebel, 2000] Bernhard Nebel. On the compilability and expressive power of propositional planning formalisms. *Journal of Artificial Intelligence Research*, 12:271–315, 2000.

[Rintanen *et al.*, 2006] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12–13):1031–1080, 2006.

[Rintanen, 2011] Jussi Rintanen. Madagascar: Scalable planning with SAT. In *IPC 2011 Planner Abstracts*, pages 61–64, 2011.

[Rintanen, 2012] Jussi Rintanen. Planning as satisfiability: Heuristics. *Artificial Intelligence*, 193:45–86, 2012.

[Rintanen, 2014] Jussi Rintanen. Madagascar: Scalable planning with SAT. In *Eighth International Planning Competition (IPC-8): Planner Abstracts*, pages 66–70, 2014.

[Robinson *et al.*, 2009] Nathan Robinson, Charles Gretton, Duc Nghia Pham, and Abdul Sattar. SAT-based parallel planning using a split representation of actions. In Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 281–288. AAAI Press, 2009.

[Seipp *et al.*, 2017] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab. https://doi.org/10.5281/zenodo.790461, 2017.

[Sievers and Helmert, 2021] Silvan Sievers and Malte Helmert. Merge-and-shrink: A compositional theory of transformations of factored transition systems. *Journal of Artificial Intelligence Research*, 71:781–883, 2021.

[Sievers *et al.*, 2014] Silvan Sievers, Martin Wehrle, and Malte Helmert. Generalized label reduction for merge-and-shrink heuristics. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2014)*, pages 2358–2366. AAAI Press, 2014.

[Sievers *et al.*, 2016] Silvan Sievers, Martin Wehrle, and Malte Helmert. An analysis of merge strategies for merge-and-shrink heuristics. In Amanda Coles, Andrew Coles, Stefan Edelkamp, Daniele Magazzeni, and Scott Sanner, editors, *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2016)*, pages 294–298. AAAI Press, 2016.

[Sievers *et al.*, 2024] Silvan Sievers, Thomas Keller, and Gabriele Röger. Merging or computing saturated cost partitionings? a merge strategy for the merge-and-shrink framework. In Bernardini and Muise [2024], pages 541–545.

[Sievers, 2018] Silvan Sievers. Merge-and-shrink heuristics for classical planning: Efficient implementation and partial abstractions. In Vadim Bulitko and Sabine Storandt, editors, *Proceedings of the 11th Annual Symposium on Combinatorial Search (SoCS 2018)*, pages 90–98. AAAI Press, 2018.

[Sinz, 2005] Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constrai nts. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP 2005)*, volume 3709, pages 827–831. Springer, 2005.

[Speck *et al.*, 2023] David Speck, Paul Höft, Daniel Gnad, and Jendrik Seipp. Finding matrix multiplication algorithms with classical planning. In Sven Koenig, Roni Stern, and Mauro Vallati, editors, *Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling (ICAPS 2023)*, pages 411–416. AAAI Press, 2023.

[Torralba and Sievers, 2019] Álvaro Torralba and Silvan Sievers. Merge-and-shrink task reformulation for classical planning. In Sarit Kraus, editor, *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*, pages 5644–5652. IJCAI, 2019.

[Torralba *et al.*, 2017] Álvaro Torralba, Vidal Alcázar, Peter Kissmann, and Stefan Edelkamp. Efficient symbolic search for cost-optimal planning. *Artificial Intelligence*, 242:52–79, 2017.

[Torralba *et al.*, 2023] Álvaro Torralba, Silvan Sievers, Rasmus G. Tollund, and Kristian Ø. Nielsen. FTSPlan: Task reformulation via merge-and-shrink. In *Tenth International Planning Competition (IPC-10): Planner Abstracts*, 2023.