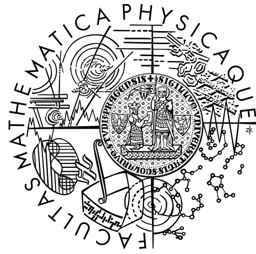

FACULTY OF MATHEMATICS AND PHYSICS
Charles University

MASTER THESIS

Kristýna Pantůčková

Compilation-based Approaches for Automated Planning

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	prof. RNDr. Roman Barták, Ph.D.
Study programme:	Computer Science
Study branch:	Artificial Intelligence

Prague 2020

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I dedicate this thesis to my supervisor for his guidance and to my family for their support.

Title: Compilation-based Approaches for Automated Planning

Author: Kristýna Pantůčková

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: prof. RNDr. Roman Barták, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: One of the possible approaches to automated planning is compilation to satisfiability or constraint satisfaction. Compilation enables to take advantage of the advancement of SAT or CSP solvers. In this thesis, we implement three of the encodings recently proposed for compilation of planning problems: the model TCPP, the $R^2\exists$-Step encoding and the Reinforced Encoding. All these approaches search for parallel plans; however, since they use different definitions of parallel step and different variables and constraints, we decided to compare their performance on standard benchmarks from international planning competitions. As the $R^2\exists$-Step encoding was not suitable for our implementation, we present a modified version of this encoding with a reduced number of variables and constraints. We also demonstrate how different definitions of parallel step in the Reinforced Encoding affect the performance. Furthermore, we suggest redundant constraints extending these encodings. Although they did not prove to be beneficial in general, they could slightly improve the performance on some benchmarks, especially in the $R^2\exists$-Step encoding.

Keywords: planning compilation CSP SAT

Contents

Introduction **3**

1 Background on planning **4**
1.1 Classical (STRIPS) planning . . . 4
1.1.1 Representation of states . . . 4
1.1.2 Classical planning problem . . . 4
1.1.2.1 Example planning problem . . . 6
1.1.3 Parallel plans . . . 7
1.2 SAT . . . 8
1.3 CSP . . . 8
1.4 Encoding of a planning problem . . . 9

2 Related works **11**
2.1 Compilation to CSP . . . 11
2.2 Compilation to SAT . . . 13
2.3 Other approaches . . . 15
2.4 Implemented models . . . 16

3 Implementation of planning models **18**
3.1 Transition Constraints for Parallel Planning . . . 18
3.1.1 Transition constraints . . . 19
3.1.2 Parallelism variables . . . 21
3.1.3 Action decoding . . . 24
3.1.4 Note to implementation regarding width of tables . . . 24
3.1.5 Redundant constraints . . . 26
3.1.5.1 Mutexes produced by the Fast Downward translator 26
3.1.5.2 Transition constraints over a longer time horizon 27
3.1.5.3 GraphPlan-like propositional mutexes . . . 27
3.2 Relaxing the Relaxed Exist-Step Parallel Planning Semantics . . . 31
3.2.1 Implementation . . . 33
3.2.1.1 Literal implementation . . . 33
3.2.1.2 Replacing chain constraints by table constraints . 34
3.2.1.3 Describing value transitions by automata . . . 36
3.2.1.3.1 Note to implementation regarding width of transition matrices . . . 40
3.2.2 Ranking . . . 40
3.2.3 Redundant constraints . . . 42
3.2.3.1 Mutexes produced by the Fast Downward translator 43
3.2.3.2 Actions disabling values of state variables . . . 43
3.3 Reinforced Encoding . . . 44
3.3.1 Implementation . . . 44
3.3.1.1 Literal implementation . . . 44
3.3.1.2 Implementation with table and other Picat constraints . . . 45
3.3.2 Modification of action conflict constraints . . . 46

3.3.2.1 Note to implementation regarding the length of a sum . . . 47
3.3.3 Redundant constraints . . . 47
3.3.3.1 Mutexes produced by the Fast Downward translator 47
3.3.3.2 Transition sequencing constraints . . . 48

4 Experimental evaluation 50
4.1 TCPP . . . 51
4.1.1 Implementation . . . 51
4.1.2 Redundant constraints . . . 52
4.1.3 Summary . . . 52
4.2 $R^2\exists$-Step encoding . . . 55
4.2.1 Implementation . . . 55
4.2.2 Ranking . . . 55
4.2.3 Redundant constraints . . . 57
4.2.4 Summary . . . 60
4.3 Reinforced Encoding . . . 60
4.3.1 Implementation . . . 61
4.3.2 Encoding of conflict actions . . . 61
4.3.3 Redundant constraints . . . 63
4.3.4 Precomputing of the initial makespan . . . 65
4.3.5 Summary . . . 66
4.4 Comparison of models . . . 67

Conclusion 77

Bibliography 78

A Documentation on Picat files 83
A.1 Translation from PDDL . . . 83
A.2 Translation from SAS and solving . . . 84
A.2.1 TCPP . . . 85
A.2.2 $R^2\exists$-Step encoding . . . 86
A.2.3 Reinforced Encoding . . . 88

B Electronic attachments 90
B.1 Benchmarks . . . 90
B.2 Source codes . . . 90
B.3 Declaration . . . 90

Introduction

This thesis focuses on solving planning tasks by compilation to a constraint satisfaction problem (CSP) or satisfiability problem (SAT). Planning is a process of selecting and arranging available actions to reach the desired state of the world from the initial state. The motivation for automated planning includes designing efficient planning systems working with a high number of actions (for instance for solving complicated tasks with a large number of participants, such as rescue missions after natural disasters) and integrating planning as a part of deliberate reasoning of autonomous artificial agents, such as satellites or aircraft (Ghallab et al. [2004]).

Traditional planning approaches include state-space search (forward or backward search), plan-space search and planning with a planning-graph (introduced by Blum and Furst [1995]). In the thesis, we solve planning problems by compilation, where we translate planning tasks to SAT or CSP models and then use a corresponding solver to find a plan. By compilation of planning problems instead of using specialized planning algorithms, we can take advantage of the progress of SAT or CSP solvers, including new powerful reasoning algorithms and heuristics (Kautz and Selman [1999]).

In this thesis, we implement some of the recent compilation-based planning approaches and we compare their efficiency on the standard benchmarks from international planning competitions (IPC). Additionally, we suggest modifications of the implementation and we extend the models by redundant constraints.

In chapter 1, we explain the terms used in the thesis and we provide the background on planning. In chapter 2, we give an overview of related works. In chapter 3, we describe the implementation of the selected planning models: TCPP proposed by Ghooshchi et al. [2017], $R^2\exists$-Step encoding proposed by Balyo [2013] and Reinforced encoding proposed by Balyo et al. [2015]. Experimental evaluation of the implemented encodings and their modifications on IPC planning domains is presented in chapter 4.

1. Background on planning

This chapter explains the terms referenced in the thesis.

1.1 Classical (STRIPS) planning

The goal of planning is to select and arrange available actions to reach the goal from the initial state of the world. States of the world can be specified by predicates that hold in the state or by values of state variables. We describe various types of state representation in subsection 1.1.1. Actions are characterized by preconditions describing the states in which the action can be executed and effects describing the state after executing the action.

1.1.1 Representation of states

A classical planning problem can be described by the set-theoretic representation, classical representation or state-variable representation (Ghallab et al. [2004]). In each of these representations, actions are defined by preconditions, the statements that must be true in the system to execute an action, and effects, the statements that will be true after executing an action.

In the **set-theoretic representation**, each state of a system is described by the set of propositions which are true in the state. Preconditions of each action contain the propositions required to execute the action. Effects describe which propositions will be added and which propositions will be removed from the state description after execution of the action. If the truth value of a proposition changes by executing actions (for example a proposition indicating the current location of a robot), we call this proposition **fluent**. A predicate that does not change its truth value (for example a proposition that states that two cells are adjacent in a grid) is called **rigid**. The **classical representation** uses first-order logic instead of propositions for fluents. Uninstantiated actions are called **operators**.

In the **state-variable representation**, we describe the world by a finite set of state variables. Each state of the world is characterized by values of these variables. Preconditions of an action contain the required values of a subset of state variables and effects set new values of some state variables. In all of the implemented models described in chapter 3, we use the state-variable representation with multi-valued state variables with finite domains.

1.1.2 Classical planning problem

Planning problem specifies the initial state, the goal and available actions. Formally, a **classical planning problem** (Ghallab et al. [2004]) is a triple $P = (\Sigma, s_0, g)$:

- $\Sigma = (S, A, \gamma)$ is a restricted state-transition system or a classical planning domain (Ghallab et al. [2016]):
  - S is a finite set of all possible states of the world

- A is a finite set of all possible actions that can change the state of the world
- γ: $S \times A \rightarrow S$ is a partial deterministic state-transition function, where $\gamma(s,\ a)$ is defined for the state s and action a if and only if a is **applicable** in s; state transitions are instantaneous (implicit time is assumed)
- the system Σ is fully observable
- the system Σ is static (the state of the system does not change until an action is executed)

- s_0 is the initial state of the world
- g is the restricted goal, the definition of a goal state of the system (each goal state satisfies g); extended goals, such as states that must be avoided on the path from the initial to the goal state or other conditions on the path, are not allowed in classical planning

In the state-variable representation that we use in this thesis, a planning problem is described by a vector of state variables $< V_1, ..., V_n >$. **Domain** of a state variable ($Domain(V_i)$) is the set of values that can be assigned to the variable. Each state is characterized by a vector of values of state variables $< v_1, ..., v_n >$ where $\forall i : 1 \leq i \leq n : v_i \in Domain(V_i)$. In the state characterized by the vector $< v_1, ..., v_n >$, it holds $\forall i : 1 \leq i \leq n : V_i = v_i$. The **initial state** s_0 is characterized by a vector of initial values of state variables $< v_1^0, ..., v_n^0 >$ where $\forall i : 1 \leq i \leq n : v_i^0 \in Domain(V_i)$. The **goal** is characterized by assignments of goal values to a subset of state variables $\{V_{k_1} = v_{k_1}^g, ..., V_{k_m} = v_{k_m}^g\}$ where $\forall i : 1 \leq i \leq m \leq n : v_{k_i}^g \in Domain(V_{k_i})$. Each state satisfying the goal assignments is a goal state.

An **action** is defined by a set of preconditions and a set of effects. Preconditions are a set of assignments of values to a subset of state variables which must hold in the state in order to execute the action. Effects are a set of assignments of values to a subset of state variables which will hold in the state after execution of the actions. For the state $< v_1, ..., v_n >$ and the action a, the **state transition function** is defined as follows: $\gamma(< v_1, ..., v_n >, a) =< w_1, ..., w_n >$, where:

- For each variable V_i whose value is defined in both preconditions and effects of a, v_i corresponds to the value in preconditions of a and w_i corresponds to the value in effects of a.
- For each variable V_i whose value is defined only in preconditions of a and not in effects, v_i corresponds to the value in preconditions and $w_i = v_i$ since preconditions must be preserved during the execution of a.
- For each variable V_i whose value is defined only in effects of a and not in preconditions, w_i corresponds to the value in effects of a and the transition with the action a is defined for each $v_i \in Domain(V_i)$.
- For each variable V_i whose value is defined neither in preconditions nor in effects of a, $w_i = v_i$ and the transition with the action a is defined for each $v_i \in Domain(V_i)$.

A solution to a classical planning problem is a **sequential plan**, a linearly ordered finite sequence of actions $\pi = <a_1, ..., a_n>$ where $\gamma(s_0, \pi) = \gamma(\gamma(... (\gamma(s_0, a_1), a_2), ...), a_n)$ satisfies g. Planning (searching for a plan) is off-line; a planner assumes that no changes occur in the system Σ during planning.

1.1.2.1 Example planning problem

As an example of a state-variable representation of a planning problem, here follows a possible state-variable representation of the problem *s2-0* from the planning domain *miconic* from standard benchmarks from international planning competitions, which simulates elevator traffic. The goal of the planning problem is to transport two passengers from their initial floors to their target floors by one elevator:

- Variables:

  rigid: *above(floor0, floor1), above(floor0, floor2), above(floor0, floor3), above(floor1, floor2), above(floor1, floor3), above(floor2, floor3), origin(passenger1, floor3), origin(passenger2, floor1), destination(passenger1, floor2), destination(passenger2, floor3)*; in the state variable representation, rigid relations are defined as predicates as in the other representations

  fluent: *lift_location* with the domain {*floor0, floor1, floor2, floor3*}, *boarded(passenger1)* with the domain {*true, false*}, *boarded(passenger2)* with the domain {*true, false*}, *served(passenger1)* with the domain {*true, false*}, *served(passenger2)* with the domain {*true, false*}

- Operators:
  - *up(x, y)*

    preconditions: *lift_location = x, above(x, y)*

    effects: *lift_location = y*
  - *down(x, y)*

    preconditions: *lift_location = x, above(y, x)*

    effects: *lift_location = y*
  - *board(passenger, floor)*

    preconditions: *origin(passenger, floor), lift_location = floor*

    effects: *boarded(passenger) = true*
  - *depart(passenger, floor)*

    preconditions: *lift_location = floor, boarded(passenger) = true, destination(passenger, floor)*

    effects: *boarded(passenger) = false, served(passenger) = true*

- Initial state: *lift_location = floor0, boarded(passenger1) = false, boarded(passenger2) = false, served(passenger1) = false, served(passenger2) = false*

- Goal: *served(passenger1) = true, served(passenger2) = true*

- One of the possible solutions (plans) is the sequence of actions <*up(floor0, floor1), board(passenger2, floor1), up(floor1, floor3), depart(passenger2, floor3), board(passenger1, floor3), down(floor3, floor2), depart(passenger1, floor2)* >

1.1.3 Parallel plans

Some planning approaches described in chapter 2 search for parallel plans, which contain more actions in one parallel step. A solution to a planning problem, a sequential plan (as described in subsection 1.1.2), can be obtained from the parallel plan by ordering actions in each parallel step. Parallel planners use different parallel semantics. In the **∀-step semantics**, each ordering of actions must be valid in each parallel step.

Formally, a **parallel plan** for a planning problem P respecting the ∀-step semantics (Ghooshchi et al. [2017]) is a sequence of sets of actions $\Pi = \ < A_1,\ ...,\ A_m >$ such that for each $t : 1 \leq t \leq m$ there exists a state s_t such that for each permutation *A'* of the action set A_t: $\gamma(s_{t-1},\ A') = s_t$, where s_0 is the initial state and s_m is a goal state. We call the length of a parallel plan (m) a **makespan**. Some planners use the **∃-step semantics**, where at least one valid action ordering must exist for each parallel step. Different versions of the ∃-step semantics are described in chapter 2. A parallel plan is also called a **layered plan**, where each action set A_t is called a layer (Barták [2011a]). While sequential planners search for a plan with the optimal sequential plan length (the number of actions in the plan) parallel planners search for a plan with the optimal makespan. As a result, parallel plans often contain redundant actions, which are not necessary for reaching the goal (Ghooshchi et al. [2017]).

For the example problem from subsubsection 1.1.2.1, a possible parallel plan with $makespan = 6$ is <{*up(floor0, floor1)*}, {*board(passenger2, floor1)*}, {*up(floor1, floor3)*}, {*depart(passenger2, floor3), board(passenger1, floor3)*}, {*down(floor3, floor2)*}, {*depart(passenger1, floor2)*} >, where the makespan is lower than the length of the sequential plan. Actions *depart(passenger2, floor3)* and *board(passenger1, floor3)* can be together in one parallel step because they can be executed in any order.

A valid parallel plan respecting the ∀-step semantics can be ensured by selecting only independent actions in each parallel step. Two actions are **independent** if neither preconditions nor effects of the first action contain any variable occurring in preconditions or effects of the second action. Formally, actions a and b are independent if $(V_{pre(a)} \cup V_{eff(a)}) \cap (V_{pre(b)} \cup V_{eff(b)}) = \emptyset$, where $V_{pre(a)}$ is the set of variables occurring in preconditions of an action a and $V_{eff(a)}$ is the set of variables occurring in effects of an action a (Balyo et al. [2015]). However, some actions can be executed in any order even though they are not independent, for instance actions sharing the same variable with the same value only in preconditions or actions sharing the same variable with the same value only in effects. Some planners with the ∀-step semantics use a different definition of a parallel step (Ghooshchi et al. [2017]) where they forbid to select a set of actions in one parallel step only if they are in conflict with the ∀-step semantics.

On the other hand, when two actions cannot be executed in the same parallel step or some values of two variables cannot occur in the same step, we say

that they are mutually exclusive (**mutex**). In some of the compilation-based approaches described in chapter 2 and also in the implemented models described in chapter 3, the encodings are extended by mutexes in an attempt to improve the performance since adding redundant information can reduce solving times by saving the time needed to deduce it.

1.2 SAT

A **Boolean satisfiability problem** (Wikipedia contributors [2020c]) is a formula consisting of Boolean variables. A Boolean variable can have a value *true* or *false*. For a Boolean variable x, we define a literal as either x or a negation of x ($\neg x$). A formula is constructed by connecting literals by the binary operators conjunction ($\wedge$) and disjunction ($\vee$). For literals x and y, the expression $x \wedge y$ is true if and only if both literals are true. The expression $x \vee y$ is true if and only if at least one of these literals is true. A disjunction of literals is called a clause. SAT solvers work with Boolean satisfiability problems in the conjunctive normal form (CNF) where a formula in CNF is a conjunction of clauses. Each Boolean satisfiability formula can be converted to an equivalent formula in CNF (Wikipedia contributors [2020a]). A SAT solver finds an assignment of values (*true/false*) to variables such that the formula is satisfied, or proves that the formula is not satisfiable.

For example, $(x_1 \vee \neg x_2) \wedge x_3$ is a Boolean satisfiability formula in CNF, where $(x_1 \vee \neg x_2)$ is a clause and x_3 is a unit clause. The formula consists of the literals $x_1, \neg x_2, x_3$, where x_1, x_2, x_3 are Boolean variables. The formula can be satisfied by any of these assignments: $\{x_1 = true, x_2 = true, x_3 = true\}$, $\{x_1 = true, x_2 = false, x_3 = true\}$ or $\{x_1 = false, x_2 = true, x_3 = true\}$. The formula $(x_1 \vee x_2) \wedge \neg x_1 \wedge \neg x_2$ is an example of a formula that is not satisfiable by any assignment.

1.3 CSP

A **constraint satisfaction problem** (Wikipedia contributors [2020b]) consists of multi-valued variables with finite domains, whose relations are described by constraints, which can be more complex than operators in SAT. A CSP solver assigns to all variables values from their domains such that all constraints are satisfied or proves that such assignment does not exist.

Here follows a simple example of a constraint satisfaction problem:

- Variables: x_1, x_2
- Domains: $Domain(x_1) = \{1, 2\}$, $Domain(x_2) = \{1, 2, 3\}$
- Constraints: $x_1 \neq x_2$, $x_1 + x_2 > 3 \vee x_1 > x_2$
- Possible solutions: $\{x_1 = 1, x_2 = 3\}$, $\{x_1 = 2, x_2 = 3\}$, $\{x_1 = 2, x_2 = 1\}$

CSP solvers provide constraints describing relations between more variables, which can have an efficient implementation. One of the most useful constraints, used in some CSP models described in chapter 2 and also in our implementation,

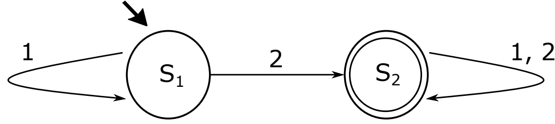

Figure 1.1: Let us assume a regular automaton shown in the picture. S_1 is the initial state, S_2 is the accepting state. Transitions are represented by directed edges labelled by letters from the input alphabet of the automaton. This automaton accepts all words that consist only of the letters 1 and 2 and contain at least one 2 (reading a letter different from 1 and 2 would lead to an error state). For example, if we use the constraint representing this automaton to interconnect the variables in the vector $<x_1, x_2, x_3>$ where $Domain(x_1) = Domain(x_2) = Domain(x_3) = \{1, 2\}$, all possible assignments except for the assignment $\{x_1 = 1, x_2 = 1, x_3 = 1\}$ would satisfy this constraint.

is a **table constraints** or tabling constraint (see Zhou and Fruhman [2020]). A table constraint describes possible values of a tuple of variables. A positive table constraint enumerates all allowed tuples of values while a negative table constraint enumerates forbidden tuples of values. Table 1.1 contains an example positive and negative table constraint.

x_1	x_2	x_3
1	1	2
1	2	1
2	1	1
2	2	1
2	1	2
1	2	2

x_1	x_2	x_3
1	1	1
2	2	2

Table 1.1: Let us assume a positive table constraint shown in the first table, which enumerates all possible values of the variables x_1, x_2, x_3 such that $Domain(x_1) = Domain(x_2) = Domain(x_3) = \{1, 2\}$ and exactly two variables have the same value. The second table is a negative table constraint enumerating all incompatible values. These tables are different representations of the same constraint.

Besides table constraints, we used the constraint modelling a **regular automaton** in our implementation described in chapter 3. In this constraint, we specify states of a regular automaton, we select the initial state and the accepting states and we create a transition matrix. The input word is a list of the variables that we want to interconnect by this constraint. The variables satisfy this constraint if the input word is accepted by the automaton. Figure 1.1 contains an example of this constraint.

1.4 Encoding of a planning problem

To translate a planning task to a satisfiability problem or a constraint satisfaction problem, we encode variables describing states in the planning problem, actions and any auxiliary information in each step of the plan by Boolean or multi-valued variables and we describe preconditions and effects of actions and

other relations by available constraints or operators. Since the problem must be described by a fixed number of variables and constraints, we encode the problem for a fixed length of a plan where we will have one copy of each variable and each constraint for each step of the plan. If the solver proves that a plan of this length does not exist, we encode the problem again with an increased plan length until the solver finds a solution.

2. Related works

This chapter describes several compilation-based approaches to automated planning.

2.1 Compilation to CSP

Do and Kambhampati [2000] used constraint satisfaction programming (CSP) to solve planning problems by compiling the planning graph from the planner Graphplan (Blum and Furst [1995]) to CSP constraints. Graphplan describes a plan by a planning graph consisting of proposition and action layers with edges describing the relations between actions and propositions. The Graphplan algorithm interleaves two phases. The forward phase adds a new proposition and action layer to the planning graph and the backward phase attempts to extract a valid plan from the graph. The planner GP-CSP encodes the planning graph in the backward phase to a constraint satisfaction problem and uses a CSP solver to extract the plan. In comparison to Graphplan, GP-CSP is not limited to directional search. The solver can utilize CSP search techniques, including non-directional search, different variable ordering algorithms, arc consistency and other methods improving search efficiency. CSP encoding also enables to express the structure of a planning problem better, which can be used to develop directed partial consistency algorithms to further accelerate the search. They also discuss advantages of CSP encodings over SAT encodings (as used in the Blackbox system proposed by Kautz and Selman [1999]). While SAT encodings can have excessive memory requirements, GP-CSP encoding of a problem is usually smaller. GP-CSP proved to be faster than both Graphplan and Blackbox on many problems.

Lopez and Bacchus [2003] focused on encoding planning problems directly to CSP. The encoding is built incrementally for increasing plan length until a valid plan is found. This approach enables to use CSP search techniques to a greater extend. They showed how compilation to CSP allows to extend planning problems by time and resources. Their encoding represents parallel plans where actions can be executed simultaneously if they are not interfering according to the definition from Graphplan. They also used various redundant constraints including propositional and action mutex constraints from Graphplan and other enhancements to improve efficiency of search. They demonstrated that their planner CSP-PLAN was more efficient than both Graphplan (Blum and Furst [1995]) and GP-CSP (Do and Kambhampati [2000]) on most of the selected problems from planning benchmarks. In comparison to Blackbox (Kautz and Selman [1998]) and the planner IPP (Koehler et al. [1997]), which also constructs a planning graph and extracts a plan from the graph directly, similarly to Graphplan, CSP-PLAN was slower on some benchmarks due to various heuristics implemented in these planners. However, they suggest enhancements which could possibly improve the efficiency of CSP-PLAN in the future.

Barták and Toropila [2008] reformulated CSP encodings of planning problems, the Straightforward model (Ghallab et al. [2004]), a model based on GP-CSP (Do and Kambhampati [2000]) and a model based on CSP-PLAN (Lopez and Bacchus

[2003]), by replacing logical constraints by table constraints. They showed that the reformulated versions of all three models were faster and the reformulated CSP-PLAN was the fastest, which they explained by the smallest number of constraints and the strongest propagation due to the types of constraints used in this model. They demonstrated how the efficiency of a planning model can be improved by a careful selection of constraints.

Barták and Toropila [2010] proposed several methods for enhancing CSP planning models by search modifications such as lifting, symmetry breaking, singleton arc consistency, nogoods learning and using a dual model inspired by the plan-space planning. They used the reformulated CSP-PLAN (Barták and Toropila [2008]) as a base model and compared it with enhanced models created by incrementally applying lifting, symmetry breaking, singleton arc consistency and nogoods learning. In the experimental evaluation, models with more enhancements were more efficient for most problems, especially for more difficult problems. The efficiency of the model with all four enhancements was further improved on difficult problems by adding the dual model constraints.

Barták [2011a] proposed a novel representation of multi-valued state variables where values of variables are represented by state transition diagrams (domain transition graphs). The solution of a planning problem is defined as synchronized paths in all transition diagrams. Their parallel planning model PaP assigns to each state variable in each time step the action modifying its value (including no-op actions, which do not change the value of a variable); therefore, only independent actions can be executed simultaneously. The constraint model, containing also redundant constraints to improve the propagation, describes the relations between state variables and actions by tabling constraints. They compared PaP with SeP (Barták and Toropila [2010]), a sequential planner improved by symmetry breaking, singleton consistency, nogoods learning, lifting, and techniques inspired by the plan-space planning. On most of the selected benchmark domains, PaP was faster. PaP also outperformed the planner Constance (Gregory et al. [2010]), which improved constraint satisfaction planning by discovering action macros (sequences of actions that compose together), on the domains where Constance outperformed SeP.

Barták [2011b] improved the PaP model by removing state variables. Their model PaP-2 contains only action variables and table constraints describing their relations. Based on the number of problems solved within a given time limit, PaP-2 outperformed PaP (Barták [2011a]) on all of the selected domains and SeP (Barták and Toropila [2009]) on all of the domains except for one, where SeP was more efficient due to the implemented nogood learning. However, PaP-2 was slower than PaP on some of the simpler problems due to the additional time consumed by generating and propagation the new constraints. Similarly to PaP, PaP-2 outperformed the sequential planner Constance (Gregory et al. [2010]) on the domains where Constance outperformed the sequential planner SeP.

Ghooshchi et al. [2017] presented the model Transition Constraints for Parallel Planning (TCPP) inspired by PaP-2 (Barták [2011b]). Similarly to PaP and PaP-2, the constraint model describes transitions in the domain transition graphs by table constrains. Unlike PaP-2, their encoding does not contain variables for actions, only state variables. They compared TCPP with PaP-2 and PaPR, the model PaP-2 reimplemented for the same constraint solver which was

used for TCPP, and other planners from the deterministic track of the International planning competition 2008 (IPC-2008): a cost optimal planner SymBA*-2 using a bidirectional A* search, a state-space search based planner YAHSP3 and a SAT-based planner Madagascar. Based on the total number of solved problems, the planners from IPC-2008 significantly outperformed the constraint-based planners TCPP, PaPR and PaP-2. In most domains, TCPP outperformed PaPR and PaPR outperformed PaP-2.

2.2 Compilation to SAT

Kautz et al. [1992] proposed an encoding for solving sequential planning problems using SAT solvers as a more flexible alternative to deduction and demonstrated how the performance can be improved by adding axioms.

Huang et al. [2010] proposed a new translation of planning problems to SAT based on the SAS+ formalism (Bäckström and Nebel [1995]) reducing the number of clauses in comparison with the STRIPS formalism (Fikes and Nilsson [1971]). Their parallel model SASE consists of action and transition variables (describing transitions of values of state variables) and constraints describing relations between actions and transitions. They proved that in comparison to STRIPS-based encodings, SASE has a smaller search space. They compared SASE to SatPlan06 (Kautz et al. [2006]), an updated version of the encoding proposed by Kautz et al. [1992], and SatPlan06^L, SatPlan06 enhanced by special types of mutex constraints (Chen et al. [2009]). In most domains, SASE outperformed both SatPlan06 and SatPlan06^L. In most cases, SASE also consumed less memory.

Rintanen et al. [2006] focused on decreasing the number of SAT solver calls needed to solve a planning problem. They proposed the $\exists$-step encoding, which allows more actions to be used during one time step on the condition that for each time step, there exists at least one valid ordering of actions executed during this step. In contrast, in $\forall$-step encodings, all orderings of parallel actions in each time step must be valid. This is achieved by constraints ensuring that all actions are applicable before the parallel step, effects of all actions are applied after the parallel step and no action destroys preconditions of any other actions. In the $\exists$-step semantics, they remove the third requirement. Based on their experiments, the $\exists$-step encoding was more time-efficient than the $\forall$-step encoding on most problems.

Wehrle and Rintanen [2007] proposed a further relaxation of the $\exists$-step plans allowing even more actions to be used in one time step. Their encoding does not require that all actions in one parallel step must be applicable in the beginning of the step on the condition that there are other actions that provide their preconditions. They compared their relaxed $\exists$-step encoding with the $\exists$-step encoding (Rintanen et al. [2006]) and SatPlan06 (Kautz et al. [2006]). The relaxed $\exists$-step encoding outperformed the other encodings in all of the selected domains, where it was possible to construct more relaxed plans.

Balyo [2013] proposed a relaxation of the relaxed $\exists$-step semantics ($R^2\exists$-Step semantics). Their encoding removes the requirement that effects of all actions must be applied in the next state. They allow actions to reset effects of other actions. The resulting encoding allows all parallel plans where in each parallel step, the actions can be executed in at least one order. Based on the experiments,

the $\mathrm{R}^2\exists$-Step encoding significantly outperformed both the $\exists$-step encoding (Rintanen et al. [2006]) and the SASE encoding (Huang et al. [2010]) in most domains and the $\mathrm{R}^2\exists$-Step encoding also required less solver calls as it produced plans with the shortest makespan.

Rintanen [2014] presented the planning system Madagascar with more versions, where some of them implement the $\exists$-step semantics. Their planner can run more SAT solvers, each for a formula representing a plan of a different length, instead of one solver for incrementally increasing plan length. A solver gets a CPU fraction proportional to γ^i $(0 < \gamma \leq 1)$ for the plan length i. They also used their own optimized SAT solver with heuristics more suitable for planning. Their planner has a performance close to the modern planners that are not based on compilation.

Balyo et al. [2015] proposed an encoding which is a combination of the transition-based SASE encoding (Huang et al. [2010]) and propositional encodings. In their Reinforced Encoding, variables represent actions, values of state variables and transitions of values of state variables. Their encoding produces parallel plans with respect to the $\forall$-step semantics allowing only independent actions to be executed during the same time step. They compared Reinforced Encoding with Direct Encoding, reimplementation of the first proposed SAT encoding (Kautz et al. [1992]) using only variables representing states and actions, SASE encoding (Huang et al. [2010]) using only action and transition variables and the $\mathrm{R}^2\exists$-Step encoding (Balyo [2013]) using a relaxed parallel semantics allowing simultaneous selection of more actions in each parallel step. Based on the total number of problems solved within a given time limit, Reinforced Encoding outperformed both SASE and Direct Encoding. The $\mathrm{R}^2\exists$-Step encoding was the most efficient; however, Reinforced Encoding outperformed the $\mathrm{R}^2\exists$-Step encoding on some domains.

Gocht and Balyo [2017] improved SAT encodings by incremental solving, which increases the performance by reusing already learnt clauses. When a plan is extended by one time point, they add to the model new universal clauses, which must hold at each time point, new transition clauses, which must hold for each pair of succeeding time points, and goal clauses for the new time point. They proposed double-ended incremental encoding, which is based on inserting new transitions between the initial and the goal state instead of adding a new goal state to extend the plan in the single-ended encoding. They used four SAT encodings to compare the effect of incremental SAT solving with single-ended and double-ended incremental encoding: the $\forall$-step and the $\exists$-step encoding from the planner Madagascar (Rintanen et al. [2006]), the Reinforced Encoding (Balyo et al. [2015]) and the $\mathrm{R}^2\exists$-Step encoding (Balyo [2013]). Based on the number of problems solved within a given time limit, the performance of the Reinforced Encoding slightly improved only with the double-ended incremental encoding. For the $\mathrm{R}^2\exists$-Step encoding, the performance improved only with the single-ended incremental encoding. For the $\exists$-step encoding from the planner Madagascar, the single-ended incremental encoding increased the number of solved problems and the double-ended encoding further improved the performance. For the $\forall$-step encoding from the planner Madagascar, incremental SAT solving, especially the double-ended encoding, improved the performance in some domains; however, in other domains, incremental SAT solving decreased the number of solved

problems, which could be caused by the efficient makespan scheduling approach implemented in the original planner Madagascar. Nevertheless, based on the comparison of solving times of individual problems, all encodings generally worked better with incremental solving and the double ended encoding was more suitable for most problems.

Froleyks et al. [2019] proposed a new algorithm for SAT-based planning called PASAR. The algorithm is based on the counterexample guided abstraction refinement technique. They used incremental SAT solving. In each iteration, the algorithm creates an abstraction of the instance of the planning problem, where all actions can be executed in parallel. If the plan found for this abstraction is not valid and the algorithm fails to transform it to a valid plan using various methods, such as forward search, the abstract encoding is refined by adding clauses from the plan used as a counterexample. This step is realized by finding interfering actions, which cannot be executed in parallel. They compared PASAR with both versions of the planner Madagascar (Rintanen [2014]), using the ∀-step or the ∃-step parallel semantics, two versions of the planner Incplan (Gocht and Balyo [2017]), using the ∀-step or the ∃-step encoding from Madagascar, and the planning system Fast Downward (Helmert [2006]) with the configuration LAMA 2011. The best version of PASAR outperformed both versions of Incplan and Madagascar. In some domains, PASAR could outperform LAMA 2001, the most successful planner.

2.3 Other approaches

Suda [2014] focused on the Property Directed Reachability algorithm (PDR) applied to automated planning and proposed further improvements of this algorithm for solving planning tasks. PDR constructs a path from the initial state to the goal state by extending the path by one step in each iteration, where a successor of a state is found by a SAT solver and the algorithm remembers reasons why a state does not have a requested successor. Similarly to planning as satisfiability, PDR increases the plan length iteratively and either finds a plan or proves that the plan of this length does not exist; however, PDR can find a plan of a length longer than the length corresponding to the current iteration. Unlike planning as satisfiability, a SAT solver does not receive the formula corresponding to the whole plan, in PDR, the solver is called only for one step in the transition system. In their planner PDRPlan, they replace SAT solver calls by planning specific procedures extending the plan in polynomial time by an algorithm which uses applicable actions. PDRPlan outperforms the original PDR with SAT solver calls. They compared PDRPlan with the planner FF (Hoffmann and Nebel [2001]), with the planning system Fast Downward (Helmert [2006]) with the configuration LAMA 2011 (Richter and Westphal [2010]), both based on heuristic search, and with the planner Mp (Rintanen [2012]) based on compilation to satisfiability. Based on their experiments, the performance of PDRPlan was comparable to the performance of the other planners in many domains and in some domains, PDRPlan solved the most problems within a given time limit.

Gregory et al. [2012] proposed a new approach, Planning Modulo Theories (PMT), inspired by Satisfiability Modulo Theories (SMT). Their planner PMT-Plan takes advantage of sub-solvers in modules instead of full compilation of

a problem for one solver. Based on their evaluation, PMTPlan could outperform the planner Metric-FF (Hoffmann [2003]), an extension of the planner FF (Hoffmann and Nebel [2001]), on some problems.

Bofill et al. [2017] proposed the $R^2\exists$-Step semantics introduced by Balyo [2013] lifted to SMT and focused on elimination of redundant actions using MaxSMT. They compared their planner with other SMT planners: the planner using the $\exists$-step semantics (Bofill et al. [2016]), the planner Springroll using the $\forall$-step semantics (Scala et al. [2016]) and SMTPlan (Cashmore et al. [2016]) focused on capturing all features of PDDL+. The $R^2\exists$-Step encoding outperformed all other encodings in most domains.

Vossen et al. [2000] focused on encoding automated planning to integer programming (IP). They compared two encodings, the direct encoding based on Graphplan and the state-change formulation, where variables representing fluents are replaced by variables representing state changes caused by actions, with the Blackbox system (Kautz and Selman [1998]). The state-change encoding, which decreases the number of constraints in comparison to the direct encoding, outperformed the direct encoding significantly; however, Blackbox was faster on most problems. They suggested preprocessing methods which could further improve the efficiency of the IP planning.

Cashmore et al. [2012] focused on compiling planning problems to quantified boolean formula (QBF). They proposed two encodings based on binary trees, which have exponentially smaller size in comparison to SAT encodings. The second presented encoding, Compact Tree Encoding, is more compact as it removes some redundancies from the first presented encoding, Flat Encoding. Their experiments showed that Compact Tree Encoding was more efficient based on both the number of solved problems and solving times. In comparison to SatPlan06 (Kautz et al. [2006]), both QBF encodings use significantly less memory; however, SatPlan06 outperformed QBF encodings based on solving times.

Gasquet et al. [2018] proposed two new QBF compact tree encodings. The first encoding, based on causal links (inspired by the plan-space planning), contains for each fluent a variable indicating if the fluent is an open condition in the current step. The second encoding, based on explanatory frame axioms (inspired by the state-space planning), defines a state by values of all fluents. In their evaluation, the encoding based on explanatory state axioms outperformed the encoding based on causal links and both encodings outperformed the compact tree encoding proposed by Cashmore et al. [2012].

2.4 Implemented models

For implementation, we selected the model TCPP (Ghooshchi et al. [2017]), the $R^2\exists$-Step semantics (Balyo [2013]) and the Reinforced Encoding (Balyo et al. [2015]). TCPP is a recent encoding proposed for compilation to CSP. The $R^2\exists$-Step semantics and the Reinforced Encoding are both among the recent approaches focused purely on encoding of the problem. Newer approaches focus also on other optimization techniques such as incremental SAT solving, which is not studied in this thesis. Another motivation for this selection is to compare TCPP to the $R^2\exists$-Step semantics and to the Reinforced Encoding since according to our knowledge, TCPP has not been compared with these models yet. Com-

paring TCPP to both these models could be interesting because they are very different. The $R^2\exists$-Step encoding uses a different definition of parallel step than TCPP, whereas the Reinforced Encoding uses a similar definition while using significantly more variables and different constraints.

3. Implementation of planning models

We implemented some of the compilation-based approaches using the programming language Picat 2.8 and we studied effects of various modifications of the original models. This chapter contains a description of these approaches, implementation details and modifications that we proposed.

All models work with problems described by multi-valued state variables using the SAS+ formalism (Bäckström and Nebel [1995]). However, planning instances used in international planning competitions are encoded in the PDDL format (Ghallab et al. [1998]); therefore, we used the translation component of the Fast Downward planning system (Helmert [2006]) to translate the tasks to SAS+.

Picat allows us to formulate the problem using various constraints and then call a solver of our choice (SAT, constraint programming, mixed integer programming or SAT modulo theory solver). For all models, we used the SAT solver provided by Picat because it outperformed other solvers.

3.1 Transition Constraints for Parallel Planning

The first implemented model is Transition constraints for parallel planning (TCPP) proposed by Ghooshchi et al. [2017] as a constraint satisfaction problem (CSP). TCPP describes a plan by value transitions of variables. This model does not require action variables as actions are described by value transitions between successive time points.

This encoding is based on **domain transition graphs** (DTG) of variables. DTG of a variable V_i is a directed graph where vertices represent values in the domain of V_i (for each value j in the domain of V_i, there is one vertex v_j representing this value). Additionally, a DTG can contain one special vertex v_* representing the **don't care value**. We use this vertex for actions which do not have the value of V_i defined in preconditions. Edges represent all actions which have V_i in effects. For an action having $V_i = j$ in preconditions and $V_i = k$ in effects, there is a directed edge $v_j \to v_k$ in the DTG of V_i. For an action having $V_i = k$ in effects where the value of V_i is not defined in precondition, there is a directed edge $v_* \to v_k$ (starting in the don't care vertex) in the DTG of V_i. For each value from the domain of V_i, we have one **no-op action**. One of the no-op actions is executed if no action changes the value of V_i during a parallel step. A no-op action can be represented by a loop on the corresponding vertex in the DTG of V_i.

Our implementation contains the method for finding the initial makespan with which we will start the search. The value of the initial makespan is based on the maximum length of the shortest paths in DTGs as proposed in the paper. In each DTG that belongs to a state variable that is a part of the definition of the goal, we find the shortest path from the initial to the goal value using the breadth-first search. The maximum of lengths of these paths corresponds to the initial makespan.

In the example problem from subsubsection 1.1.2.1, we have 5 DTGs,

one for each state variable. Figure 3.1 shows DTGs of the state variables *lift_location*, *boarded(passenger1)* and *served(passenger1)*. The state variable *boarded(passenger2)* has a DTG similar to the DTG of *boarded(passenger1)* and the state variable *served(passenger2)* has a DTG similar to the DTG of *served(passenger1)* with actions using different floors.

TCPP searches for a parallel plan. The plan corresponds to synchronized transitions in DTGs of all variables, each starting in the initial state and ending in a goal state if it is defined. These transitions are represented by table constraints. Any set of non-interfering actions can be used in one parallel step. Two actions are **interfering** if at least one of these conditions holds:

- there is a variable that is in both preconditions and effects of both actions
- there is a variable that is in both preconditions and effects of one action and in effects of the other action
- there is a variable that is in both preconditions and effects of one action and in preconditions of the other action
- there is a variable that has a value in preconditions of one action different from the value in effects of the other action
- there is a variable that has a value in preconditions of one action different from the value in preconditions of the other action
- there is a variable that has a value in effects of one action different from the value in effects of the other action

The model contains two types of variables for each time point T:

- $V_i[T]$ representing the value of the i-th state variable at the time point T; domains of these variables correspond to domains of state variables
- $P_i[T]$ representing the value of the parallelism variable of the i-th state variable at the time point T; parallelism variables prevent selection of conflict actions in one parallel step as explained in the following text

3.1.1 Transition constraints

The constraint model contains three types of constraints:

- constraints ensuring that the values of state variables at the time 1 correspond to the definition of the initial state
  $V_i[1] = init_value$ for each variable V_i
- constraints ensuring that the values of state variables at the last time point correspond to the definition of the goal
  $V_i[makespan] = goal_value$ for each variable whose value is defined in the goal
- one transition table constraint for each state variable and for each pair of successive time points $T,\ T+1$; these constraints describe transitions of values of state variables between successive time points

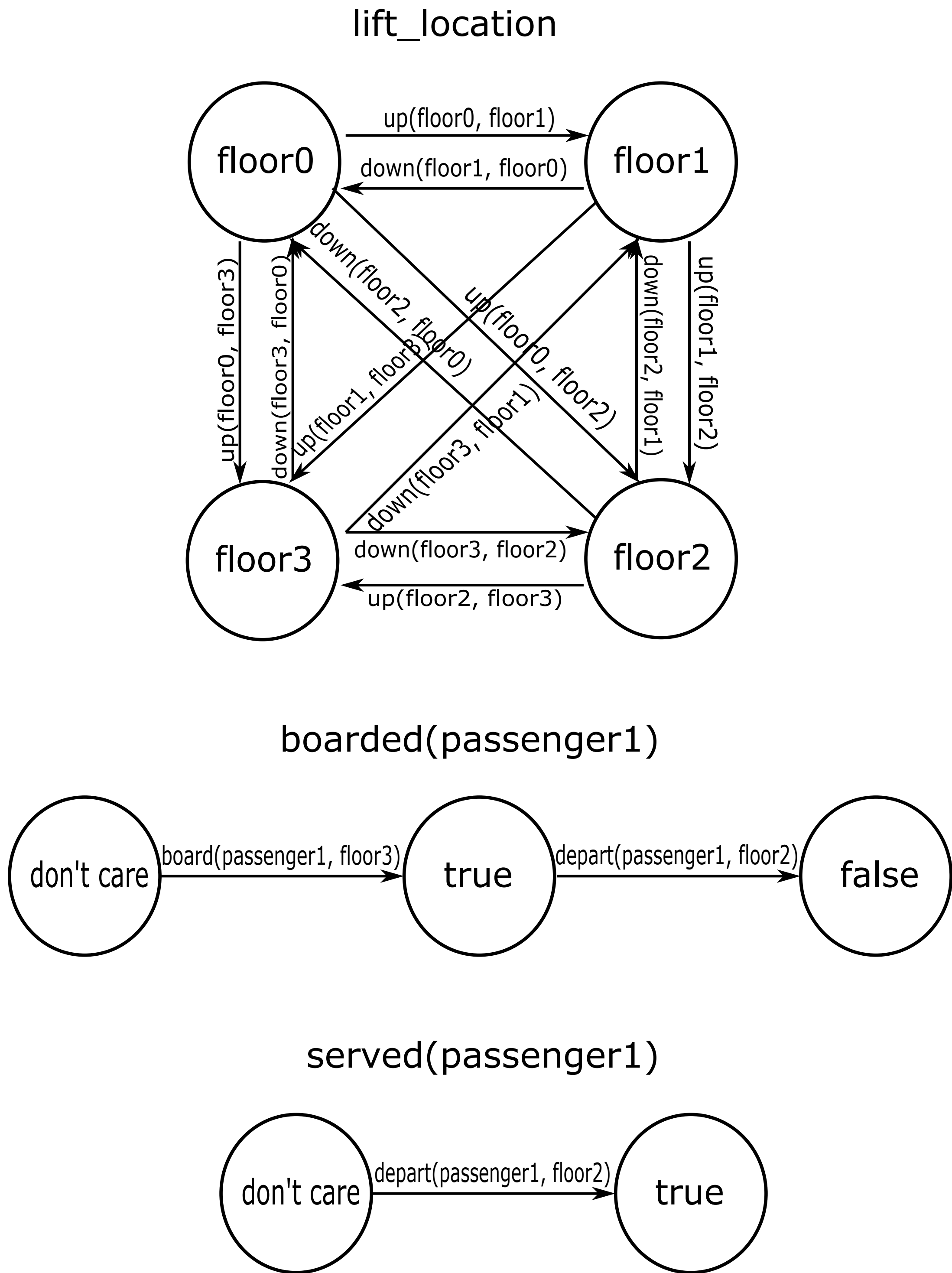

Figure 3.1: Domain transition graphs (DTGs) of the multi-valued state variables *lift_location*, *boarded(passenger1)* and *served(passenger1)* from the example from subsubsection 1.1.2.1. No-op actions correspond to loops on vertices, which are not shown in the pictures. The state variable *boarded(passenger2)* has a DTG similar to the DTG of *boarded(passenger1)* and the state variable *served(passenger2)* has a DTG similar to the DTG of *served(passenger1)* with actions using different floors.

Each transition variable has one transition table for the transition $T \rightarrow T+1$ for each $T : 1 \leq T \leq makespan - 1$; as a result, there are $|V| * (makespan - 1)$ transition table constraints in the description of the planning problem with $|V|$ state variables for a given makespan. The table of the variable V_i enumerates all actions relevant for V_i. In this context, **actions relevant for V_i** are actions that contain V_i among effects (including no-op actions). The scope of a table constraint of V_i for the transition $T \rightarrow T+1$ contains:

- $V_j[T]$ and $V_j[T+1]$ for each state variable V_j that figures in preconditions or effects of actions relevant for V_i
- $P_j[T]$ for each state variable V_j that figures in effects of actions relevant for V_i

Picat allows us to use don't care values for variables whose values are not defined in preconditions or effects of an action as suggested in the paper.

As a result, a transition table of a variable V_i has $3 * |V(V_i)|$ columns, where $|V(V_i)|$ is the number of the state variables occurring in preconditions or effects of all actions actions relevant for V_i, and $|A(V_i)| + |Domain(V_i)|$ rows, where $|A(V_i)|$ is the number of the actions relevant for V_i and $|Domain(V_i)|$ is the size of the domain of V_i (there is one no-op action for each value of V_i).

3.1.2 Parallelism variables

For each state variable, there is one **parallelism variable** for each time point. Parallelism variables prevent selection of interfering actions during the same parallel step. For each action relevant for a state variable V_i, we choose a value for the parallelism variable P_i. For each two actions that are in conflict, P_i has a different value. When a variable occurs in a table constraint as one of the effect variables of any action, its parallelism variable also occurs in this table constraint. In a lot of problems, many parallelism variables have domains of size 1 as they are not necessary to avoid simultaneous selection of conflict actions in one time step.

Each parallelism variable has a special value defined for each action occurring in the DTG of the respective state variable. To find these values while minimizing the domain sizes of parallelism variables, we construct for each parallelism variable P_i and each pair of values (v, w) from the domain of the state variable V_i a **parallel conflict graph**.

The parallel conflict graph for the parallelism variable P_i and the value pair (v, w) from the domain of V_i is defined as follows:

- Each action containing $V_i = w$ in effects is represented by a vertex in the graph. There are two types of vertices:
  1. Vertices of the first type represent actions having $V_i = v$ in preconditions and $V_i = w$ in effects
  2. Vertices of the second type represent actions which have $V_i = w$ in effects but do not have the value of V_i defined in preconditions.

- Edges in a parallel conflict graph represent the parallel conflict. Actions represented by vertices that are connected by an edge cannot be selected in one time step. Undirected edges in a parallel conflict graph are defined as follows:

  1. Each vertex of the first type is connected with all other vertices.
  2. Each vertex of the second type is connected with all vertices of the first type.

Nevertheless, we do not need to connect the vertices representing actions whose selection in the same parallel step is already prevented by another condition. Selection of two actions in the same parallel step is prevented if at least one of these preconditions holds:

- preconditions of both actions contain the same state variable with different values
- effects of both actions contain the same state variable with different values
- effects of both actions contain the same state variable and the parallelism variable of this state variable already has different values assigned for these actions

We divide the vertices to classes where each class is represented by one value of the parallelism variable P_i. All actions represented by vertices of the second type can be in the same class (represented by the same value of P_i) as they are not in parallel conflict with each other. Since actions that have V_i only in effects, unlike actions having V_i both in effects and preconditions, can appear in more parallel conflict graphs, we choose for instance the value 0 and we assign it to all vertices of the second type in all parallel conflict graphs. We can assign the same value also to all no-op actions of the state variable. The remaining values are assigned by colouring vertices of the parallel conflict graph. We used the greedy Welsh-Powell colouring algorithm (Welsh and Powell [1967]) as suggested in the paper. The colours used in one parallel conflict graph can be used again in an other graph constructed for different precondition and effect values because simultaneous selection of actions represented by vertices from different parallel conflict graphs is prevented by different precondition or effect values.

Let us assume a parallel conflict graph from Figure 3.2 constructed for the parallelism variable P_i, the precondition $V_i = \mathit{1}$ and the effect $V_i = \mathit{2}$:

- Actions A_j, A_k, A_l, A_m, A_n contain $V_i = \mathit{2}$ in effects.
- Actions A_l, A_m, A_n contain $V_i = \mathit{1}$ in preconditions.
- Actions A_j, A_k do not have the value of V_i defined in preconditions; therefore, they can be executed in any order and P_i can have the same value assigned for these actions.
- Selection of A_m and A_n in the same parallel step is prevented by a value of an other state variable in preconditions or effects (for instance, $V_h = \mathit{3}$ in preconditions of A_m and $V_h = \mathit{4}$ in preconditions of A_n).

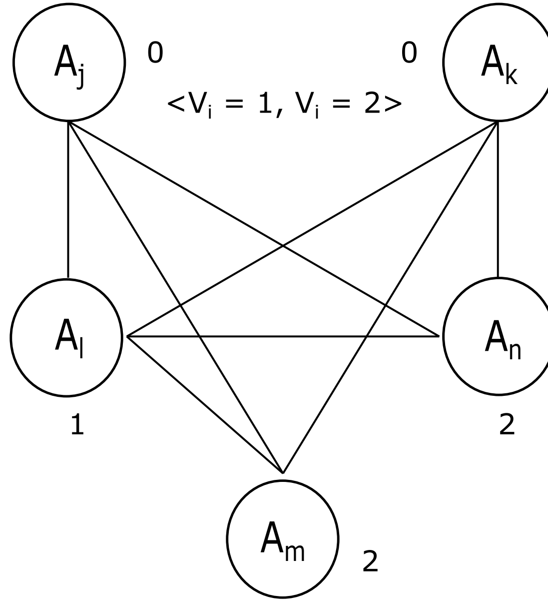

Figure 3.2: An example parallel conflict graph for the precondition value 1 and the effect value 2 of a variable V_i. Actions A_j and A_k have V_i only in effects, the other actions have the variable in both preconditions and effects. The numbers next to vertices represent the values of the parallelism variable P_i assigned to the actions.

- All interfering actions whose selection in the same parallel step is not prevented are connected by edges.
- The numbers next to the vertices represent colours assigned to them by a graph-colouring algorithm. These numbers also correspond to the values of P_i assigned to these actions.
- Execution of an action sets the corresponding value of P_i; therefore, execution of an action with a different value of P_i during the same parallel step is prevented.

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

Table 3.1: Example transition table of the model TCPP. The symbol * denotes don't care values.

Let us assume a variable V_i whose transition table for the transition between the time points T and $T+1$ is shown in Table 3.1. A value of this variable can be changed by two actions, where:

- The two actions that can modify the value of V_i correspond to the first two numeric rows of the table.

- There are two other variables which figure in preconditions or effects of these actions: V_j and V_k. These variables figure in the scope of the table besides V_i.
- P_i, P_j and P_k are parallelism variables of V_i, V_j and V_k. Parallelism variables figure in the scope of the table along with the corresponding state variables.
- The first action, corresponding to the first row, changes the value of V_i from 1 to 2 and the value of V_j from 3 to 4. The action requires in preconditions $V_k = 4$, but V_k does not figure in effects. Therefore, we must set the value to be the same after applying this action to prevent it to be changed by other actions because the precondition value must be preserved during the execution of the action.
- The second action, corresponding to the second row, changes the value of V_i to 2 and the value of V_j to 4, but their values are not specified in preconditions. This action does not operate on V_k. When the value is not defined, we use the symbol * in the table.
- Effect values of V_i are the same for both actions; therefore, these actions are in conflict. Since no values of any variables in preconditions or effects of these actions prevent selection of these two actions in the same parallel step, the value of the parallelism variable P_i must be different in the first two rows.
- Since different values of P_i already prevent simultaneous execution of these actions, the value of P_j can be the same in the first two rows.
- The last two value rows represent no-op actions of V_i.

3.1.3 Action decoding

As the model does not contain variables for actions, the plan must be decoded from values of state and parallelism variables. In each time step T, we choose for execution each action a satisfying the following conditions:

- a is applicable at the time T (all preconditions are satisfied at the time T)
- a is relevant for the time point $T + 1$ (all effects are applied at the time $T + 1$)
- Values of parallelism variables at the time T are not in conflict with a (all parallelism variables that have a defined value for a have this value at the time T)

3.1.4 Note to implementation regarding width of tables

During the implementation, we had to cope with problems regarding sizes of tables. The transition table of the variable V_i describing possible value transitions of V_i between time points T and $T + 1$ must contain all related state variables (variables occurring in preconditions or effects of any action modifying V_i). As

each of these tables contains up to 3 variables for each related variable ($V_j[T]$, $V_j[T+1]$ and the parallelism variable $P_j[T]$), some tables have can have hundreds of columns and hundreds of rows. In our experiments, Picat could not work with tables with slightly more than 100 columns; as a result, solving some of the more difficult problems (for example in the domain *airport*) was not possible.

The number of columns in transition tables can be slightly decreased by observing which parallelism variables are necessary in a table. In a transition table, we need parallelism variables of state variables occurring in effects of at least one action described in the table since variables do not have values of parallelism variables defined for actions which do not modify their value. We do not need to put parallelism variables which can have only one value to transition tables. However, the decrease in table sizes did not help us overcome this issue.

Although the size of a table constraint is not explicitly limited, in our experiments Picat could not work with problem descriptions containing tables with about 100 or more columns. To be able to solve larger problems, we decided to set a limit for the number of state variables in each transition table. The value was empirically set to 30 to ensure that the width of each table is less than 100 as each state variable adds 2 or 3 columns to the table depending on whether we need its parallelism variable in the table.

If the number of state variables exceeds the limit, we split them into more tables. These tables are connected by auxiliary variables to ensure that the state transitions are valid. This approach resembles binarization of constraints, where we represent each non-binary constraint by a variable whose domain contains tuples of values of the original variables that are compatible with the constraint. We connect these new variables by constraints ensuring that each original variable has the same value in all tuples selected from domains of the new variables (Barták [1998]). Similarly to binarization, we need to interconnect the constraints such that the value of each state variable is the same in all table constraints.

As auxiliary variable for the transition table of V_i for time points T and $T+1$, we introduce the variable $E_i[T]$ indicating which action was applied on V_i at the time point T (which edge of DTG was used). The domain of this variable corresponds to the set $\{1, \ldots, |A(V_i)|, |A(V_i)|+1, \ldots, |A(V_i)| + |Domain(V_i)|\}$, where $|A(V_i)|$ is the number of actions which modify the value of V_i and $|Domain(V_i)|$ is the size of the domain of V_i (no-op actions are included). We can use the action as a unique pointer to the other tables of V_i since exactly one transition must be executed in the DTG of each state variable between each two time points.

$V_{i_1}[T]$	$V_{i_1}[T+1]$	...	$V_{i_{30}}[T]$	$V_{i_{30}}[T+1]$	$P_{i_1}[T]$	...	$P_{i_{30}}[T]$	$E_{i_1}[T]$

$V_{i_{31}}[T]$	$V_{i_{31}}[T+1]$	...	$V_{i_{60}}[T]$	$V_{i_{60}}[T+1]$	$P_{i_{31}}[T]$	...	$P_{i_{60}}[T]$	$E_{i_1}[T]$

$V_{i_{61}}[T]$	$V_{i_{61}}[T+1]$	...	$V_{i_{90}}[T]$	$V_{i_{90}}[T+1]$	$P_{i_{61}}[T]$	...	$P_{i_{90}}[T]$	$E_i[T]$

Table 3.2: Example transition table of the model a transition table variables split to more tables. Shows headers of the three tables to which the transition table of V_{i_1} for the transition from T to $T+1$ was split.

Let us assume a variable V_{i_1} whose transition table would contain 90 variables

in its scope. Table 3.2 shows how the table can be split to more tables:

- The figure shows the headers of the three resulting tables to which the transition table of V_{i_1} for the transition from T to $T + 1$ was split.
- The actions modifying V_{i_1} contain variables V_{i_1}, ..., $V_{i_{90}}$ among preconditions and effects. These variables are split to 3 tables.
- The variable $E_i[T]$ connects the tables by pointing to the correct row of the tables.

In subsection 4.1.1, we can see that this modification allows the Picat SAT solver to solve some problems that could not be solved with the original TCPP encoding.

3.1.5 Redundant constraints

Efficiency of an encoding can be improved by adding redundant constraints to make obvious observations explicit and save the resources needed to derive them (Do and Kambhampati [2000]). We attempted to affect running times by several types of redundant constraints and we evaluate their usefulness in subsection 4.1.2.

3.1.5.1 Mutexes produced by the Fast Downward translator

The authors of the paper improved the efficiency of their model by encoding mutexes (mutually exclusive values), which are produced by the Fast Downward translator for some domains. Mutexes are values that two state variables cannot have at the same time. For each pair of variables for which the translator found at least one mutex value pair, there will be one negative table constraint for each $T : 1 \leq makespan$ with two columns denoted by the variables. Each row will contain one mutex value pair found by the translator. Picat allows us to encode mutexes as negative table constraints as suggested in the paper. In our experimental evaluation, these mutexes did not affect the performance significantly.

$V_i[T]$	$V_j[T]$
1	2
3	4

Table 3.3: Example of a negative mutex table from the model TCPP for variables V_i and V_i and the time point T. The table contains values of two variables that are forbidden at same time point.

Let us assume two variables V_i and V_j. Table 3.3 is an example negative table constraint enumerating their mutex values:

- This table enumerates mutex values of V_i and V_j (values that are forbidden at the same time point).
- The Fast Downward translator found two pairs of mutex values of these variables.

3.1.5.2 Transition constraints over a longer time horizon

Transition table constraints describe value transitions of state variables between two successive time points. To improve propagation between values of a state variable over a longer time horizon, we encoded transitions of values between remoter time points. In the DTG of each variable, we computed the length of the shortest path between each two values using the Floyd–Warshall algorithm (Floyd [1962]). These distances correspond to the minimum number of actions which are necessary to change the value of the state variable from the first to the second value. Since two actions setting a different value of one state variable cannot be used in one time step, the distance also represents the number of time steps which are necessary for transition between these two values.

We tried two ways of encoding these constraints. Firstly, we encode them as table constraints enumerating for each pairs of values whose distances in the DTG are not bigger than the required time. Secondly, we encode these constraints as negative table constraints forbidding pairs of values whose distances are bigger than the required time. This approach was motivated by the fact that this encoding is usually shorter.

For each variable, there is one table constraint for each distance in the DTG which adds at least one value pair to the transitive closure. The tables have two columns denoted by the value of the variable at the time point T and the value at a distant time point. Each row contains a pair of values reachable by the given maximum number of steps. These constraints can be constructed with the time complexity $\mathcal{O}(|V| * max_{V_i}|Domain(V_i)|^3)$, where $|V|$ is the number of variables and $max_{V_i}|Domain(V_i)|$ corresponds to the maximum domain size. $\mathcal{O}(|Domain(V_i)|^3)$ is the complexity of the Floyd-Warshall algorithm in a DTG of one variable. Similarly to the mutexes produced by the Fast Downward translator, these types of redundant constraints could not considerably improve the performance either.

Let us assume a variable V_i with $Domain(V_i) = \{1, 2, 3, 4\}$ whose DTG is a simple path from the value 1 to 4 ($1 \rightarrow 2 \rightarrow 3 \rightarrow 4$). Table 3.4 contains tables enumerating possible transitions over a longer time horizon for this variable.

- The first table enumerates all pairs of values between which the variable can transit using at most two actions.
- Then, transitions extended by the value pair {1, 4}, which requires at least 3 actions, are enumerated for each distance bigger than 2 (we eliminate paths which do not exist in the DTG).

Table 3.5 describes transitions of V_i from the previous example by negative table constraints:

- By at most 2 actions, the variable cannot transit from the value *1* to *4*, as described in the first table.
- The second table contains unreachable transitions.

3.1.5.3 GraphPlan-like propositional mutexes

We also experimented with mutexes similar to mutexes used in GraphPlan (Lopez and Bacchus [2003]). These constraints describe the possible values of

V_i[T]	V_i[T + 2]
1	1
2	2
3	3
4	4
1	2
2	3
3	4
1	3
2	4

V_i[T]	V_i[T + distance]
1	1
2	2
3	3
4	4
1	2
2	3
3	4
1	3
2	4
1	4

Table 3.4: Table constraints describing transitions of the variable V_i over a longer time horizon. The first table is defined for each $T : 1 \leq T \leq makespan - 2$ and enumerates all transitions of V_i that are possible in at most two steps. The second table is defined for each $distance, T : 3 \leq distance \leq makespan - 1, 1 \leq T \leq makespan - distance$ and enumerates all possible transitions of V_i.

$V_i[T]$	$V_i[T + 2]$
1	4
2	1
3	1
3	2
4	1
4	2
4	3

$V_i[T]$	$V_i[T + distance]$
2	1
3	1
3	2
4	1
4	2
4	3

Table 3.5: Negative table constraints describing transitions of the variable V_i over a longer time horizon. The tables enumerate value transitions that are not possible in the given number of steps. The first table is defined for each $T : 1 \leq T \leq makespan - 2$ and enumerates all transitions of V_i that are not possible in at most two steps. The second table is defined for each $distance, T : 3 \leq distance \leq makespan - 1, 1 \leq T \leq makespan - distance$ and enumerates all transitions of V_i that are not possible in any number of steps.

$V_i[T]$	$V_j[T]$	$V_i[T - 1]$	$V_j[T + 1]$
1	2	*	*
2	1	*	*
1	1	*	*
2	2	2	*
2	2	*	2

Table 3.6: Example conditional mutex table constraint of the variables V_i and V_j. The table enumerates possible values of these variables based on the values at the previous time point. If the variables can have the given values regardless of the previous value of one of these variables, the table contains the don't care symbol * in the respective cell.

two variables based on their values in the previous time point. For each two values of two variables at the time T it holds that either at least one of them had the same value at the time $T - 1$ or these values were set either by one action or by two actions which are not in conflict and thus can be used in the same time step. Therefore, each value pair that cannot be set in one parallel step is permitted on the condition that at least one value was already set at the previous time point. In other words, these values are mutually exclusive (mutex) on the condition that neither of them was already set in the previous parallel step. The other value pairs are permitted regardless on the previous values. We encoded these constraints as table constraints. There are $|V| * (|V| - 1)$ tables for each $T : 2 \leq T \leq makespan$, where $|V|$ is the number of state variables in the planning problem. Each table has four columns denoted by values of two variables in two successive time points. Each row of the table contains one value pair of the variables and the corresponding values in the previous time step. Each value pair corresponds to one or two rows of the table. If the value pair must be supported by a value of one variable in the previous time point, the table contains one row where the value of the first variable in the previous time point is specified and one row where the value of the second variable is specified. For the values that are not specified, we use the don't care value.

Let us assume two variables V_i and V_j whose conditional mutex values are described in Table 3.6, where:

- $Domain(V_i) = Domain(V_j) = \{1, 2\}$
- There is no action that has both $V_i = 2$ and $V_j = 2$ among effects and there are also no two actions providing these effects together which are not in conflict. Therefore, if both variables have the value 2, then at least one of these variables must have this value in the previous time point.
- The other pairs of values can be set in one time step; therefore, these variables can have any values in the previous time point.

The time complexity of construction of constraints of this type is $\mathcal{O}(|V|^2 * max_{V_i}|Domain(V_i)|^2 * max_{V_i}|A_{Eff_v(V_i)}|^2 * max_{A_i}(scope(A_i)))$, where $|V|$ is the number of variables, $max_{V_i}|Domain(V_i)|$ corresponds to the maximum domain size, $|A_{Eff_v(V_i)}|$ is the maximum number of actions that set the value of

V_i in effects to one specific value from its domain and $scope(A_i)$ is the number of variables which are in preconditions or effects of the action A_i. Here we assume that we can find the value of each precondition or effect variable of each action with the time complexity $\mathcal{O}(1)$ and we can find the list of actions that set the value of V_i to one specific value from its domain also with the time complexity $\mathcal{O}(1)$. For all possible values of two variables we check for each pair of actions providing this values in effects if they are compatible. For testing compatibility of actions, we need to verify that preconditions and effects of the first action are not in conflict with preconditions or effects of the second action. In the experimental evaluation, adding redundant constraints of this type to the model affected the performance negatively. Both translation and solving times increased after extending the model by these redundant constraints.

3.2 Relaxing the Relaxed Exist-Step Parallel Planning Semantics

Next, we implemented the model proposed by Balyo [2013] as a SAT model. In contrast to the ∀-step parallel semantics used in TCPP, we do not require that actions selected in each parallel step must be executable in any order. In the relaxed relaxed ∃-step (R^2∃-Step) parallel semantics, we require that the actions must be executable in at least one order. In this model, the order is fixed. For each planning problem, the authors propose to find the action ordering by modified topological sort of vertices in an enabling graph.

Enabling graph of a planning problem is a directed graph where all actions available in the problem are represented by vertices and edges connect effects of actions with preconditions of other actions that are provided by the effects. We can define the **enabling graph** as follows:

- For each action A_j, there is the vertex U_j representing this action in the graph.
- For each precondition $V_i = x$ of the action A_j, there is a directed edge $U_k \rightarrow U_j$ for each action A_k having $V_i = x$ in effects.

Let us assume an enabling graph from Figure 3.3. This enabling graph was constructed for the following planning problem:

- The planning problem contains variables V_m, V_n and V_o and actions A_i, A_j, A_k, A_l; therefore, there are four vertices denoted by A_i, A_j, A_k, A_l.
- Actions A_j and A_k provide preconditions of A_i; therefore, there are directed edges $A_j \rightarrow A_i$ and $A_k \rightarrow A_i$.
- Action A_l provides preconditions of actions A_j and A_k; therefore, there are directed edges $A_l \rightarrow A_j$ and $A_l \rightarrow A_k$.

As proposed in the paper, we find ranks of actions by the modified topological sort of vertices in the enabling graph using the depth-first search (DFS). However, enabling graphs can contain directed cycles; therefore, DFS has to be modified to ignore cycles.

The original model contains three types of variables:

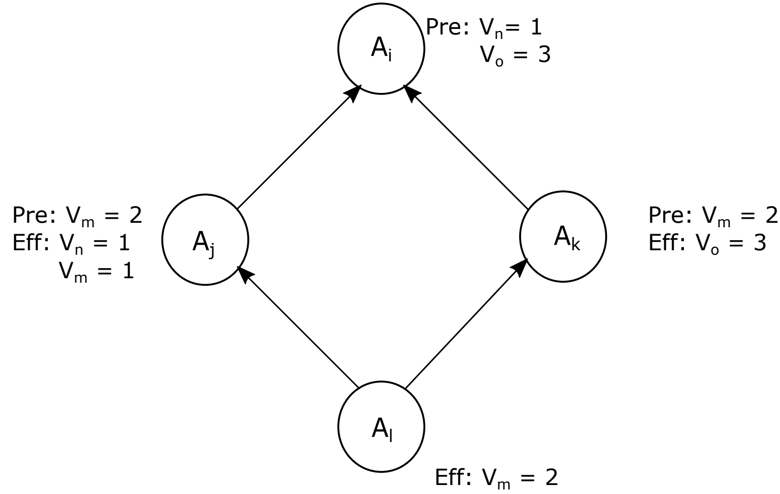

Figure 3.3: An example enabling graph of a planning problem. Vertices are denoted by actions. Next to vertices, there are preconditions (Pre) and effects (Eff) of the actions. Edges connect effects and preconditions of actions.

- Boolean state variables $V_i^v[T]$ indicating whether the value of the i-th state variable at the time point T is equal to v (for each time step T, each state variable V_i and each value v from its domain)
- Boolean action variables $A_i[T]$ indicating whether the i-th action was used during the time step T (for each time step T and each action A_i)
- Boolean chain variables $C_i^{v,\ a}[T]$ used to describe relations between preconditions and effects of actions in one time step (for each time step T, state variable V_i, value v from its domain, action a); as preconditions of an action do not have to be satisfied in the previous parallel step and can be provided by actions preceding the action in the given action ordering, we use chain variables in chain constraints to describe these relations (chain constraints are explained in the following text)

The model contains these constraints:

- Constraints enforcing values of state variables in the first time point
- Constraints enforcing values of goal state variables in the last time point
- Constraints ensuring that each variable has at most one value at each time point (at most one variable $V_i^v[T]$ can be true at each time point t for each state variable V_i)
- Action precondition clauses ensuring for each precondition of each action that if this action is used, the precondition is either valid at the beginning of the time step or provided by an action of a lower rank
- Action effect clauses ensuring that each effect of each action which is used in the time step is either applied at the end of the time step or the effect value is changed by an action of a higher rank

- Constraints ensuring that the value of each variable at the time point $T + 1$ is either the same as the value at the time point T or the value is an effect of an action used in the previous time step

- Chain constraints describing relations between preconditions and effects of actions: for an assignment $V_i = x$, we call each action that has $V_i = x$ in preconditions a **requiring action**. Each action that has $V_i = x$ in effects is a **supporting action** and each action that has among effects an assignment $V_i = y$ where $x \neq y$ is an **opposing action**. An action a requiring $V_i = x$ can be executed during the time step T if two conditions are satisfied. Firstly, either $V_i = x$ is true in the beginning of the time step T or this precondition is satisfied by executing a supporting action which has a lower rank than a. Secondly, the precondition $V_i = x$ is not destroyed by executing an opposing action after the precondition is satisfied. These relations are described by chain constraints. We describe the meaning of values of chain variables in chain constraints in detail in subsubsection 3.2.1.1.

3.2.1 Implementation

3.2.1.1 Literal implementation

Firstly, we implemented the model almost literally as described in the paper. We used the same action and chain variables, but instead of Boolean state variables $V_i^v[T]$ we used multi-valued variables $V_i[T]$ with domains corresponding to domains of state variables.

We used multi-valued state variables in the constraints as follows:

- initial state constraints:
  $V_i[1] = init_value$ for each variable V_i

- goal state constraints:
  $V_i[makespan] = goal_value$ for each variable V_i whose value is defined in the goal

- action precondition clauses:
  $A_j[T] = 0 \vee V_i[T] = precondition_value \vee A_{k_1}[T] = 1 \vee ... \vee A_{k_n}[T] = 1$ for each precondition $V_i = precondition_value$ of the action A_j, where $A_{k_1}, ...,$ A_{k_n} are actions having $V_i = precondition_value$ in effects and preceding A_j in the ordering by rank

- action effect clauses:
  $A_j[T] = 0 \vee V_i[T + 1] = effect_value \vee A_{k_1}[T] = 1 \vee ... \vee A_{k_n}[T] = 1$ for each effect $V_i = effect_value$ of the action A_j, where $A_{k_1}, ..., A_{k_n}$ are actions having the variable V_i in effects and succeeding A_j in the ordering by rank

- state consistency clauses:
  $V_i[T + 1] \neq value \vee V_i[T] = value \vee A_{k_1}[T] = 1 \vee ... \vee A_{k_n}[T] = 1$ for each variable V_i and each value $value$ from its domain, where $A_{k_1} ... A_{k_n}$ are all actions having $V_i = value$ in their effects

- chain clauses:
  For each variable V_{var} and for each value val from the domain of V_{var}, we sort all actions that are requiring, opposing or supporting the assignment $V_{var} = val$ by ranks. For each time step T, the relations of these actions are described by four types of chain constraints. In these constraints, we use auxiliary binary chain variables. The value of the chain variable $C_{var,val,a}[T]$ indicates whether the assignment $V_{var} = val$ has been destroyed by any opposing action in the moment when we would execute the action A_a during the time step T ($C_{var,val,a}[T] = 1$ if $V_{var} = val$ was destroyed, otherwise $C_{var,val,a}[T] = 0$). Chain constraints of the first type ensure that if the assignment $V_{var} = val$ had been destroyed at the moment when we would execute the action preceding A_a, then also $C_{var,val,a}[T] = 1$ if A_a is not a supporting action. The chain constraints of the second type ensure that a requiring action A_a is not executed if $C_{var,val,a}[T] = 1$. The constraints of the third type ensure that $C_{var,val,a}[T] = 1$ if we execute an opposing action A_a. The constraints of the fourth type ensure that if $C_{var,val,prev}[T] = 1$ where A_{prev} is the action preceding A_a and A_a is a supporting action, then either A_a is executed or $C_{var,val,a}[T] = 1$. We can write these constraints as follows:

  1. $C_{var,\ val,\ prev}[T] = 0 \lor C_{var,\ val,\ a}[T] = 1$ for each time T and each variable V_{var}, value val from its domain and action A_a which is either requiring or opposing the assignment $V_{var} = val$, where A_{prev} is the action situated right before A_a in the list of actions requiring, opposing and supporting $V_{var} = val$ sorted by ranks
  2. $C_{var,\ val,\ prev}[T] = 0 \lor A_a[T] = 0$ for each variable V_{var}, value val from its domain, action A_a requiring $V_i = val$ and time T
  3. $A_a[T] = 0 \lor C_{var,\ val,\ a}[T] = 1$ for each variable V_{var}, value val from its domain, action A_a opposing $V_{var} = val$ and time T
  4. $C_{var,\ val,\ prev}[T] = 0 \lor C_{var,\ val,\ a}[T] = 1 \lor A_a[T] = 1$ for each variable V_{var}, value val from its domain, action A_a supporting $V_i = val$ and time T, where A_{prev} is the action situated right before A_a in the list of actions requiring, opposing and supporting $V_{var} = val$ sorted by ranks

However, this formulation turned out to be very inefficient for Picat SAT solver (see subsection 4.2.1); therefore, we focused firstly on lowering the number of variables by removing chain variables and secondly on lowering the total number of constraints.

3.2.1.2 Replacing chain constraints by table constraints

The chain constraints describe relations between preconditions and effects of actions. Each action A_j has a set of preconditions $V_i = v$. All these preconditions must be satisfied before the action is executed. As we execute actions at each time step in one special order, all actions preceding the action A_j with respect to this ordering can change the value of V_i. Therefore, the action A_j can be executed at the time T if and only if all its preconditions are either already satisfied at the time T and not destroyed by any opposing action preceding A_j or provided by

A_n[T]	A_m[T]	A_l[T]	A_k[T]	A_j[T]	V_i[T]
0	*	*	*	*	*
1	0	1	*	*	*
1	0	0	0	1	*
1	0	0	0	0	10

Table 3.7: Example chain table constraint used in the implementation of the $R^2\exists$-Step encoding. Indicates when the action A_n can be executed based on its precondition $V_i = 10$. The other actions in the header are supporting or opposing this precondition.

any supporting action A_k preceding A_j and not destroyed by any opposing action situated between A_k and A_j. In Picat, these relations can be described by table constraints. We replaced all four chain constraints and also action precondition constraints by enumerating all possible configurations that provide preconditions of each action.

There is one table constraint for each precondition of each action and each $T : 1 \leq T \leq makespan - 1$. In a table, there is one column for the respective requiring action, one column for the respective state variable and one column for each action supporting or opposing the precondition. One row of the table corresponds to the situation where the respective action is not executed and each of the other rows corresponds to one situation where the precondition is enabled by a supporting action or the value of the state variable at the previous time point.

Let us assume an action A_n with a precondition $V_i = 10$ whose chain table constraint for the time T is shown in Table 3.7. The constraint describes the following situation:

- A_j, A_k, A_l, A_m are all actions supporting or opposing $V_i = 10$ and preceding A_n; these actions are sorted by ranks in the descending order
- A_j and A_l are supporting actions, A_k and A_m are opposing actions
- the first row describes the situation where A_n is not executed and therefore the value of V_i is irrelevant (we use the symbol * for don't care values)
- the next two rows describe the situations where the precondition is provided by one of the supporting actions; in this case, no opposing action with a higher rank can be executed
- the last row represents the situation where the precondition is already provided at the beginning of the time step; in this case, no opposing action can be executed

As we can observe in the experimental results (see subsection 4.2.1), this encoding improves running times of the SAT solver. We attempted to improve the efficiency by merging action effect clauses and state consistency clauses into table constraints. The value of any variable is determined by the last action which either modifies its value in effects or contains this variable only in preconditions

$V_i[T+1]$	$V_i[T]$	$A_j[T]$	$A_k[T]$
1	*	1	0
2	*	*	1
1	1	0	0
2	2	0	0

Table 3.8: An example table constraint used in the implementation of the R$^2\exists$-Step encoding. Describes relations of the value of the variable V_i at the time point T, its value at the previous time step and execution of actions that change the value of this variable.

(in this case, the value must stay the same after the execution of this action). If none of these actions is executed, the value of the variable stays the same as in the previous time point. In transition table constraints, we enumerate all these possibilities. There will be one table constraint for each state variable and for each $T : 2 \leq T \leq makespan$. Each table contains one column denoted by the value of the variable at the time point T, one column for the time point $T + 1$ and one column for each action that contains this variable in preconditions or effects. There is one row for each action and one row for each no-op action (each no-op action corresponds to one value of the variable). We demonstrated in the experimental evaluation that this change can improve the efficiency in some domains.

Let us assume a variable V_i whose value at the time point $T + 1$ can be explained by the transition table constraint in Table 3.8. The constraint describes the value transition of V_i as follows:

- A_j and A_k are all actions which have V_i in preconditions or effects, A_j has a lower rank than A_k
- A_j enforces the value of V_i to be 1 at the next time point, A_k sets the value to 2, V_i contains only values 1 and 2 in its domain
- the first tuple represents the situation where the last executed action is A_j
- the second tuple represents the situation where the last executed action is A_k; in this case, it is irrelevant if also A_j was executed in the same time step because A_k resets its effect
- the last two tuples represent no-op actions, where none of actions modifying the value of V_i is executed at the time T

3.2.1.3 Describing value transitions by automata

To decrease the number of constraints, we described the evolution of the value of each variable during one time step by a regular automaton (this constraint is also provided by Picat). In the automaton of the variable V_i for the time step T, there is an initial state q_0, one accepting state q_f and $|Domain(V_i)|$ additional states $q_1 \ldots q_{|Domain(V_i)|}$, where each state corresponds to one value from the domain of V_i. Actions are represented by letters of the input alphabet. The evolution of values starts with the value $V_i[T]$ and ends with the value $V_i[T + 1]$

We can describe transitions in the automaton as follows: Firstly, the automaton reads the value $V_i[T]$ and transits from the initial state to the state labelled by the value of $V_i[T]$. This rule ensures that the sequence of values of V_i in the time step T accepted by the automaton starts in the value of $V_i[T]$.

Then, the automaton reads the sequence of all actions relevant for V_i. In this context, **action relevant for V_i** is each action which has V_i in preconditions or effects. One letter in the input alphabet is reserved for actions that were not executed in the time step T. For each action A_j, such that $A_j[T] = 0$, the automaton reads the special symbol, which represents the universal no-op action, and the automaton stays in the previous state. For each action A_j, such that $A_j[T] = 1$, the automaton reads the unique identifier assigned to A_j and if the current state of the automaton is consistent with preconditions of A_j, the automaton transits to the state given by effects of A_j. These rules ensure that the temporary value of V_i is during the time step T changed if and only if an action containing V_i in effects is executed and preconditions related to V_i of all executed actions are satisfied.

After reading all actions, the automaton reads the letter corresponding to $V_i[T + 1]$. Here we have a unique identifier for each value from the domain of V_i different from identifiers of actions. If the automaton reads the identifier corresponding to the state in which the automaton currently is, then the automaton transits to the accepting state. This rule ensures that the automaton of $V_i[T]$ finishes in the state corresponding to the value of $V_i[T + 1]$.

The transition function is defined as follows:

- $\delta(q_0, input_value_j) = q_j$ for each value j from the domain of V_i ($input_value_j$ is the symbol identifying an input value)
- $\delta(q_j, a) = q_k$ for each action identifier a such that $V_i = value_k$ is in effects of the action and either $V_i = value_j$ is in preconditions or V_i is not in precondition of this action
- $\delta(q_j, a) = q_j$ for each action identifier a such that $V_i = value_j$ is in preconditions of this action and V_i is not in effects of this action
- $\delta(q_j, noop) = q_j$ where *noop* is the identifier reserved for all actions which are not executed (the universal no-op action)
- $\delta(q_j, output_value_j) = q_f$ for each value j from the domain of V_i ($output_value_j$ is the symbol identifying an output value)
- $\delta(q, x) = error_state$ for each transition not defined above

There is one automaton defined for each state variable and each time point $T : 1 \leq T \leq makespan - 1$. By this formulation, the size of the input alphabet of the automaton of $V_i[T]$ is $|Domain(V_i)| + |A(V_i)| + 1$, where $|A(V_i)|$ is total number of actions that have V_i in preconditions or effects. We can use the same symbols for the input and output values as the transitions are distinguished by the origin state.

The size of the alphabet can be reduced, if we consider value transitions *precondition_value* → *effect_value* instead of actions. The size of the input alphabet

is therefore $|Domain(V_i)| + |T(V_i)| + 1$ where $|T(V_i)|$ is the number of all possible transitions between values of V_i including no-op actions and transitions from the don't care vertex. Each action is then represented by the unique identifier of the corresponding transition. More actions can now have the same identifier.

The transition function is redefined as follows:

- $\delta(q_0,\ input_value_j) = q_j$ for each value j from the domain of V_i ($input_value_j$ is the symbol identifying an input value)
- $\delta(q_j,\ t) = q_k$ for each transition identifier t such that $value_k$ is the goal value of the transition and either $value_j$ is the initial value or the initial value is the don't care value
- $\delta(q_j,\ t) = q_j$ for each transition identifier t, which represents the no-op transition on the value $\mathrm{value_j}$; these transitions correspond to actions where V_i is in preconditions but not in effects
- $\delta(q_j,\ noop) = q_j$ where *noop* is the identifier used for all transitions (actions) which are not executed (the universal no-op action)
- $\delta(q_j,\ output_value_j) = q_f$ for each value j from the domain of V_i ($output_value_j$ is the symbol identifying an output value)
- $\delta(q,\ x) = error_state$ for each transition not defined above

Let us assume a regular automaton in Figure 3.4 describing the evolution of the value of a variable V_i during the time step T, where:

- $Domain(V_i) = \{1,\ 2\}$
- A_j changes the value of V_i from 1 to 2
- A_k requires $V_i = 1$ in preconditions
- The input word of the automaton is the sequence $<input_value_id(V_i[T]),\ transition_id(A_j[T],\ T_j[T]),\ transition_id(A_k[T],\ T_k[T]),\ output_value_id(V_i[T+1])>$, where T_j and T_k are identifiers of the transitions of V_i caused by actions A_j and A_k and $transition_id(T_l[T],\ A_l[T])$ is a function that returns the letter representing the transition T_l if the action A_l was executed during the time step T ($A_l[T] = 1$) or the letter representing the universal no-op action if the action was not executed ($A_l[T] = 0$); the function can be defined as follows: $transition_id(T, A) = T * A$
- The input alphabet contains 6 letters: we use 2 letters to represent $V_i[T]$ and no-op actions (here the meaning of the symbol can be derived from the state in which the symbol is read), 2 letters representing $V_i[T+1]$ (here we need to distinguish no-op actions and output values), one letter representing the universal no-op action and one letter for the only non-no-op transition $(1 \rightarrow 2)$
- In the initial state, the automaton reads the value of $V_i[T]$ and transits to the corresponding state

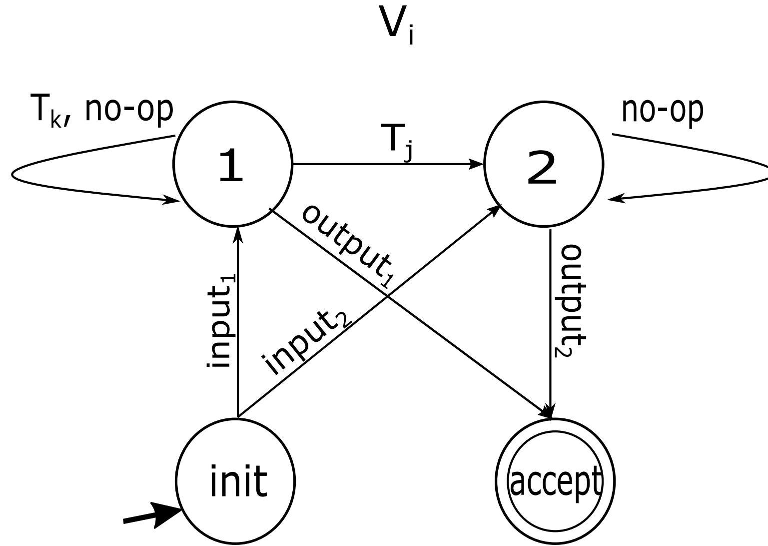

Figure 3.4: An example automaton describing evolution of values of the variable V_i during the time step T. States 1 and 2 represent the values of V_i, *init* is the initial state of the automaton and *accept* is the accepting state. Edges are denoted by letters from the input alphabet. The letter no-op represents the universal no-op action (representing the situation when one of the actions is not executed), T_j and T_k represent execution of the corresponding transitions (corresponding to the actions A_j and A_k), *input₁* and *input₂* correspond to the letters indicating that the value of V_i is 1 or 2 in the previous time point T (input value of the automaton) and *output₁* and *output₂* correspond to the letters indicating that the value of V_i is 1 or 2 in the current time point $T + 1$ (output value of the automaton). The input sequence of the automaton starts with the value of $V_i[T - 1]$. Then the sequence contains for each action having V_i in preconditions or effects either the letter representing the corresponding transition or the universal no-op symbol, where these symbols are sorted by ranks of the actions. The sequence ends with the value of $V_i[T]$. The variable V_i has one copy of this automaton for each time point $T : 1 \leq T \leq makespan - 1$, where *makespan* is the length of the parallel plan.

- For each action in the sequence sorted by action ranks, the automaton reads either the symbol representing the respective transition (for actions that are executed) or the universal no-op symbol (for actions that are not executed)

- The automaton transits to the accepting state when it reads the symbol representing the output value 1 in the state 1 or the symbol representing the output value 2 in the state 2

- The transitions that are not defined lead to an error state

3.2.1.3.1 Note to implementation regarding width of transition matrices During the implementation of the $R^2\exists$-Step encoding, we had to cope with a similar problem as we described in subsection 3.1.4. In the planning problems that we used in experimental evaluation, Picat could not work with automata with the input alphabet slightly larger than 100. To overcome this problem, we split the automata to more parts each having the number of input actions at most 100, which also decreases the number of transitions (represented in the input alphabet) to at most 100. This limit was suitable for all problems that we tested. The automata are connected by new auxiliary variables. The auxiliary variable for the first automaton of V_i represents the intermediate value of V_i after applying the first 100 actions during the time step T. This value is the input value of the second automaton. We need a new auxiliary variable for connecting each two successive automata.

Although using regular automata with unlimited width of transition matrices instead of table constraints did not improve the efficiency, according to the experimental results presented in subsection 4.2.1, the encoding with regular automata with limited width has the best performance.

Let us assume a transition automaton of V_i, which had to be split to more automata as shown in Figure 3.5, where:

- there are 300 actions which have V_i in preconditions or effects, each automaton is defined for 100 actions (at most 100 transitions are defined in each automaton)

- each automaton contains one part of the transition sequence between the input and output value of its input sequence

- the input sequence of the first automaton starts with $V_i[T]$ and ends with the first intermediate value

- the input sequence of the second automaton starts with the first intermediate value and ends with the second intermediate value

- the input sequence of the third automaton starts with the second intermediate value and ends with $V_i[T + 1]$

3.2.2 Ranking

The efficiency of the model is affected by ranking of actions. A better ordering of actions allows solving a planning problem in fewer parallel steps. Firstly, we implemented the ranking based on enabling graphs proposed in the paper. Then,

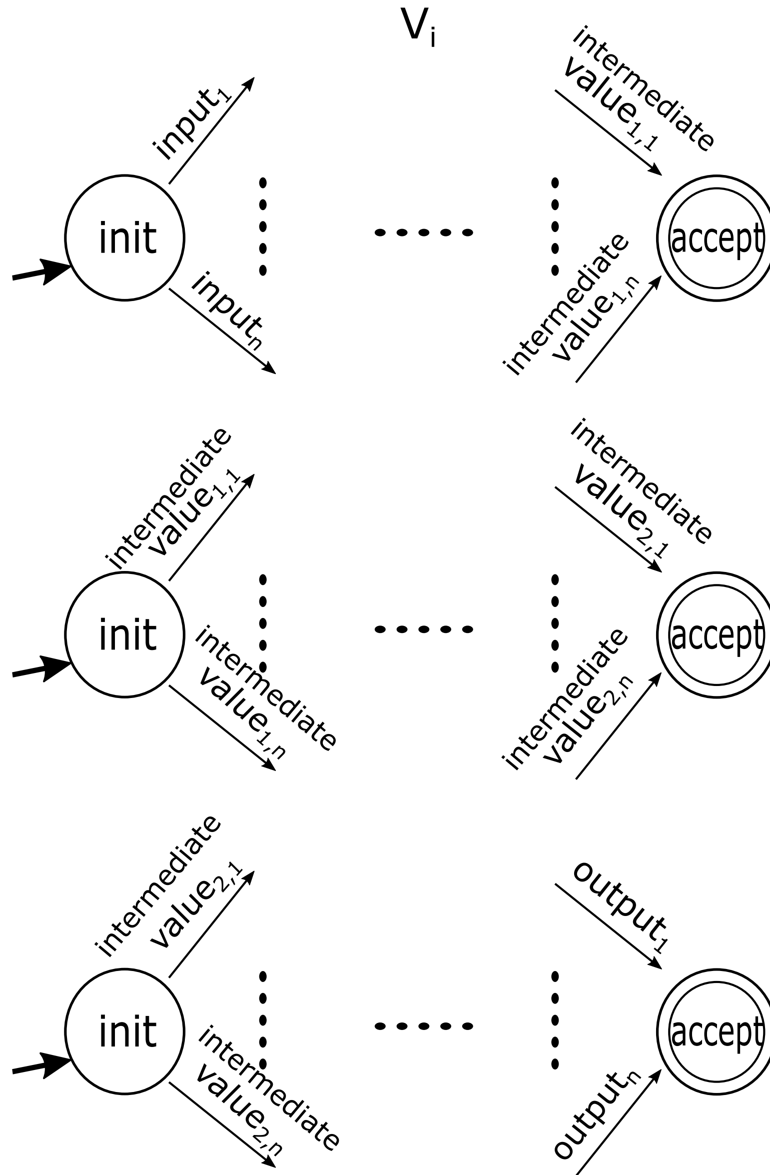

Figure 3.5: An example automaton describing evolution of values of the variable V_i during the time step T that was split to three automata because the transition matrices of the states of the automaton were too wide. Each automaton contains only a subset of all transition of the variable. Each automaton accepts one part of the sequence. The first value in the input sequence of the first automaton is $V_i[T]$, $V_i[T+1]$ is the output value of the third automaton. There are two intermediate values connecting the output of the first and the input of the second automaton and the output of the second and the input of the third automaton. The state variable V_i has a copy of each of these automata for each time point $T : 1 \leq T \leq makespan$, where *makespan* is the length of the parallel plan.

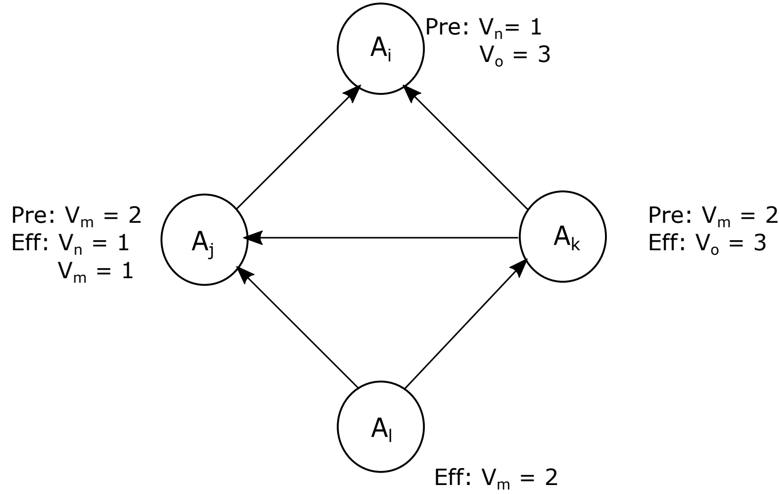

Figure 3.6: An example enabling-disabling graph of a planning problem. Vertices are denoted by actions. Next to vertices, there are preconditions (Pre) and effects (Eff) of the actions. $A_k \rightarrow A_j$ is a disabling edge, the other edges are enabling.

we modified the ranking by adding additional disabling edges to describe relations between actions, which are not yet connected in the enabling graph. For a pair of actions A_i, A_j where $V_k = v$ is an effect of A_i and $V_k = w$ is a precondition of A_j and $v \neq w$ (A_i destroys a precondition of A_j), disabling edge is an edge directed from the vertex representing A_j to the vertex representing A_i. We create an **enabling-disabling** graph from an enabling graph by adding disabling edges between vertices that are not yet connected by enabling edges. When we are searching for the topological sorting in an enabling-disabling graph using DFS, we always start in the vertex with the lowest indegree (number of incoming edges).

Figure 3.6 contains an enabling-disabling graph created from the enabling graph from the example presented in the beginning of section 3.2. The effect $V_m = \mathit{1}$ of the action A_j destroys the precondition $V_m = \mathit{2}$ of the action A_k. Since these actions were are not connected by an enabling edge, we add a disabling edge $A_k \rightarrow A_j$. In the original enabling graph (Figure 3.3), the topological sort could find two possible action orderings: $< A_l, A_k, A_j, A_i >$ and $< A_l, A_j, A_k, A_i >$. The disabling edge forces the algorithm to place A_k before A_j. If we use the ordering $< A_l, A_k, A_j, A_i >$, we can execute all actions in one parallel step and therefore we can find a plan with a shorter makespan.

According to the results of the experiments presented in subsection 4.2.2, the addition of disabling edges could significantly increase the number of solved problems in some domains. Also the total number of problems solved within the given time limit was higher for the ranking with disabling edges.

3.2.3 Redundant constraints

Similarly to TCPP, we attempted to improve the efficiency of R$^2\exists$-Step by redundant constraints. In this section, we describe the suggested redundant constraints.

$A_j[T]$	$V_i[T+1]$
0	*
1	1
1	2

Table 3.9: An example table constraint describing which values of the variable V_i are possible after executing the action A_j during the previous time step.

3.2.3.1 Mutexes produced by the Fast Downward translator

Similarly to the previous model, we examined the impact of extending the encoding by mutexes produced by the translator of Fast Downward (see subsubsection 3.1.5.1).

3.2.3.2 Actions disabling values of state variables

In a regular automaton that we use to describe the evolution of the value of the variable V_i during the time step T, the edges representing actions are oriented in the direction of ranking. When we execute an action, it can lead to a part of the automaton, where some vertices representing values are no longer reachable. Therefore, after choosing an action, some values may not be available for V_i at the time point $T + 1$. To improve the propagation and make the restriction of the domain of $V_i[T + 1]$ obvious, we propose constraints to describe these combinations of actions at the time T and values at the time $T + 1$. We use a table constraint to describe values of V_i forbidden by the action A_j. We find values compatible with A_j by walking through an automaton in the opposite direction to the ranking and remembering values reachable after each action. There will be one table constraint for each state variable and each action having the variable in preconditions or effects for each $T : 2 \leq T \leq makespan$.

Let us assume a variable V_i and an action A_j where the relation between the value of V_i at the time point $T + 1$ and the execution of A_j during the previous time step can be described by Table 3.9 as follows:

- $Domain(V_i) = \{1, 2, 3, 4\}$
- If the action A_j is not executed, it does not restrict possible values of V_i.
- In the automaton of V_i, there are is an edge from the vertex 3 to 2 labelled by A_j. No actions of a rank higher than $rank(A_j)$ lead to 3 or 4. Therefore, if the action A_j is executed, the value of $V_i[T + 1]$ cannot be 3 or 4.

These redundant constraints can be constructed in the time $\mathcal{O}(|V| * max_{V_i}|A(V_i)| * max_{V_i}|Domain(V_i)|)$, where $|V|$ is the number of state variables, $max_{V_i}|Domain(V_i)|$ corresponds to the maximum domain size of a state variable and $|A(V_i)|$ is the number of actions having the variable V_i in preconditions or effects. For each variable we need to walk through actions in the reverse ranking order and enumerate all values from the domain of the variable which are available after applying the action. In the experimental evaluation (see subsection 4.2.3), these constraints could slightly improve the performance in some domains.

3.3 Reinforced Encoding

The third implemented model is Reinforced Encoding proposed by Balyo et al. [2015] for SAT solvers. This model combines variables for actions, transitions and state variables:

- Boolean state variables $V_i^v[T]$ indicating whether the value of the i-th state variable at the time point T is equal to v (for each time step T and each value v of each state variable V_i)
- Boolean action variables $A_i[T]$ indicating whether the i-th action was used in the time T (for each time step T and each action A_i)
- Boolean transition variables $C_i^{v1 \rightarrow v2}[T]$ indicating whether the variable V_i changed the value from v_1 to v_2 during the time step T (for each time step T, each state variable V_i and each pair of values v_1, v_2 of the state variable)

We used multi-valued variables for state variables and for transitions:

- state variable $V_i[T]$ with the domain corresponding to $Domain(V_i)$ for each time step T and each state variable V_i
- transition variable $T_i[T]$ for each time step T and each state variable V_i with the domain corresponding to $\{1, ..., |Transitions(V_i)| + |Domain(V_i)|\}$, where $|Transitions(V_i)|$ is the number of all possible transitions of the variable V_i; since only one transition can occur on one variable during one time step, we assign a unique identifier to each possible transition of the variable V_i including no-op transitions.
- Boolean action variable $A_i[T]$ for each time step T and each action A_i

3.3.1 Implementation

3.3.1.1 Literal implementation

Firstly, we implemented the constraints as described in the paper:

- Transition effect clauses $T_i[T] \neq t \lor V_i[T] = e$, where t is a transition of V_i ending in the value e of the variable V_i, for each time point T
- Transition precondition clauses $T_i[T] \neq t \lor V_i[T - 1] = p$, where t is a transition of V_i beginning in the value p of the variable V_i, for each time point T
- Value explanation clauses $V_i[T] \neq v \lor T_i[T] = t_{k_1} \lor ... \lor T_i[T] = t_{k_m}$, where t_{k_1} ... t_{k_m} are transitions of the variable V_i ending in the value v, for each time point T
- Action transition clauses $A_j[T] = 0 \lor T_i[T] = t$, where A_j is an action providing the transition t of the variable V_i, for each time point T
- Transition explanation clauses $T_i[T] \neq t \lor A_{k_1}[T] = 1 \lor ... \lor A_{k_n}[T] = 1$, where $A_{k_1}, ..., A_{k_n}$ are actions providing the transition t of the variable V_i, for each time point T

V_i[T - 1]	V_i[T]	T_i[T - 1]
1	2	1
*	1	2
1	1	3
2	2	4

V_i[1]	T_i[1]
1	2
2	4

Table 3.10: Example tables describing relations of the transition variable of a variable and the values of the state variable in two successive time points. The first table is defined for each $T : 2 \leq T \leq makespan$. The second table enumerates possible transitions at the initial time point based on the value in the initial state.

- Action conflict clauses $A_j[T] = 0 \vee A_k[T] = 0$, where A_j and A_k are conflict actions, for each time point T
- Initial transition clauses $T_i[1] \neq t$, where t is a transition which is not applicable in the initial state for V_i
- Goal clauses $V_i[makespan] = g$, where g is the goal value of V_i

Since the literal encoding results in large number of constraints, we attempted to reduce the number of them by utilising constraints provided by Picat.

3.3.1.2 Implementation with table and other Picat constraints

The number of constraints can be slightly decreased by describing preconditions and effects of transitions by table constraints instead of clauses. The scope of a transition table of V_i contains variables $V_i[T - 1]$, $V_i[T]$ and $T_i[T]$ and enumerates all possible transitions with their precondition and effect values (including no-op transitions). We can use the don't care value for transitions with undefined precondition value of V_i. There is one table constraint for each state variable and each time point $T : 2 \leq T \leq makespan$ describing transitions between two successive time points. Each table has three columns denoted by values of the variable at the time points $T - 1$ and T and the transition during the time step $T - 1$. Each row corresponds to one transition of this variable (including no-op transitions). For each state variable, there is also one table constraint containing only transitions that are possible from the initial state. This table contains two columns denoted by the value of the state variable and the transition of the variable at the first time point.

Let us assume a variable V_i whose transitions can be described by table constraints from Table 3.10 as follows:

- $Domain(V_i) = \{1, 2\}$.
- $V_i[1] = 2$

- The first table is defined for each $T > 2$, the second table describes possible transitions in the initial state.
- The transition number 1 changes the value of V_i from 1 to 2.
- The transition number 2 changes the value of V_i from any value to 1.
- Transitions 3 and 4 are no-op transitions.

By using these tables, we can remove the original transition effect clauses, transition precondition clauses, value explanation clauses and initial transition clauses from the constraint model. To lower the total number of constraints, we can describe the relations between action and transition variables using a single equivalence for each value of each transition variable. Any non-no-op transition of a variable occurs if and only if at least one action providing this transition is executed during the time step. For no-op transitions, we need only one implication.

Let us assume a transition $T_i = t$ whose relations with actions during the time step T can be described as follows:
$T_i[T] = t \Leftrightarrow A_j[T] \vee A_k[T] \vee A_l[T]$
$A_m[T] \vee A_n[T] \Rightarrow T_i[T] = u$

- The transition t of the variable V_i is provided by actions A_j, A_k and A_l.
- At least 1 of these action variables is equal to 1 during the time step T if and only if the transition t occurs.
- The transition u is a no-op transition required by actions A_m and A_n.

These constraints replace transition explanation clauses and action transition clauses in the constraint model. However, none of these modifications could improve the performance of the encoding (see subsection 4.3.1). The original constraints were more efficient.

3.3.2 Modification of action conflict constraints

We can also modify action conflict clauses. Firstly, we will slightly redefine conflict actions. By Balyo et al. [2015], two actions cannot be executed during the same time step if they are not independent. Actions are not independent if at least one variable from preconditions or effects of the first action occurs in preconditions or effects of the second action. By this definition, two actions cannot be executed together for instance if they have the same precondition. However, if this precondition variable does not occur in effects of any of these actions, these actions could be selected in one parallel step because they can be executed in any order. We will now allow selection of these pairs of actions in the same time step.

During each time step, at most one action that causes any transition $Value_1 \rightarrow Value_2$ of the variable V_i where $Value_1 \neq Value_2$ can be executed. In each parallel step, we can replace action conflict clauses by one constraint for each state variable ensuring that at most one of the actions that have this variable both in effects and preconditions is executed. For these constraints, we used

the Picat predicate *sum/1*, where the sum of the involved action variables must be less than 1 to ensure that at most one of the actions is executed.

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

In the sequence passed as an argument to the predicate *sum/1*, there are only actions having the variable V_i in both effects and preconditions. Since we do not forbid selection of actions having V_i only in effects or only in preconditions in the same parallel step, we allow plans with more actions in each parallel step.

Actions with V_i only in preconditions correspond to a no-op transition $x \rightarrow x$ for a value x from the domain of V_i. Actions corresponding to the same no-op transition are not in conflict as they can be executed in any order. Actions having V_i only in effects can be selected in the same parallel step with actions related to the same transition $* \rightarrow x$ as these actions can also be executed in any order. Selection of more actions with V_i both in effects and preconditions in one step is prevented. Here we follow the definition of parallel conflict from Ghooshchi et al. [2017].

3.3.2.1 Note to implementation regarding the length of a sum

The lists inside the sums must contain all actions related to the variable. In our experiments, Picat could not work with lists with slightly more than 200 elements. Therefore, we had to split the actions to more sums by introducing auxiliary variables representing intermediate sums. The limit of 200 elements was suitable for all tested planning problems.

Let us assume a variable V_i whose sequence of actions in action conflict constraints is too long. We can split the sum to more constraints as follows:
$sum(<A_{k_1}[T], ..., A_{k_2 00}[T]>) = S_{i,1}[T]$
$sum(<A_{k_2 01}[T], ..., A_{k_4 00}[T]>) = S_{i,2}[T]$
$S_{i,1}[T] + S_{i,2}[T] \leq 1$

- The sum of all variables of actions having V_i in both preconditions and effects, A_{k_1}, ..., $A_{k_4 00}$, are split to two sums.
- At most one action is executed during the time step T if and only if the sum of all action variables is at most 1.
- Since also the intermediate sums can be also equal to at most 1, the domains of the intermediate sum variables of V_1, $S_{1,1}$ and $S_{1,2}$, correspond to $\{0, 1\}$.

3.3.3 Redundant constraints

3.3.3.1 Mutexes produced by the Fast Downward translator

Similarly to the other models, we can add the mutexes generated by the Fast Downward translator component (see subsubsection 3.1.5.1).

$T_i[T]$	$T_i[T+1]$
2	1
3	1
*	2
2	3
3	3
1	4
4	4

Table 3.11: An example transition sequencing constraint. Describes all possible transitions of one variable in two successive time points. The symbol * denotes the don't care value.

3.3.3.2 Transition sequencing constraints

We evaluated the efficiency of extending Reinforced Encoding by transition sequencing constraints. We were inspired by action sequencing constraints proposed by Barták [2011a] although our constraints are weaker. Transition sequencing constraints are encoded using table constraints and enumerate all transitions of one variable that are possible in two successive time steps. For each state variable, there will be one transition sequencing table constraint for each time point $T : 1 \leq T \leq makespan - 1$. Each table has two columns corresponding to possible transitions between the time points T and $T + 1$. For each transition, there is at least one table row. If the transition starts in the don't care value, one row corresponds to this transition with the don't care value for the preceding transition. Otherwise, there is one row for each transition that can precede this transition.

Let us assume a variable V_i whose transition sequencing table constraint is shown in Table 3.11:

- There are 4 possible transitions of the variable V_i:
  1. $1 \rightarrow 2$
  2. $* \rightarrow 1$ (* is the don't care value)
  3. no-op transition $1 \rightarrow 1$
  4. no-op transition $2 \rightarrow 2$
- Before transition number 1, there can be any transition ending in the value 1 (transition 2 or 3).
- Before the transition number 2, there can be any transition because the precondition value is not defined.
- Before a no-op transition, there can be any transition ending in the corresponding value.

The complexity of the construction of transition sequencing constraints is $\mathcal{O}(|V| * max_{V_i}|T(V_i)|^2)$, where $|V|$ is the number of variables and $max_{V_i}|T(V_i)|$ is the maximum number of possible transitions of a variable. For each variable,

we need to enumerate for each transition all other transitions and check if they can precede it.

4. Experimental evaluation

This chapter contains experimental evaluation of the implemented models. We performed experiments to compare different versions of our implementation of the models and to evaluate usefulness of the proposed redundant constraints. The goal of the experiments was also to compare different models. All experiments were run on a computer with the Intel Core i7-8550U CPU @ 1.80GHz processor and 16 GB of RAM. For solving the instances, we used the SAT solver provided by Picat 2.8. In most experiments, we measure for each domain the total number of instances solved within the time limit of 60 minutes per instance. Besides the solving time, the total time includes also the translation time and the time used for all processing steps. We measure the total time from translation from PDDL to finding a plan. For evaluation of encoding modifications, we used planning problems from the domains *airport*, *grid*, *miconic*, *pegsol*, *storage* and *zenotravel*. For comparison of different models, we used in addition the domains *tpp*, *parcprinter* and *woodworking*.

We have chosen the domain *miconic* since it contains a wide range of problems from simple problems that were solved with all encodings, including the most literal implementations, to difficult problems that were not solved with any of the implemented models. On the domain *airport*, we demonstrated how limitations regarding sizes of table constraints had to be overcome. The domain *grid* contains difficult problems where the number of solved problems was low with all encodings and with some encodings, we could not solve any problem. The other domains were selected from the domains used for experimental evaluation in the three original papers. The domains *storage* and *zenotravel* were used by Ghooshchi et al. [2017] and Balyo [2013]. The domains *pegsol*, *parcprinter* and *woodworking* was used by Balyo [2013] and Balyo et al. [2015]. The domain *tpp* was used by Ghooshchi et al. [2017].

The domain *airport* from the International Planning Competition 2004 contains problems from the airport ground traffic control. Problems from the domain *grid* from the International Planning Competition 1998 represent grids with locked cells and keys of different shapes where a robot must move keys to required cells. An older domain *miconic* describes elevator control with more elevators and passengers. In the goal state of each problem, all passengers are served. The domain *parcprinter* from the International Planning Competition 2011 models a multi-engine printer which can process more task simultaneously. The domain *pegsol* from the International Planning Competition 2011 contains problems from the board game peg solitaire. The domain *storage* from the International Planning Competition 2006 describes logistic problems where the goal is to move crates from containers to depots by hoists. The domain *tpp* from the International Planning Competition 2006 models the travelling purchase problem, a generalization of the travelling salesman problem, where the goal is to select a subset of markets such that the required products can be purchased. The domain *woodworking* from the International Planning Competition 2011 models a woodworking workshop. In the domain *zenotravel* from the International Planning Competition 2002, the goal is to transport persons from their initial locations to their destinations by planes.

4.1 TCPP

In this section, we evaluate different versions of the model TCPP proposed by Ghooshchi et al. [2017].

4.1.1 Implementation

In this section, we evaluate the effect of shortening tables by introducing variables represented edges as described in subsection 3.1.4. This modification was motivated by the observation that in some domains, some SAS+ variables can interact with many different variables in actions; as a result, their transition tables have many columns columns. We can see the number of problems solved by the original TCPP in the second numeric column in Table 4.1 and the number of problems solved by TCPP with reduced table width in the next column.

In the domain *airport*, the solver fails to solve many problems due to sizes of transition tables. The failure is manifested as *domain_error*. According to A User's Guide to Picat (Zhou and Fruhman [2020]), this error indicates that there was an object which had a value which was not in its domain. However, we observed that this error occurred in our experiments when a table constraint or an other constraint was too large although the sizes of the constraints are not explicitly limited in the Picat documentation. We could resolve this error by reducing sizes of the constraints. When we limit the width of transition tables by introducing additional variables, the number of solved problems in the domain *airport* significantly increases. Introducing additional variables and splitting transition tables to more tables slightly increases solving times; however, the number of solved problems is not affected considerably. In the next experiments, we always use the version with limited width of transition tables.

TCPP							
domain	total	o	l	lt	ln	lc	lm
airport	50	8	**23**	21	20	15	22
grid	5	1	1	**2**	**2**	1	1
miconic	150	**59**	58	57	57	57	-
pegsol	20	**15**	**15**	**15**	14	11	-
storage	30	**16**	**16**	**16**	**16**	15	**16**
zenotravel	20	**16**	**16**	**16**	15	**16**	-
total	275	115	**129**	127	124	115	128

Table 4.1: Comparison of modifications of TCPP. Number of problems solved within 1 hour for the original TCPP (*o*), TCPP with limited table width (*l*), *l* with transition tables describing transitions of values over a longer time horizon (*lt*), *l* with transition tables describing transitions of values over a longer time horizon described by negative table constraints (*ln*), *l* with conditional mutex contraints (*lc*), *l* without mutex constraints produced by Fast Downward (*lm*). In the last column, the dash is used for domains where Fast Downward does not produce mutexes and the value would equal to the value in the column *l*.

4.1.2 Redundant constraints

In this section, we evaluate usefulness of three types of redundant constraints described in subsection 3.1.5. As we can see in the last four columns of Table 4.1, extending the constraint model by longer time horizon was slightly helpful in the domain *grid* where the number of solved problems increased by one. However, in the domain *airport*, adding these constraints decreased the number of solved problems by one and in the domain *pegsol*, these types of constraints had no effect. Judging by the results in the domain *airport*, the implementation is more efficient with table constraints than with negative table constraints. The second type of redundant constraints, forbidding values that cannot be assigned to two variables during the same time step, decreased the efficiency of the model, especially in the domains *airport* or *pegsol*. Figure 4.1 shows the number of problems solved with TCPP with two different types of redundant constraints with increasing time. We can see that especially the second type of redundant constraints decreases the performance significantly. The mutex constraints proposed in the original paper slightly improved the performance.

Table 4.2 shows translation and solving times of TCPP with and without the second type of redundant constraints (conditional mutex constraints) in the domain *airport*, where the number of solved problems decreased the most. We can see that both translation and solving times increased with the redundant constraints and the differences in solving times were more significant. However, in the last three problems in the table, the second model could not translate the problems as the files with problem descriptions were too large (the failure was manifested by the error *Symbol table overflow*).

Based on these results, none of these three types of redundant constraints could significantly improve the performance on the tried domains. Extending the model by additional constraints sometimes increased the total time of translation and solving. Especially, the conditional mutex constraints worsened the performance significantly in some domains.

4.1.3 Summary

We had to slightly modify TCPP to overcome limitations regarding widths of table constraints in descriptions of the tested planning problems. None of the suggested redundant constraints proved to be useful in most domains. Some of them even significantly decreased the performance in some domains. Therefore, for comparison with the other models, presented in section 4.4, we have chosen the version with limited width of transition tables extended only by the mutexes proposed by the Fast Downward translator as proposed in the original paper.

Table 4.3 shows comparison of this version with the results of the final version of TCPP presented by Ghooshchi et al. [2017]. We used all domains listed in the beginning of this chapter. In contrast with our experiments, the authors of the paper performed their experiments with the memory limit of 4 GB and with a different computer. We assume that this could be, apart from using a different solver, the reason why we solved in most of this domains, except for the domain *grid*, more problems within the same time limit.

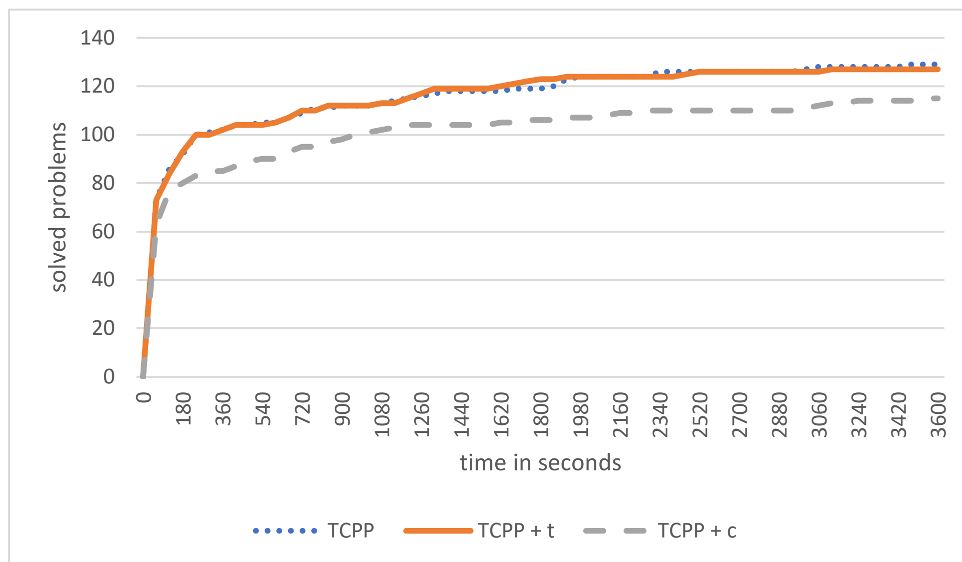

Figure 4.1: This graph shows the number of problems solved with TCPP in time given in seconds. *TCPP* is TCPP with limited width of transition tables, *TCPP + t* is *TCPP* with the constraints describing transitions over a longer time horizon (encoded by positive table constraints), *TCPP + c* is *TCPP* with the constraints describing which values of two variables cannot be set during one time step based on their values at the previous time point (conditional mutex constraints).

Solving and translation times in the domain *airport*				
problem	translation		solving	
	TCPP	TCPP + c	TCPP	TCPP + c
p01	0	0	0	2
p02	1	0	0	1
p03	0	1	0	3
p04	1	4	2	116
p05	1	3	3	31
p06	4	10	3	48
p07	3	6	3	44
p08	10	16	59	713
p09	8	23	150	2782
p10	1	4	3	108
p11	2	5	2	43
p12	4	10	5	59
p13	5	11	4	66
p14	5	23	91	975
p15	6	23	19	303
p16	12	31	178	-
p17	15	47	363	-
p18	24	53	891	-
p19	16	62	524	-
p20	38	66	1421	-
p21	73	-	406	-
p22	147	-	837	-
p36	132	-	1066	-

Table 4.2: Compares solving and translation times (rounded to seconds) of the original TCPP and TCPP with conditional mutex constraints (*TCPP + c*) in the domain *airport*. Translation time includes the time needed to produce a Picat file with the description of the model (see subsection A.2.1 for description of file formats) and the time needed to compile this file. Solving time is the sum of solving times for all makespans (until the problem is solved). These times do not include calls of programs, loading modules and decoding of actions. The table contains only problems that were solved with at least one model. A dash means that the problem was not solved or translated.

domain	total	original TCPP	our TCPP
airport	50	21	**22**
grid	5	**2**	1
miconic	150	29	**58**
storage	30	10	**16**
tpp	30	10	**28**
zenotravel	20	12	**16**

Table 4.3: Number problems solved within the time limit of 60 minutes with the model TCPP. Compares our results with the results presented in the original paper.

4.2 $R^2\exists$-Step encoding

4.2.1 Implementation

In this section we demonstrate efficiency of different ways of encoding the relaxed relaxed ∃-step planning semantics ($R^2\exists$-Step) proposed by Balyo [2013]. As we can see in the second numeric column of Table 4.4, almost no problems could be solved with the original encoding using auxiliary variables in chain constraints. This encoding requires too many variables and constraints, and therefore it is not usable with Picat and its SAT solver. In unsolved instances, the failure was usually manifested as *domain_error*. Removing auxiliary constraints from the model (see subsubsection 3.2.1.2) allows us to solve significantly more problems as we can observe especially in the domains *miconic*, *airport*, *storage* and *zenotravel*.

By replacing all constraints by table constraints, the number of solved problems increases only in the domain *pegsol*. In this domain, the previous models failed to solve any of the problems due to memory issues (the solver ran out of memory during the first few steps). For describing the evolution of each value during each time step by a regular automaton, we used the version of regular automata where the input alphabet corresponds to possible transitions to minimize the size of the input alphabet (see subsubsection 3.2.1.3). Solely by using regular automata, we cannot improve the performance in any domain. As the results of the encoding with table constraints and the encoding with regular automata were similar, we used regular automata as the basis of the next versions of the model since it requires fewer constraints. However, in the domain *airport*, the most suitable encoding is the description using table constraints only to replace chain constraints. In this domain, we solve less problems the model with table constraints replacing all original constraints or with the model with regular automata than with the model which replaces only chain constraints by table constraints. Solving times of the two less successful encodings were longer. This issue could not be solved by any presented modification of this encoding.

In some domains (*miconic*, *storage* and *zenotravel*), some SAS+ state variables have many possible transitions in various actions; as a result, the resulting automata are too large to be accepted by Picat in descriptions of the tested problems, which is again manifested by the *domain_error*. This issue can be resolved by limiting the width of transition matrices by splitting actions to more automata (see paragraph 3.2.1.3.1). Since the performance in the other domains is not affected by this modification and this model provides the highest numbers of solved instances in most of the tried domains except for *airport* and *pegsol*, we use this encoding in all following experiments. Figure 4.2 shows how the number of solved problems increases with solving time. In this figure, we can clearly see the dominance of the final version of the encoding with regular automata with limited width of transition matrices.

4.2.2 Ranking

In this section, we evaluate the effect of modifying ranking by adding disabling edges to the enabling graph (see subsection 3.2.2). The motivation for this alteration was to capture relations between actions which are not in described in the enabling graph. As we can see in the fifth and sixth numeric column of

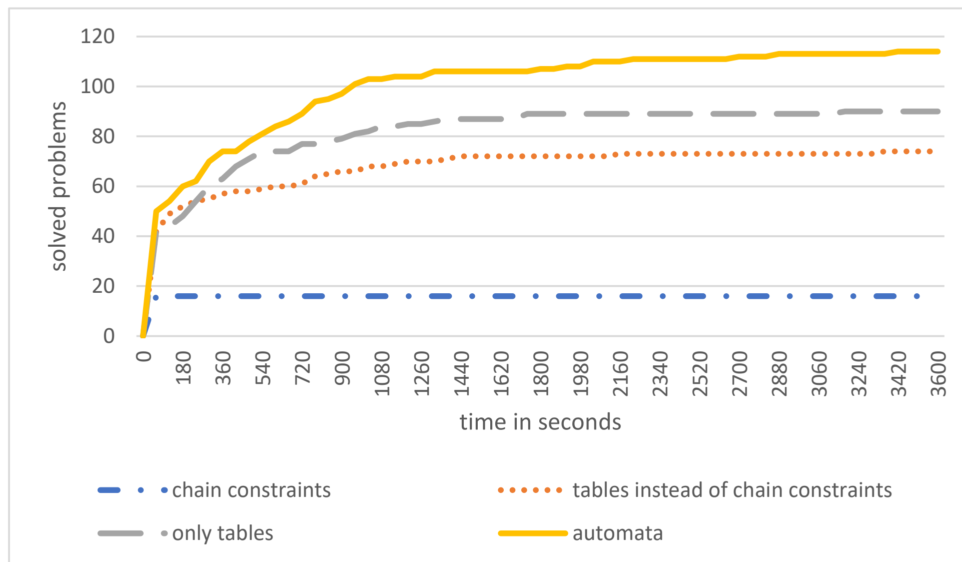

Figure 4.2: This graph shows the number of problems solved with the $R^2\exists$-Step encoding in time given in seconds. The original $R^2\exists$-Step encoding with chain variables and chain constrains is denoted by *chain constrains*, *tables instead of chain constraints* represents the $R^2\exists$-Step encoding with chain constraints replaced by table constraints, *only tables* is the $R^2\exists$-Step encoding with all constraints replaced by tables and *automata* represents the encoding with regular automata with limited width of transition matrices.

$R^2\exists$-Step encoding									
domain	total	orig	t	ta	r	rl	rld	rldm	rldmv
airport	50	0	**20**	16	15	15	14	15	14
grid	5	0	0	0	0	**1**	0	0	0
miconic	150	15	35	35	35	48	58	-	**60**
pegsol	20	0	0	**20**	19	19	18	-	18
storage	30	1	12	12	12	16	**17**	**17**	**17**
zenotravel	20	0	7	7	7	15	15	-	**16**
total	275	16	74	90	88	114	122	123	**125**

Table 4.4: Comparison of modifications of the $R^2\exists$-Step encoding. Number of problems solved within 1 hour for the original $R^2\exists$-Step encoding (o), $R^2\exists$-Step encoding with table constraints replacing chain variables and chain constraints (t), $R^2\exists$-Step encoding with table constraints replacing all constraints (ta), $R^2\exists$-Step encoding with regular automata (r), $R^2\exists$-Step encoding with regular automata with limited transition matrix width (*rl*), *rl* with ranking found in an enabling graph with added disabling edges (*rld*), *rld* with the Fast Downward mutexes (*rldm*), *rldm* with constraints describing possible values of variables in the next time step encoding after executing each action (*rldmv*). In the column *rldm*, the dash is used for domains where Fast Downward does not produce mutexes and the value would be equal to the value in the column *rld*.

Table 4.4, this version is more suitable for the domain *miconic* as the number of solved instances increases considerably. In the other domains, the difference in the performance of these two rankings is negligible. In Figure 4.3, we can see how the new ranking slightly outperforms the original ranking.

4.2.3 Redundant constraints

In this section, we evaluate the effect of redundant constraints. As a base, we used the encoding by regular automata with reduced width and ranking based on enabling graph with added disabling edges. The last two columns of Table 4.4 contain the number of solved instances of the model extended by the mutexes produced by Fast Downward and the model with Fast Downward mutexes and also the constraints describing possible values of variables after applying each action (see subsubsection 3.2.3.2).

In the last but one column of Table 4.4, there are results of the model extended by the mutexes produced by the Fast Downward translator (only for domains, where the mutexes were produced). Only in the domain *airport*, we can solve one more problem with these mutexes. In the last column, there are results with both types of redundant constraints. In the domains *miconic* and *zenotravel*, these constraints slightly improved the performance, while in the domain *airport*, the performance slightly decreased. In the other domains, the performance was not affected. In Figure 4.4, we can see that the performance with the second type of redundant constraints is very similar to the performance without them.

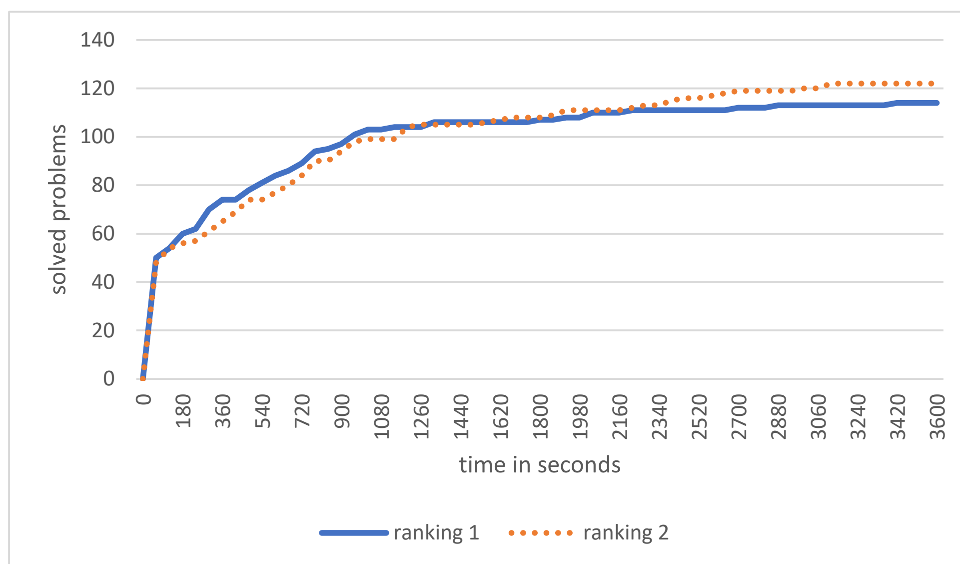

Figure 4.3: This graph shows the number of problems solved with the $R^2\exists$-Step encoding with different ranking of actions. The original ranking based on enabling graphs is represented by *ranking 1* and the ranking with added disabling edges where the search starts from the vertex with the lowest indegree is represented by *ranking 2*.

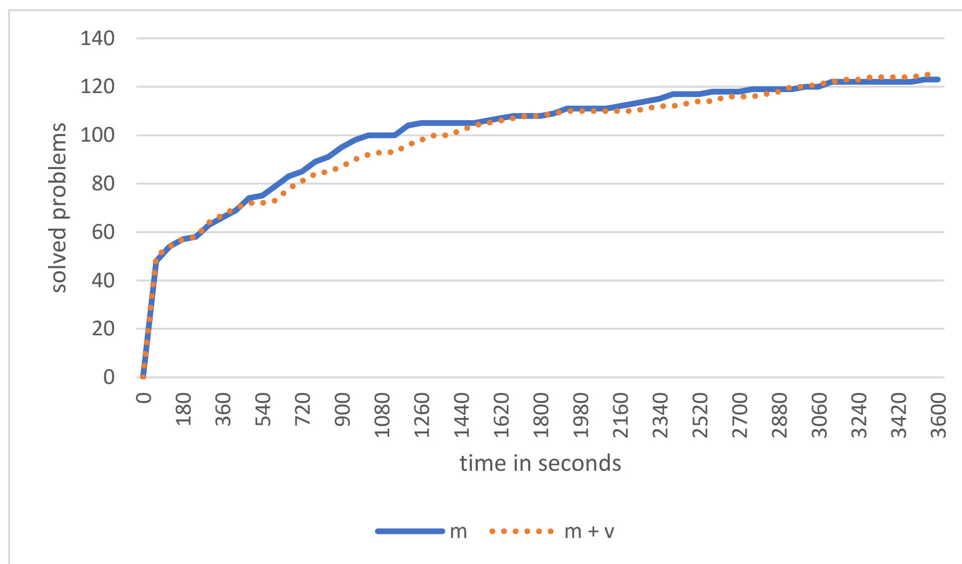

Figure 4.4: This graph shows the number of problems solved with the $R^2\exists$-Step encoding only with Fast Downward mutexes (m) and with both Fast Downward mutexes and the constraints describing remaining available values of state variables after execution of actions (m + v).

4.2.4 Summary

Since the performance of the original constraints and variables was very low in our settings, we modified the encoding using table constraints or regular automata. Our final version of the $R^2\exists$-Step encoding uses regular automata with limited sizes of transition tables to describe the evolution of values of state variables during one time step. We could improve the total number of problems solved within the limit of 60 minutes by adding disabling edges to the enabling graph that we use to find the ranking of actions. However, in some domains, the performance slightly decreased. We could also slightly increase the number of solved problems by adding redundant constraints directly enumerating values of state variables that are possible in the next step after applying an action.

In Table 4.5, we compared our results with the results presented by Balyo [2013]. We used the domains listed in the beginning of this chapter. Firstly, we used our version with regular automata with limited width of transition matrices, disabling edges and both types of mutexes as this version was the most successful in the previous experiments. However, we could solve only 10 instances in the domain *woodworking*. We observed that in this domain, the redundant constraints decreased the performance significantly; as a result, we decided to use the encoding without the new type of redundant constraints only with the mutexes produced by Fast Downward. We used this version also for comparison with the other models in section 4.4. Nevertheless, the authors of the original paper used a different computer with a different processor and with less memory. They also used only 20 problems in the domain *storage*. Moreover, the authors used the time limit of 30 minutes for translation and additional 30 minutes for solving. As we did not measure these times separately, we used a total limit of 30 minutes for our results. Nevertheless, the results are comparable. In 3 domains, the results are the same. In the other domains, our results were slightly worse.

domain	total	original $R^2\exists$-Step	our $R^2\exists$-Step
parcprinter	20	**20**	**20**
pegsol	20	**19**	18
storage	30	**19**	17
woodworking	20	**20**	**20**
zenotravel	20	**15**	**15**

Table 4.5: Number problems solved within the time limit of 30 minutes with the $R^2\exists$-Step encoding. Compares our results with the results presented in the original paper.

4.3 Reinforced Encoding

In this section, we compare different versions of the Reinforced Encoding proposed by Balyo et al. [2015]. The results are shown in Table 4.6.

4.3.1 Implementation

In this section, we evaluate performance of different implementations of the Reinforced Encoding. As we can see in Table 4.6, using table constraints for describing the relations between transitions and values of variables in two consecutive time points and using equivalences to describe the relations between actions and transitions (see subsubsection 3.3.1.2) does not improve the performance. On the contrary, as we can see in the domain *pegsol*, the number of solved problems is lower after introducing the table constraints. Figure 4.5 shows the number of problems solved with increasing time with different types of constraints. We can see that the results are very similar; however, the model with the original constraints has the best performance.

Reinforced Encoding												
domain	total	o	t	te	tea	oe	oa	oea	oal	oalm	oalms	oalmi
airport	50	11	11	11	**20**	11	**20**	**20**	**20**	**20**	**20**	**20**
grid	5	0	0	0	0	0	0	0	0	**1**	**1**	**1**
miconic	150	40	39	39	40	38	40	40	**53**	-	52	50
pegsol	20	14	11	13	**14**	**14**	**14**	**14**	**14**	-	**14**	13
storage	30	12	12	12	**16**	12	**16**	**16**	**16**	**16**	**16**	**16**
zenotravel	20	7	7	7	7	7	7	7	**16**	-	15	15
total	275	84	80	82	97	82	97	97	119	**120**	118	115

Table 4.6: Comparison of modifications of the Reinforced Encoding. Reinforced Encoding (o), Reinforced Encoding with table constraints describing relations between transitions and values of variables (t), t with equivalences describing relations between transitions and actions (te), te with sums ensuring that conflict actions are not executed simultaneously (tea), the original Reinforced Encoding with equivalences describing relations between transitions and actions (oe), the original Reinforced Encoding with conflict actions encoded as sums of action variables (oa), oe with conflict actions encoded as sums of action variables (oea), oa with limited length of sums (oal), oal with the Fast Downward mutexes ($oalm$), where a dash denotes rows where Fast Downward does not produce mutexes, $oalm$ with transition sequencing constraints ($oalms$), $oalm$ with precomputed initial makespan ($oalmi$).

4.3.2 Encoding of conflict actions

Changing the encoding of conflict actions (see subsection 3.3.2) increased the number of solved problems especially in the domain *storage*. The redefinition of action conflict, allowing more actions to be executed in one time step, in comparison to the original model allowing only independent actions to be executed in parallel, decreases the makespan and decreases the time needed to solve the planning problem. Similarly to TCPP and the $R^2\exists$-Step encoding, we had to overcome limitations of Picat regarding the length of sums (see subsubsection 3.3.2.1). The fourth from the last column of Table 4.6 contains the number of solved problems after this slight modification. We can see that in the domains *miconic* and *zenotravel*, many problems were not solved only due to this limitation. In Figure 4.6, which shows the number of problems solved with increasing

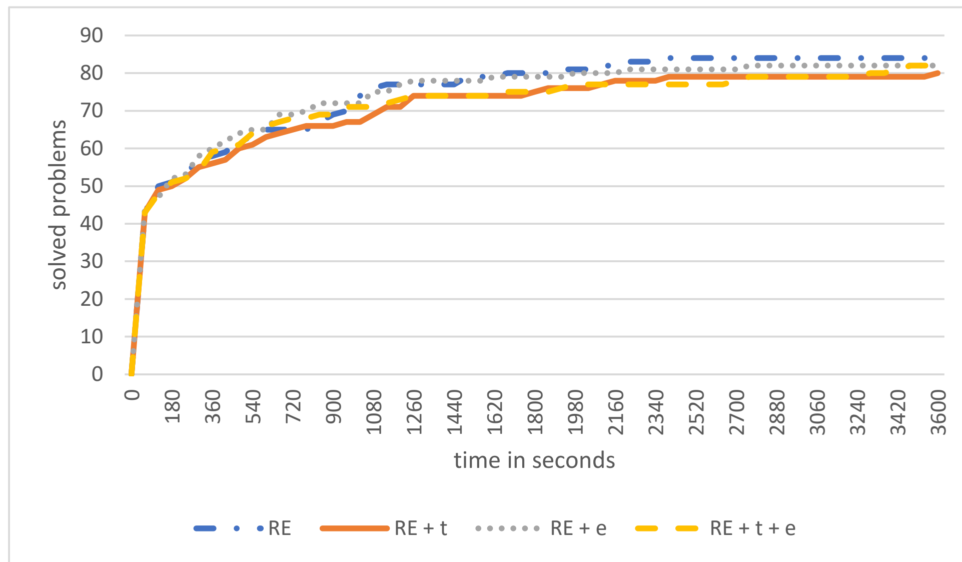

Figure 4.5: This graph shows the number of problems solved with the Reinforced Encoding in time given in seconds. *RE* is the original Reinforced Encoding, *RE + t* represents the Reinforced Encoding with table constraints describing the relations of transitions and values of variables, *RE + e* represents the Reinforced Encoding with relations between transitions and actions described by equivalences, *RE + t + e* is the Reinforced Encoding with both table constraints and equivalences.

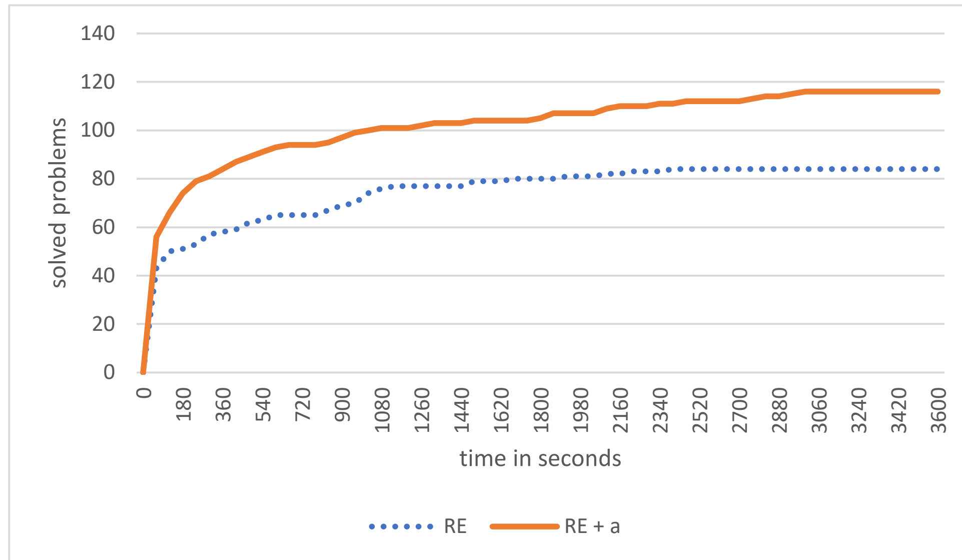

Figure 4.6: This graph shows the number of problems solved with the Reinforced Encoding with the original action conflict constraints (RE) and with the modified definition of parallel step with limited length of action lists in sums ($RE + a$) in time given in seconds.

time, we can see how the encoding with the modified definition of parallel step outperforms the original encoding.

Table 4.8 and Table 4.9 compare makespans of the Reinforced Encoding with different definitions of parallel conflict in the domains *storage* and *airport*. Makespans of the second version are the same as makespans of TCPP as we use the same definition of parallel step. In these domains, the number of problems solved with TCPP and with the Reinforced Encoding with the modified definition of action conflict is very similar.

4.3.3 Redundant constraints

The third from the last column of Table 4.6 shows that the mutexes produced by the Fast Downward translator could slightly increase the number of solved problems in some domains. As we can see in the last but one column of Table 4.6, the transition sequencing constraints (see subsubsection 3.3.3.2) did not have a positive effect on the performance and the performance even slightly decreased in some domains. In Figure 4.7, we can see that the performance is very similar with the redundant constraints.

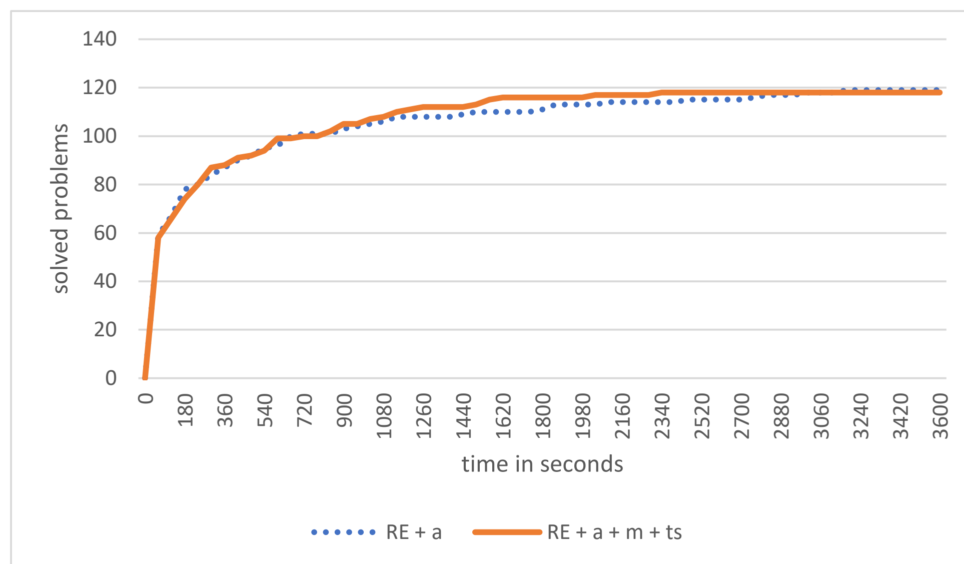

Figure 4.7: This graph shows the number of problems solved with the Reinforced Encoding with the modified parallel step definition ($RE + a$) and $RE + a$ extended by both Fast Downward mutexes and transition sequencing constraints $RE + a + m + ts$ in time given in seconds.

4.3.4 Precomputing of the initial makespan

Finally, we studied the effect of computing of the initial makespan which is a part of the model TCPP as proposed by Ghooshchi et al. [2017]. The results are shown in the last column of Table 4.6. As we can see, this modification did not improve the performance in any domain and in some domains, the performance decreased. It seems that the preprocessing consumes too much time while the problems with short makespans in the first few iterations contain few variables and constraints; therefore, these problems can be solved fast and skipping them did not prove to be beneficial. Moreover, in the domain *miconic*, where the performance was the most decreased by this preprocessing, the initial makespans were still equal to 1 as the goal variables indicating if passengers were served have only two values. In other domains except for the domain *airport*, usually at most 3 iterations were eliminated. On the other hand, in the domain *airport*, more than 20 iterations were eliminated in some problems which decreased the total times (solving and translation times) of some instances as we can see in Table 4.7. For most instances, the solving times were decreased to less than one third. However, even though for the largest problems the preprocessing could remove more than 60 initial steps, more problems still could not be solved due to memory limitations. For domains with large domains of goal variables, we assume that this modification could be useful.

Solving times in the domain *airport* with or without precomputed makespans			
problem	no	yes	makespan
p01	3	2	2
p02	3	2	9
p03	4	2	9
p04	23	15	2
p05	15	4	21
p06	24	7	21
p07	26	5	21
p08	78	28	21
p09	150	49	21
p10	15	13	2
p11	15	4	21
p12	27	8	21
p13	36	7	19
p14	104	38	21
p15	65	15	20

Table 4.7: Comparison of total times (solving and translation times) in seconds of the Reinforced Encoding for the solved instances in the domain *airport*. The first numeric column contains solving times where the initial makespan was equal to 1 (labelled by *no*), the next column (labelled by *yes*) contains solving times with precomputed makespans and the last column contains the precomputed initial makespans.

Makespans of Reinforced Encoding in the domain *storage*		
problem	action conflict 1	action conflict 2
p01	3	3
p02	3	3
p03	3	3
p04	8	8
p05	8	6
p06	8	6
p07	14	14
p08	12	8
p09	11	7
p10	18	18
p11	17	11
p12	-	9
p13	18	18
p14	-	11
p15	-	9
p16	-	11

Table 4.8: Makespans of problems solved within an hour in the domain *storage* by the original Reinforced Encoding (action conflict version 1) and the encoding with the modified definition of parallel conflict inspired by TCPP (action conflict version 2). A dash denotes a problem that was not solved.

4.3.5 Summary

We attempted to reformulate the Reinforced Encoding in several ways. However, the most efficient version was the one with the original constraints. Extending the model by transition sequencing constraints also did not prove to be useful. Nevertheless, we successfully improved the performance by using a different definition of a parallel step which was used by Ghooshchi et al. [2017]. Our final version of the encoding contains the original constraints except for the action conflict actions. Instead of them, we used the different definition of a parallel step encoded as sums of action variables with limited length of lists of action variables. We also extended the model by the mutexes produced by the Fast Downward translator.

In Table 4.10, we compared our results with the results presented by Balyo et al. [2015]. However, the authors used a different computer with less memory. In their paper, they present the number of problems solved in each domain in the time limit of 30 minutes. However, they measured only the time used for solving and not for translation. We measured the total time. For comparing the models, we used the domains listed in the beginning of the chapter. In the domain *parcprinter*, the number of solved problems was the same and in the domain *pegsol*, our result was slightly better. Nevertheless, in the domain *woodworking*, we solved only half of the problems solved by the original model. In this domain, the Reinforced Encoding failed to solve many problems due to the *domain_error* produced by Picat. In the previous experiments, this error was usually caused sizes of some constraints. We assume that this error could be

Makespans of Reinforced Encoding in the domain *airport*		
problem	action conflict 1	action conflict 2
p01	8	8
p02	9	9
p03	17	9
p04	20	20
p05	21	21
p06	41	21
p07	41	21
p08	-	26
p09	-	27
p10	18	18
p11	21	21
p12	39	21
p13	37	19
p14	-	26
p15	-	22
p16	-	27
p17	-	28
p18	-	31
p19	-	30
p20	-	32

Table 4.9: Makespans of problems solved within an hour in the domain *airport* by the original Reinforced Encoding (action conflict version 1) and the encoding with the modified definition of parallel conflict inspired by TCPP (action conflict version 2). A dash denotes a problem that was not solved.

resolved by identifying these constraints and reducing their size using additional variables.

domain	total	original Reinforced Encoding	our Reinforced Encoding
parcprinter	20	**20**	**20**
pegsol	20	10	**14**
woodworking	20	**20**	10

Table 4.10: Number problems solved within the time limit of 30 minutes with the Reinforced Encoding. Compares our results with the results presented in the original paper.

4.4 Comparison of models

In this section we compare the number of problems solved within one hour by TCPP, $R^2\exists$-Step encoding and Reinforced Encoding. For each encoding, we firstly used the version with the highest total number of solved problems. For TCPP, we used the version with limited width of transition tables and the mutexes produced by the Fast Downward translate component. For the $R^2\exists$-Step encoding, we used

the encoding using regular automata with limited width of transition matrices, the ranking found in the enabling graph with added disabling edges and both types of mutexes. However, in the domain *woodworking*, we could solve only 10 problems with this version of the $R^2\exists$-Step encoding; therefore, we decided to use only the mutexes produced by Fast Downward. With this version, we could solve all 20 problems in this domain. Also in the domain *tpp*, with could solve only 20 problems with both types of mutexes, while without the second type of mutexes, we could solve 24 problems. For the Reinforced Encoding, we used the version with the original constraints, with the modified definition of parallel step with sums with limited length and with mutexes produced by the Fast Downward translator. The results are shown in Table 4.11 and Figure 4.8 shows the number of problems solved with increasing time.

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

Table 4.11: Number of problems solved within one hour. Compares TCPP with limited width of transition tables and Fast Downward mutexes, $R^2\exists$-Step encoding with regular automata with limited width of transition matrices, with enabling graph with added disabling edges and with Fast Downward mutexes and Reinforced Encoding with table constraints replacing chain constraints and a modified definition of a parallel step encoded by sums with limited length.

As we can see in Table 4.11, all three encodings solved a similar number of problems in the domains *grid*, *parcprinter*, *storage* and *zenotravel*. In the domain *miconic*, the results were comparable. In the domain *woodworking*, the Reinforced Encoding solved significantly less problems than both of the other encodings. Here, Reinforced Encoding failed to solve many problems due to the *domain_error* produced by Picat.

In the domain *airport*, both TCPP and Reinforced Encoding solved significantly more problems than the $R^2\exists$-Step encoding, whose solving times were too long. As we already mentioned in subsection 4.2.1, the domain *airport* was the only domain where the performance of the $R^2\exists$-Step encoding worsened by replacing the original constraints by table constraints or automata. If we used the original constraint model with only chain constraints replaced by tables, the number of solved problems would be 20 (see Table 4.4), which is similar to the other encodings.

In the domain *pegsol*, where both TCPP and the Reinforced Encoding fail to solve some problems solved by the $R^2\exists$-Step encoding in the given time limit, we

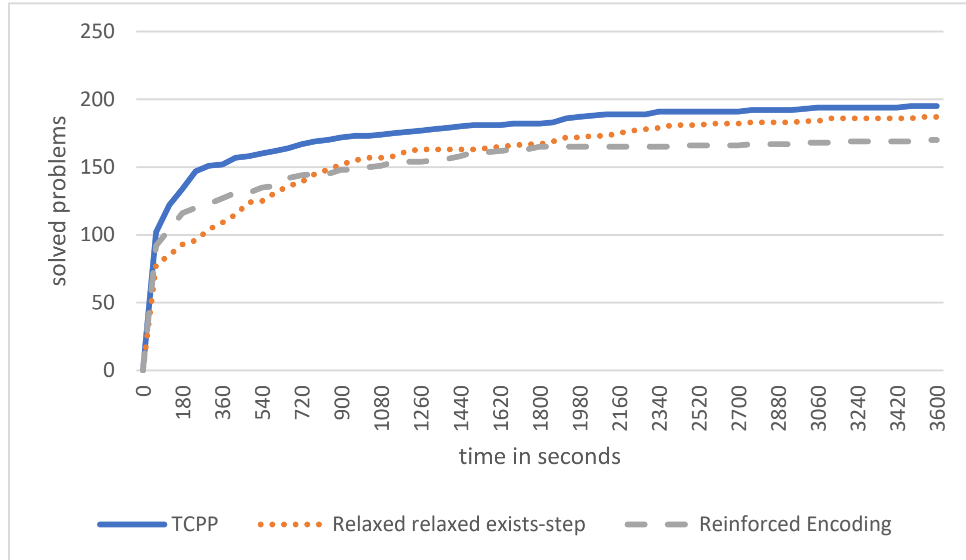

Figure 4.8: This graph shows the number of problems solved with TCPP, $R^2\exists$-Step encoding and Reinforced Encoding in time given in seconds.

assume that the $R^2\exists$-Step encoding solved more problems than the other encodings due to the reduced makespan. Table 4.16 compares makespans of TCPP and the $R^2\exists$-Step encoding for solved problems in the domain *pegsol*. The number of parallel steps corresponds to the number of the solver calls. In the $\forall$-Step semantics of TCPP (and also the final version of the Reinforced Encoding), the plan is at least as long as in the $R^2\exists$-Step semantics because each parallel step in the $\forall$-Step semantics is valid also in the $R^2\exists$-Step semantics as the actions in a parallel step do not destroy preconditions of other parallel actions. The plans produced by TCPP in the domain *pegsol* are more than twice as long as the plans produced by the $R^2\exists$-Step encoding.

In Table 4.15, we compare makespans in the domain *airport*. The number of solver calls of TCPP can be decreased by computing the minimum makespan and eliminating some of the initial solver calls (see section 3.1). In the domain *airport*, this preprocessing can decrease the number of steps significantly and the final number of solver calls is often lower than the makespan in the $R^2\exists$-Step encoding. However, the number of eliminated steps depends on domains of the goal variables. When the domains are small, the distances in the DTGs are short and the initial makespan is small. In the domain *pegsol* (see Table 4.16), the initial makespan is always equal to 2.

Nevertheless, less solver calls does not always imply shorter solving time, as we can see in Table 4.18, which compares solving times of solved problems in the domain *pegsol*. For some problems, the solving time is considerably lower for TCPP although the number of solver calls is significantly higher. Table 4.17 contains solving times for the solved problems in the domain *airport*. In this

domain, solving times are significantly higher for the R^2∃-Step encoding for most problems including the problems where the number of solver calls is lower.

In the domain *tpp*, TCPP solved more instances than both the Reinforced Encoding and the R^2∃-Step encoding. In the unsolved problems, these two encodings spent all time in the first iteration. When we tried to solve the problems from this domain with the R^2∃-Step encoding with more redundant constraints (the redundant constraints specifying available values of variables after executing actions), and we could solve only 20 instances, which brought the results of the R^2∃-Step encoding closer to the results of the Reinforced Encoding. Based on this observation that the R^2∃-Step encoding performed worse with more constraints, we assume that also the Reinforced Encoding might have solved less problems due to a higher number of constraints.

Number of variables per step in the domain *zenotravel*			
problem	TCPP	R^2∃-Step	Reinforced Encoding
p01	8	135	137
p02	8	135	137
p03	16	294	298
p04	16	294	298
p05	16	478	480
p06	18	495	498
p07	18	495	498
p08	24	1125	1131
p09	26	1156	1163
p10	28	1187	1195
p11	26	1588	1589
p12	28	1625	1627
p13	30	1662	1665
p14	38	6839	6798
p15	48	10173	10093
p16	50	13580	13465
p17	60	18130	17965
p18	60	22405	22200
p19	70	27635	27890
p20	70	32815	33170

Table 4.12: Comparison of the number of variables (including all auxiliary variables) per each time step except for the last step (in the last time step, the R^2∃-Step encoding would contain only state variables) in TCPP with reduced width of transition tables, the R^2∃-Step encoding with regular automata with reduced width of transition matrices and the Reinforced Encoding with the modified definition of the parallel step encoded as sums of action variables with limited length in the domain *zenotravel*.

In Table 4.12, we compare the number of variables in each time step for the last version of all encodings in the domain *zenotravel*. Table 4.13 contains for each encoding the number of variables in the last step of the solved problems. Since TCPP contains only state variables, this encoding has significantly less variables

Number of variables in the last step in the domain *zenotravel*			
problem	TCPP	$R^2\exists$-Step	Reinforced Encoding
p01	16	139	137
p02	48	409	685
p03	96	596	1490
p04	96	596	1490
p05	96	1442	2400
p06	96	999	2490
p07	126	999	2988
p08	144	2262	5655
p09	182	2325	6978
p10	196	4777	7170
p11	182	3189	9534
p12	196	3264	9762
p13	240	5001	11655
p14	266	20536	40788
p15	384	20370	70651
p16	400	-	94255

Table 4.13: Comparison of the number of variables of TCPP, the $R^2\exists$-Step encoding and the Reinforced Encoding in the last step in the solved problems from the domain *zenotravel*. Each column contains the number of variables in the last time step. A dash means that the problem was not solved. Problems which are not in the table were not solved with any encoding.

than the other encodings. Nevertheless, the effect of having less variables is not clearly visible here as the $R^2\exists$-Step encoding solves only one problem less than TCPP and the Reinforced Encoding, having only slightly more variables than the $R^2\exists$-Step encoding, solves the same number of problems as TCPP.

Based on the aggregate results, TCPP solved the highest total number of problems, the $R^2\exists$-Step encoding was in the second place and the Reinforced Encoding was in the last place. If we compare results for individual domains, TCPP was in the first place in 6 domains, the $R^2\exists$-Step encoding in 5 domains and the Reinforced Encoding only in 3 domains. The $R^2\exists$-Step encoding is strictly better than TCPP in 3 domains, whereas TCPP is strictly better than the $R^2\exists$-Step encoding in 4 domains. The Reinforced Encoding is not strictly better than TCPP in any domain; however, in comparison with the $R^2\exists$-Step encoding, the Reinforced Encoding is strictly better in 3 domains. The Reinforced Encoding did not strictly outperform both of the other encodings simultaneously in any domain. In most domains, the results of the Reinforced Encoding are closer to TCPP than to the $R^2\exists$-Step encoding. We assume that the reason is that these two encodings use the same parallel semantics. As the Reinforced Encoding never dominated TCPP, the results imply that with the $\forall$-step parallel semantics, it is beneficial to reduce the number of constraints and variables in the model. In some domains, the model with the $R^2\exists$-Step semantics dominated the $\forall$-step semantics; nevertheless, in other domains, TCPP was more successful. Since TCPP also solved the highest total number of problems while having a similar

number of constraints as the final version of the R^2∃-Step encoding, we conclude that reducing the number of variables may be more beneficial than reducing the number of SAT solver calls.

Finally, we add a comparison of our models with 2 planners competing in IPC 2014. The first one is a SAT-based planner Madagascar (Rintanen [2014]) and the second one is the planner YAHSP3 (Vidal [2014]), a state-space search based heuristic planner. We used the results presented by Ghooshchi et al. [2017] and we show them along with our results in Table 4.14. However, the authors limited the memory during the experiments and they used a different computer. Regarding the total number of solved problems, both Madagascar and YAHSP3 solved more problems than any of our model. In the domains *miconic*, *storage*, *tpp* and *zenotravel*, some of our models had the same or a better performance than Madagascar. We assume that it was caused by using more memory; nevertheless, even with more memory, we could not reach the performance of the planner YAHSP3 in any domain.

domain	total	TCPP	R^2∃-Step	RE	Madagascar	YAHSP3
airport	50	23	15	20	**45**	**45**
grid	5	1	0	1	2	**5**
miconic	150	58	58	53	49	**150**
storage	30	16	17	16	14	**23**
tpp	30	28	24	20	28	**30**
zenotravel	20	16	15	16	15	**20**
total	286	142	129	126	153	**273**

Table 4.14: The number of problems solved within the time limit of 1 hour with the encodings TCPP, R^2∃-Step encoding and Reinforced Encoding (*RE*) and by the planners Madagascar and YAHSP3.

Makespans in the domain *airport*			
problem	TCPP	TCPP + initial makepan	$R^2\exists$-Step
p01	9	8	6
p02	10	2	5
p03	10	2	6
p04	21	20	15
p05	22	2	13
p06	22	2	13
p07	22	2	13
p08	27	7	18
p09	28	8	18
p10	19	18	9
p11	22	2	13
p12	22	2	14
p13	20	2	13
p14	27	7	-
p15	23	3	16
p16	28	8	-
p17	29	9	-
p18	32	12	-
p19	31	11	-
p20	33	13	-
p21	54	2	-
p22	54	2	-
p36	61	2	-

Table 4.15: Comparison of makespans of the solved problems corresponding to the number of the solver calls of TCPP and the $R^2\exists$-Step encoding in the domain *airport*. The column *TCPP + initial makespan* corresponds to TCPP with the initial makespan computed from distances from the initial to the goal vertices in the DTGs of state variables (some of the initial solver calls are eliminated).

Makespans in the domain *pegsol*			
problem	TCPP	TCPP + initial makespan	$R^2\exists$-Step
p01	25	24	10
p02	22	21	9
p03	23	22	9
p04	27	26	12
p05	24	23	8
p06	22	21	8
p07	24	23	9
p08	27	26	12
p09	26	25	10
p10	23	22	8
p11	24	23	9
p12	25	24	9
p13	22	21	9
p14	26	25	8
p15	21	20	9
p16	-	-	9
p17	-	-	10
p18	-	-	-
p19	-	-	12
p20	-	-	-

Table 4.16: Comparison of makespans of the solved problems corresponding to the number of the solver calls of TCPP and the $R^2\exists$-Step encoding in the domain *pegsol*. The column *TCPP + initial makespan* corresponds to TCPP with the initial makespan computed from distances from the initial to the goal vertices in the DTGs of state variables (some of the initial solver calls are eliminated).

Solving times in the domain *airport* in seconds		
problem	TCPP	$R^2\exists$-Step
p01	5	4
p02	4	4
p03	5	7
p04	18	272
p05	11	401
p06	20	666
p07	18	745
p08	113	2636
p09	223	2930
p10	17	91
p11	11	714
p12	22	1195
p13	25	896
p14	140	-
p15	41	3011
p16	269	-
p17	538	-
p18	1282	-
p19	758	-
p20	1944	-
p21	861	-
p22	1919	-
p36	2318	-

Table 4.17: Comparison of solving times (in seconds) of the problems solved with TCPP and with the $R^2\exists$-Step encoding in the domain *airport*. A dash denotes a problem that was not solved.

Solving times in the domain *pegsol* in seconds		
problem	TCPP	$R^2\exists$-Step
p01	696	1634
p02	225	867
p03	646	936
p04	374	1469
p05	1339	709
p06	412	635
p07	1832	1033
p08	358	1540
p09	2971	1434
p10	639	603
p11	1090	1239
p12	1034	982
p13	216	835
p14	1250	645
p15	217	750
p16	-	1316
p17	-	1480
p18	-	-
p19	-	1805
p20	-	-

Table 4.18: Comparison of solving times (in seconds) of the problems solved with TCPP and with the $R^2\exists$-Step encoding in the domain *pegsol*. A dash denotes a problem that was not solved.

Conclusion

In this thesis, we implemented three of the recently proposed compilation-based planning approaches: TCPP (Ghooshchi et al. [2017]), the $R^2\exists$-Step encoding (Balyo [2013]) and the Reinforced Encoding (Balyo et al. [2015]). We suggested modifications of the implementation and we extended the models by redundant constraints. We compared different versions of the encodings and also different encodings with each other on benchmarks from international planning competitions.

For the $R^2\exists$-Step encoding, modifications of the encoding decreasing the number of variables and constraints were necessary as the performance of the original encoding was very low when it was implemented in Picat and solved by the Picat SAT solver. In the Reinforced Encoding, we could improve the performance by modification of the encoding of parallel conflict. Extending the models by additional constraints mostly did not improve the performance. Redundant constraints could slightly improve the performance only in some domains, while in the other domains, the performance was affected negatively. In the final comparison, we solved the highest number of problems in total with TCPP, which has the lowest number of variables. However, the $R^2\exists$-Step encoding, which needs the lowest number of iterations to find a plan, outperformed TCPP in some domains.

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

Bibliography

The 2002 competition. http://ipc02.icaps-conference.org/, a. Home page of the International Planning Competition 2002. Online. Accessed 12 April 2020.

The 2004 Competition. http://idm-lab.org/wiki/icaps/ipc2004/deterministic/, b. Home page of the deterministic track of the International Planning Competition 2004. Online. Accessed 12 April 2020.

5th international planning competition. http://idm-lab.org/wiki/icaps/ipc2006/deterministic/, c. Home page of the deterministic track of the International Planning Competition 2006. Online. Accessed 12 April 2020.

International Planning Competition 2011. http://www.plg.inf.uc3m.es/ipc2011-deterministic/, d. Home page of the deterministic track of the International Planning Competition 2011. Online. Accessed 12 April 2020.

The 1st International Planning Competition, 1998. http://ipc98.icaps-conference.org/, e. Home page of the International Planning Competition 1998. Online. Accessed 12 April 202.

Picat 2.8, December 2019. Computer software. Available from `http://picat-lang.org`.

Artificial Intelligence Group - University of Basel. Fast Downward benchmark collection. `https://github.com/aibasel/downward-benchmarks`. Online. Accessed 12 October 2019.

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

Tomáš Balyo, Roman Barták, and Otakar Trunda. Reinforced encoding for planning as SAT. *Acta Polytechnica CTU Proceedings*, 2(2):1–7, 2015.

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

Roman Barták. A novel constraint model for parallel planning. In *Twenty-Fourth International Florida Artificial Intelligence Research Society Conference*, January 2011a.

Roman Barták. On constraint models for parallel planning: The novel transition scheme. In *11th Scandinavian Conference on Artificial Intelligence (SCAI)*, volume 227, pages 50–59, January 2011b.

Roman Barták and Daniel Toropila. Reformulating constraint models for classical planning. In *Florida Artificial Intelligence Research Society Conference*, pages 525–530, 2008.

Roman Barták and Daniel Toropila. Revisiting constraint models for planning problems. In *International Symposium on Methodologies for Intelligent Systems*, pages 582–591. Springer, 2009.

Roman Barták and Daniel Toropila. Solving sequential planning problems via constraint satisfaction. *Fundamenta Informaticae*, 99(2):125–145, 2010.

Roman Barták. Binarization of constraints. `https://ktiml.mff.cuni.cz/~bartak/constraints/binary.html`, 1998. Online. Accessed 19 May 2020.

Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. Technical Report CMU-CS-95-221, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, December 1995.

Miquel Bofill, Joan Espasa, and Mateu Villaret. A semantic notion of interference for planning modulo theories. In *Twenty-Sixth International Conference on Automated Planning and Scheduling*, 2016.

Miquel Bofill, Joan Espasa, and Mateu Villaret. Relaxed ∃-step plans in planning as smt. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 563–570. AAAI Press, 2017.

Michael Cashmore, Maria Fox, and Enrico Giunchiglia. Planning as Quantified Boolean Formula. In *20th European Conference on Artificial Intelligence*, volume 242, pages 217–222, 2012.

Michael Cashmore, Maria Fox, Derek Long, and Daniele Magazzeni. A compilation of the full PDDL+ language into SMT. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

Yixin Chen, Ruoyun Huang, Zhao Xing, and Weixiong Zhang. Long-distance mutual exclusion for planning. *Artificial Intelligence*, 173(2):365–391, 2009.

Minh Binh Do and Subbarao Kambhampati. Solving planning-graph by compiling it into CSP. In *The Fifth International Conference on Artificial Intelligence Planning and Scheduling Systems*, 2000.

Fast Downward contributors. Fast Downward 19.06, 2019. Computer Software. Available from `http://www.fast-downward.org/`.

Richard E Fikes and Nils J Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

Robert W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5 (6):345, 1962.

Nils Froleyks, Tomas Balyo, and Dominik Schreiber. Pasar—planning as satisfiability with abstraction refinement. In *Twelfth Annual Symposium on Combinatorial Search*, 2019.

Olivier Gasquet, Dominique Longin, et al. Compact tree encodings for planning as QBF. *Inteligencia Artificial*, 21(62):103–113, 2018.

Malik Ghallab, Craig Knoblock, David Wilkins, Anthony Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden, Scott Penberthy, David Smith, Ying Sun, and Daniel Weld. PDDL – the planning domain definition language. 1998. AIPS-98 Planning Competition Committee.

Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning and acting*. Cambridge University Press, 2016.

Nina Ghanbari Ghooshchi, Majid Namazi, MA Hakim Newton, and Abdul Sattar. Encoding domain transitions for constraint-based planning. *Journal of Artificial Intelligence Research*, 58:905–966, 2017.

Stephan Gocht and Tomáš Balyo. Accelerating sat based planning with incremental sat solving. In *Twenty-Seventh International Conference on Automated Planning and Scheduling*, 2017.

Peter Gregory, Derek Long, and Maria Fox. Constraint based planning with composable substate graphs. In *19th European Conference on Artificial Intelligence*, pages 453–458, 2010.

Peter Gregory, Derek Long, Maria Fox, and J Christopher Beck. Planning modulo theories: Extending the planning paradigm. In *Twenty-Second International Conference on Automated Planning and Scheduling*, 2012.

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

Jörg Hoffmann. The Metric-FF planning system: Translating ”ignoring delete lists” to numeric state variables. *Journal of artificial intelligence research*, 20: 291–341, 2003.

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

Ruoyun Huang, Yixin Chen, and Weixiong Zhang. A novel transition based encoding scheme for planning as satisfiability. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

Henry Kautz and Bart Selman. Blackbox: A new approach to the application of theorem proving to problem solving. In *AIPS98 Workshop on Planning as Combinatorial Search*, volume 58260, pages 58–60, 1998.

Henry Kautz and Bart Selman. Unifying sat-based and graph-based planning. In *Sixteenth International Joint Conference on Artificial Intelligence*, volume 99, pages 318–325, 1999.

Henry Kautz, Bart Selman, and Joerg Hoffmann. Satplan: Planning as satisfiability. In *5th International Planning Competition*, volume 20, page 156, 2006.

Henry A Kautz, Bart Selman, et al. Planning as satisfiability. In *10th European Conference on Artificial Intelligence*, volume 92, pages 359–363. Citeseer, 1992.

Jana Koehler, Bernhard Nebel, Jörg Hoffmann, and Yannis Dimopoulos. Extending planning graphs to an ADL subset. In *European Conference on Planning*, pages 273–285. Springer, 1997.

Adriana Lopez and Fahiem Bacchus. Generalizing GraphPlan by formulating planning as a CSP. In *Eighteenth International Joint Conference on Artificial Intelligence*, volume 3, pages 954–960, 2003.

Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

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

Jussi Rintanen. Madagascar: Scalable planning with SAT. *Proceedings of the 8th International Planning Competition (IPC-2014)*, 21, 2014.

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.

Enrico Scala, Miquel Ramirez, Patrik Haslum, and Sylvie Thiebaux. Numeric planning with disjunctive global constraints via SMT. In *Twenty-Sixth International Conference on Automated Planning and Scheduling*, 2016.

Martin Suda. Property directed reachability for automated planning. *Journal of Artificial Intelligence Research*, 50:265–319, 2014.

Vincent Vidal. YAHSP3 and YAHSP3-MT in the 8th international planning competition. *Proceedings of the 8th International Planning Competition (IPC-2014)*, pages 64–65, 2014.

Thomas Vossen, Michael Ball, Amnon Lotem, and Dana Nau. Applying integer programming to AI planning. *The Knowledge Engineering Review*, 15(1):85–100, 2000.

Martin Wehrle and Jussi Rintanen. Planning as satisfiability with relaxed ∃-step plans. In *Australasian Joint Conference on Artificial Intelligence*, pages 244–253. Springer, 2007.

Dominic J. A. Welsh and Martin B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85–86, 1967.

Wikipedia contributors. Conjunctive normal form — Wikipedia, the free encyclopedia, 2020a. URL `https://en.wikipedia.org/w/index.php?title=Conjunctive_normal_form&oldid=945093526`. Online. Accessed 6 May 2020.

Wikipedia contributors. Constraint satisfaction problem — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Constraint_satisfaction_problem&oldid=944596213, 2020b. Online. Accessed 12 April 2020.

Wikipedia contributors. Boolean satisfiability problem — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Boolean_satisfiability_problem&oldid=945452085, 2020c. Online. Accessed 12 April 2020.

Neng-Fa Zhou and Jonathan Fruhman. A user's guide to Picat. http://retina.inf.ufsc.br/picat_guide, 2020. Online. Accessed 26 March 2020.

A. Documentation on Picat files

This attachment contains a documentation of Picat programs that we used to compile planning tasks to the selected encodings, description of their interfaces and description of formats of the generated auxiliary files (see section B.2).

A.1 Translation from PDDL

The benchmarks used for evaluation in chapter 4 are encoded in PDDL. We used the translator component of the planning system Fast Downward to translate the files to the SAS+ format (see Fast Downward contributors [2019]). The Fast Downward translator outputs for each planning instance a file containing its description in SAS+.

We then translate the planning task description from SAS+ to Picat predicates as this description is convenient for compilation to different models. The translation is achieved by the program `sas_to_picat.pi`. The program reads the output of the Fast Downward translator from the standard input and creates the file `problem_def.pi` in the directory specified as an argument. The program does not support conditional effects and axioms and ignores action costs since we do not use them in any of the implemented model.

The output file contains these predicates:

- `variable(Name, AxiomLayer, Range, Values)`
  - `Name` is the name of the variable in the SAS+ description.
  - `AxiomLayer` is the axiom layer of the variable. We do not use this value.
  - `Range` is the size of the domain of the variable.
  - `Values` is a list containing names of all values of this variable.
- `mutex_defined` is a predicate which is defined as *true* if Fast Downward produced mutexes for the planning task and *false* otherwise.
- `mutex(Variable1, Value1, Variable2, Value2)`
  - `Variable1`, `Variable2` are names of variables in mutex.
  - `Value1` and `Value2` are names of values of that cannot be assigned to the corresponding variables at the same time.
- `init(Variable, Value)` defines the initial state of a variable. `Value` is the value of `Variable` in the initial state.
- `goal(Variable, Value)` defines a goal value. `Value` is the goal value of `Variable`.
- `operators_defined` is a predicate which is defined as *true* if there are any actions defined in the SAS+ output (should be *true* for all correctly defined problems).

- `operator(OperatorName)` defines the operator. `OperatorName` is the name of one of the operator defined in the SAS+ file.

- `precond(Operator, Variable, Value)` defines the precondition `Variable = Value` of `Operator`.

- `effect((Operator, Variable, Value)` defines the effect `Variable = Value` of `Operator`.

A.2 Translation from SAS and solving

Predicates in the file `problem_def.pi` are loaded to the fact database of programs generating the final problem descriptions. All these programs expect the file `problem_def.pi` in their path. The final definition of a planning task is written to another Picat file, which defines all allowed value assignments for all variables in the model. We can then load the definition and call a solver to label the variables.

Some functions, which are used by more programs, are defined in the file `common_translate_lib.pi`. The programs import this file as a module and expect the file to be in their paths during the execution.

Each file that describes a planning task encoded as TCPP, $R^2\exists$-Step or Reinforced Encoding contains the function `problem(Makespan) = Variables`. This function returns for the specified `Makespan` (number of steps in the parallel plan) the array `Variables` containing all variables defined in the model satisfying all constraints.

Additionally, all translation programs generate the map file `map.pi`. As actions, variables and values are represented by numerical identifiers in the final description of a planning task, we store the mapping in the map file.

The map file contains predicates which connect the original names from the SAS+ file with the assigned identifiers:

- `var(Id, Name)` for variables

- `val(VariableId, ValueName, ValueId)` for values of variables, where `ValueName` is the name of the value which is represented by `ValueId` in the domain of the variable represented by `VariableId`

- `op(Id, Name)` for operators

Since each model uses different variables, we must call a special decoder for each model to decode actions from the solution. The decoded actions are always written to the file `solution.plan`. Each action in the plan is on a separate line in the file together with the time step: `Time: Action`. Actions are represented by names defined in the original SAS+ file.

For plan validation, we use the program `validate_plan.pi`. This program reads the plan in the above described format from the standard input and verifies if all goals and preconditions of all actions are satisfied. If we pass the number 0 as an argument to the program, only the sequential plan is verified (actions in the order in which they are written in the file). Otherwise, the program verifies for each parallel step if all permutations of all actions that are executed in the step are valid.

A.2.1 TCPP

Planning tasks are translated to TCPP by the program `picat_to_tcpp.pi`. The program generates the output file `tcpp_problem_def.pi` containing TCPP constraints and auxiliary files `map.pi` and `decoder_info.pi`. All files are created in the directory passed as the first argument of the program.

Additionally, we can pass the following arguments represented by lower-case characters:

- `s` to reduce the width of transition tables by introducing auxiliary variables (see subsection 3.1.4)
- `m` to generate negative table constraints for mutexes generated by the Fast Downward translator (see subsubsection 3.1.5.1)
- `l` to generate tables describing value transitions of variables over a longer time horizon as table constraints (see subsubsection 3.1.5.2)
- `n` to generate tables describing value transitions of variables over a longer time horizon in the form of negative table constraints (see subsubsection 3.1.5.2); if both `l` and `n` are passed, the program will use `l`
- `c` to generate conditional mutexes of all pairs of variables (see subsubsection 3.1.5.3)

Along with the output file and the map file, the program generates an auxiliary file `decoder_info.pi`. Since TCPP does not contain variables for actions, we must decode actions in the plan from values of state variables and parallelism variables in all time points. This file contains preconditions and effects of actions defined by the numerical identifiers of variables and values and also values of parallelism variables.

This information is encoded in the following predicates:

- `precond(OperatorId, PreconditionVariableId, PreconditionValueId)`
  for each precondition of each operator, where the value represented by `PreconditionVariableId` has the value represented by `PreconditionValueId` in preconditions of the operator represented by `OperatorId`.
- `edge(VariableId, PreconditionValueId, EffectValueId, OperatorId)`
  for each precondition and effect of each operator, where the variable represented by `VariableId` has the value represented by `PreconditionValueId` in preconditions and the value `EffectValueId` in effects of the operator represented by `OperatorId`. If a variable is only in preconditions, then `EffectValueId` is equal to `PreconditionValueId`. If the value of an effect variable is not defined in precondition, then `PreconditionValueId` is equal to 0 (values are indexed from 1).
- `paral(VariableId, ParallelismValue, OperatorId)`
  for parallelism variable and each operator which contains the corresponding

state variable in effects, where the parallelism variable of the state variable represented by `VariableId` has the value `ParallelismValue` assigned for the operator represented by `OperatorId`. For the *no-op* action of the variable `Variable` corresponding to the value `Value`, `OperatorId` is represented by `noop(Variable, Value)` (`ParallelismValue` will be equal to 0).

The function
`problem(Makespan) = Variables`
defined in the file `tcpp_problem_def.pi` returns an array of two 2-dimensional arrays:
`Variables = {Vars, ParVars}`

- `Vars` contains values of state variables in all time points. `Vars[I][J]` is the value of the I-th variable at the time point J.
- `ParVars` contains values of parallelism variables. `ParVars[I][J]` is the value of the parallelism variable of the J-th variable at the time point I.

The program `tcpp_sat.pi` loads the problem defined in the file `tcpp_problem_def.pi`, which is located in its path and attempts to solve the problem with the given makespan using the SAT solver provided by Picat. The program writes one number to the standard output: 1 if the planning problem was solved or 0 if no solution was found. We need to call this program repeatedly with increasing makespan until a plan is found. If the solver finds a solution, the program uses the program `decode_actions.pi` to decode a plan from values of variables. The program expects two arguments: the makespan and the name of the output directory where the file with the plan should be created.

The program `decode_actions.pi` decodes actions from values of these variables passed as an argument to the predicate `decode_act(Solution, OutDir)` together with the requested output directory, where the program enumerates to the file `solution.plan` for each time step all actions executed during this step (see subsection 3.1.3) in the format described in the beginning of section A.2. The program `decode_actions.pi` expects `map.pi` and `decoder_info.pi` in the path.

Additionally, the program `picat_to_tcpp.pi` computes the minimum possible makespan, which corresponds to the maximum distance from the initial value to the goal value of a variable. The program writes this value to the standard output.

A.2.2 $R^2\exists$-Step encoding

For translation from SAS+ to the $R^2\exists$-Step encoding, we created the program `picat_to_rres.pi`. The problem description in the $R^2\exists$-Step semantics is written to the file `rres_problem_def.pi` in the directory passed as the first argument to the program. In the same directory, the program creates also the files `map.pi` and `decoder_info.pi`. The file `decoder_info.pi` contains for each action a fact `rank(Action, Rank)`, where `Action` is the integer identifier of the action and `Rank` is its rank. This information is used to decode the plan.

In addition, we can pass the following lower-case characters as parameters to the program `rres_problem_def.pi`:

- Parameters specifying types of chain constraints:

- `t` for implementation with table constraints replacing auxiliary variables and chain constraints (see subsubsection 3.2.1.2)
- `a` for implementation replacing all constraints by table constraints (see subsubsection 3.2.1.2)
- `r` for describing evolution of values of variables in parallel steps by regular automata (see subsubsection 3.2.1.3)
- `s` for reducing the width of the automata by splitting related actions to more automata (see paragraph 3.2.1.3.1)
- If more of these parameters are passed, then the program prefers parameters with respect to the following precedence: `s` > `r` > `a` > `t`

* `d` to add disabling edges to the enabling graph (see subsection 3.2.2)
* `m` to generate negative table constraints for mutexes generated by the Fast Downward translator (see subsubsection 3.2.3.1)
* `v` to generate table constraints describing possible values remaining for each variable at the next time point after applying each action (see subsubsection 3.2.3.2)

When a parameter `t`, `r` or `s` is passed to the program, the function
`problem(Makespan) = Variables`
in the file `rres_problem_def.pi` returns an array of three 2-dimensional arrays: `Variables = {Vars, Actions, ActionsByTime}`. Otherwise, the function returns an array of four 2-dimensional arrays: `Variables = {Vars, Actions, AuxVars, ActionsByTime}`.

* `Vars` contains values of state variables in all time points. `Vars[I][J]` is the value of the I-th variable at the time point `J`.
* `Actions` contains values indicating for each action and each time step if the action was executed, where `Actions[I][J]` corresponds to the J-th action at the time I. Actions are sorted by ranks.
* `AuxVars` are values of auxiliary variables (see subsubsection 3.2.1.1). `AuxVars[I][J]` is the value of the I-th auxiliary variable at the time J.
* `ActionsByTime` is the transposed array `Actions` (`ActionsByTime[I][J]` corresponds to the I-th action at the time J). Actions are sorted by ranks.

The program `rres_sat.pi` loads the problem defined in the file `rres_problem_def.pi` and attempts to solve the problem with the makespan specified as the first argument using the SAT solver. The program writes one number to the standard output: 1 if the planning problem was solved or 0 if the problem was not solved. We need to call this program repeatedly with increasing makespan until it finds a solution. If the solver finds a solution, the program uses the program `action_names_serial.pi` to write the names of actions to the file `solution.plan`, which will be created in the directory whose name is given as the second argument.

The program `action_names_serial.pi` contains the predicate `write_names(Solution, OutDir)`, which accepts in arguments the array `Actions` and the name of the requested output directory. The predicate translates the values of these variables to the plan and writes it to the file `solution.plan`. The resulting plan executes one action during each time step. The program `action_names_serial.pi` expects `map.pi` and `decoder_info.pi` in the path.

A.2.3 Reinforced Encoding

The program `picat_to_reinf.pi` translates the SAS+ problem description to the Reinforced Encoding. The result is written to the file `reinf_problem_def.pi` located in the directory passed as the first argument to the program. In the same directory, the file `map.pi` is created.

Any combination of the following lower-case characters can be passed to the program as next parameters:

- `t` for table constraints connecting the transition of each state variable during each time step with the value of the corresponding state variable before and after the transition (see subsubsection 3.3.1.2)
- `e` for describing relations between each transition and the corresponding actions by an equivalence (see subsubsection 3.3.1.2)
- `c` for modification of conflict action constraints (see subsection 3.3.2)
- `l` for limiting the length of sums in the modified conflict actions constraints corresponding to the option `c` (see subsubsection 3.3.2.1); even if `c` is not selected, the modified definition of parallel conflict will be used
- `s` for transition sequencing constraints (see subsubsection 3.3.3.2)
- `m` to include mutexes produced by the Fast Downward translator component (see subsubsection 3.3.3.1)
- `i` to compute the minimum makespan based on distances from initial to goal values of variables; the minimum makespan will be written to the standard output

The function
`problem(Makespan) = Variables`
defined in the file `reinf_problem_def.pi` returns an array of 3 two-dimensional arrays: `Variables = {Vars, Actions, Transitions}`

- `Vars` contains values of state variables at all time points. `Vars[I][J]` is the value of the I-th variable at the time point `J`.
- `Actions` contains values indicating for each action and each time step if the action was executed, where `Actions[I][J]` corresponds to the J-th action at the time I.

- `Transitions` contains transitions of all state variable at all time points. `Transitions[I][J]` is the transition of the I-th variable at the time point `J`.

The program `reinf_sat.pi` attempts to solve the problem defined in the file `rres_problem_def.pi` with the makespan specified as the first argument using the SAT solver. If the program solves the problem, it writes the number 1 to the standard output; otherwise, the program writes 0 to the standard output. To find a plan, this program must be called repeatedly with increasing makespan. After solving the problem, the program uses the program `action_names_serial.pi` to write the plan to the file `solution.plan` located in the directory whose name is given as the second argument.

The parallel plan is decoded from action variables by the predicate `write_names(Solution, OutDir)` implemented in the program `action_names.pi`. The predicate expects `Actions` as the first argument and the requested output directory for the file `solution.plan` as the second argument. The program `action_names` expects the file `map.pi` in the path.

B. Electronic attachments

B.1 Benchmarks

Electronic attachments contain all domains that we used for experimental evaluation in chapter 4 in the folder *Benchmarks*. All domains are from Fast Downward benchmark collection.

B.2 Source codes

In the folder *Source codes*, electronic attachments contain the Picat source codes implemented for translation of planning problems to the selected encodings, plan decoding and validation of plans. See Appendix A for documentation. Along with Picat files, we also included the file `run.sh` containing commands for Cygwin, where we show how the Picat programs can be used to solve a planning problem.

B.3 Declaration

Additionally, electronic attachments contain the signed solemn declaration.