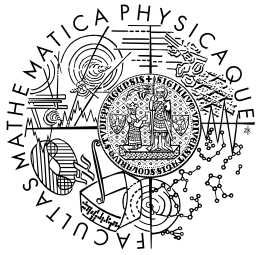


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Martin Gráf

Visual Editing of Domain Control Knowledge for Plannning

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: prof. RNDr. Roman Barták, Ph.D.

Study programme: Computer Science

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. 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 would like to thank my supervisor prof. RNDr. Roman Barták, Ph.D. for the patience and guidance this thesis wouldn't be possible without. I would also like to thank my parents Petr and Martina for too many things to list.

Title: Visual Editing of Domain Control Knowledge for Plannning

Author: Martin Gráf

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: The ability to only define the physics of an environment in classical planning tasks has been a long-standing obstacle in practical applications of such an approach. Current generic planners are typically capable of finding a solution to a given problem, but their inability to consider domain-specific constraints is often mirrored in a significant performance gap when compared to domain-specific algorithms. Remedying this gap would prove invaluable in making classical domain-independent planners viable in production environments.
In this paper, we will first introduce the area of classical planning and briefly touch on popular approaches to solving planning tasks. We will then show the principle of Attributed Transition-Based Domain Control Knowledge, which encodes additional problem-specific information into a domain. Finally, we will present our implementation along with experimental results.

Keywords: Knowledge Engineering; Classical Planning; Domain Control Knowledge

Název práce: Vizuální editace doménové řídící informace pro plánování

Autor: Martin Gráf

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: prof. RNDr. Roman Barták, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Schopnost definovat pouze možnosti prostředí v úkolu klasického plánování je dlouhodobou překážkou pro praktické aplikace tohoto přístupu. Moderní obecné plánovače jsou typicky schopné nalézt řešení daného problému, ale jejich neschopnost využít informací specifických pro doménu se často projeví ve výrazném rozdílu výkonu oproti algoritmům přizpůsobeným dané doméně. Pro použitelnost obecných plánovačů v produkčních prostředích je tudíž klíčové tento výkonostní rozdíl dohnat.
V této práci nejprve představíme téma klasického plánování a krátce shrneme běžné přístupy k řešení plánovacích problémů. Poté popíšeme princip Attributed Transition-Based Domain Control Knowledge, což je technika pro zakódování kontextuálních informací do domény a problému. Nakonec odprezentujeme naší implementaci spolu s experimentálními výsledky.

Klíčová slova: Znalostní inženýrství; Klasické plánování; Doménové kontrolní znalosti

Contents

Introduction

Attributed Transition-Based Domain Control Knowledge is a technique meant to improve the performance of planners solving domain independent classical planning problems. In attempting to find a sequence of actions as efficiently as possible, the commonly used formalisms describing only physics of an environment can sometimes allow planners to perform nonsensical sequences of actions. Such sequences can sometimes lead to expansive branches of computation that needlessly inflate the branching factor.

These issues are currently being adressed through either more nuanced planning techniques, classical planning extensions, or through simply discarding the feature of domain independence in favour of domain-specific planning techniques. The dominance of domain-specific planners in terms of performance hinders the development of domain independent planners and planning techniques, thus slowing down development and research in the area of classical planning as a whole, instead resulting in a confusing mix of techniques and approaches that are only sometimes beneficial to classical planning as a whole.

Improving the performance of domain independent planning is difficult. Improving search techniques can only take us so far, seeing as they can either be utilized by domain-specific planners as well, or fail to measure up to the performance gains of domain-specific approaches. It then becomes key to find ways of encoding domain-specific contextual information into planning tasks in a domain-independent way, so as to allow domain experts to enjoy both good performance, and the wider variety of tools available to domain-independent planning. ATB-DCK attempts to do just that.

ATB-DCK is a technique of encoding additional contextual information into planning tasks inspired by modelling the internal logic of domains using finite automata. The domain expert needs to only choose a domain, determine what inefficiencies are present in the domain definitions and cause the greatest hurdles during the planning process, use this knowledge to model an ATB-DCK, which represents the internal logic which sequences of actions in a given domain should follow in order to avoid these pitfalls, and define the logical rules used to infer the initial situation of this new logic.

This thesis first summarizes key features, properties, and definitions of classical planning, briefly ellaborates upon existing DCK techniques, reherses the original description of ATB-DCK as done by Chrpa and Barták[1], ellaborates on implementation details of encoding ATB-DCK, presents the software created to illustrate ATB-DCK encoding in practice. Finally, a new ATB-DCK for a specific domain is designed and experimentally evaluated, in order to illustrate how the ATB-DCK design migh work in practice and what performance changes can be expected.

1 Classical Planning

Classical planning[2] aims to find a sequence of steps an agent should take if he intends to transform the world from its initial state, into its desired state. It presumes that the environment is static, fully observable, and deterministic. In a static environment, an agent has any amount of time to deliberate the best possible action. A static environment doesn't change independently of the agent's actions. An observable environment gives the agent access to any relevant information about itself. This information may describe objects present in the environment, or their relationships, as well as any other special properties the environment may have. In a deterministic environment, the agent knows how the environment changes and reacts to every action the agent could take [2, 3].

Even so, meeting all of the environmental constraints is no guarantee that the planning problem is computationally achievable. While the search for a plan might be fairly straightforward in principle, nearly all planners struggle with controlling the combinatorial explosion associated with large and complex environments[3]. To illustrate, the problems of plan existence and determining the length of the plan both fall into the PSPACE class - a class larger than NP referring to problems that can be solved by a deterministic Turing machine with a polynomial amount of space[2, 4]. Nonetheless, real-world implementations of planners and planning uses rarely concern themselves with such possibly unsolvable problems. In practice, planners employ several clever techniques and helpful additional constraints to maximize their speed and robustness for specific kinds of problems. The most important of these planners for this thesis will be generic planners.

Note that most definitions are either directly quoted or paraphrased from the book Artificial Intelligence: A Modern Approach (both 2016 and 2021 editions)[4, 3], the book Automated Planning: Theory and Practice[2], or the paper Attributed Transition-Based Domain Control Knowledge for Domain-Independent Planning[1].

1.1 Classical representation

The **classical representation** describes the environment using a language L composed of first-order logic predicates P_L. To properly describe an environment, we first need to describe what objects we are working with (constant symbols), what relationships any possible objects can have (atoms with variable symbols), and finally what specific relationships hold for which objects (grounded atoms).

Definition 1 (Terms, atoms, grounded atoms)**.** *A **term** is either a variable symbol, or a constant symbol. Variable symbols are constructs used to generalize predicates. **Constant** symbols represent objects within a given environment.*

*An **atom** is a predicate of the form $p(t_1, ...)$, where $p \in P_L$ and t_i are only terms. An atom or its negation is also called a **literal**. An atom $p(t_1, ...)$ is **grounded** if t_i is a constant symbol for all i. Grounded atoms are sometimes also called **instances** of atoms.*

Example (**Terms, atoms, grounded atoms**). *terms* ← {
constant symbols: $Mark, Alice$
variable symbols: x, y } ▷ As we can see, constant symbols refer to specific entities, while variable symbols are generic standins.
atom ← likes(x, y) ▷ Object x likes object y
grounded atom ← likes$(Mark, Alice)$ ▷ Mark likes Alice

Once possible objects and relationships are defined, a formalism describing the states of the world (state), how the agent can change the world (planning operator), and specific instances of these changes (actions) are needed. These concepts combined provide us with a powerful tool of modelling the relevant parts of the world, which will allow us to manipulate the world easier.

Definition 2 (State, planning operator, action)**.** *A **state** is a finite set of grounded atoms.*

*A **planning operator**, is a tuple* $a = (name(a), preconditions(a), effects(a))$*. name takes the form* $name(a) = op\_name(v_1, ..., v_n)$*, where op_name is a unique identifier and* $v_1, ..., v_n$ *are variable symbols.* $preconditions(a)$ *are sets of literals (positive or negative) that must hold true for the operator to be applicable.* $effects(a)$ *are sets of predicates which shall hold true once the operator is applied.*

*An **action** is a planning operator, where all predicates found in preconditions or effects have had all variable symbols substituted for constant symbols. In other words, an action is a fully grounded instance of a planning operator.*

It can sometimes be advantageous to split $effects(a)$ *or* $preconditions(a)$ *into positive and negative predicates, denoted as* $effects(a) = (effects(a)^+ \cup effects(a)^-)$ *and* $preconditions(a) = (preconditions(a)^+ \cup preconditions(a)^-)$ *respectively.*

*Note that we adopt the **closed world** assumption, meaning that any predicates which aren't either explicitly mentioned or aren't a logical consequence of the stated predicates do not hold.

In the following example, an agent is working with a world where Mark likes Alice, and both Alice and Mark are single. The agent can then transform the world by either having a person x who likes person y ask out person y, and then having the person y either accept or reject said proposition while similar preconditions hold. Since there is no operator to make person y like person x, if person y doesn't already like person x, the only possible outcome is rejection.

Example (**State, planning operator, action**). *State* ← { likes(Mark, Alice), single(Alice), single(Mark) } ▷ Both Mark and Alice are currently single. Mark also likes Alice.
Operators ← {
ask_out ← {
name: ask_out(x, y)
preconditions: likes$(x, y) \wedge$ single(x)
effect: asked_out(x, y) }
accept ← {
name: accept(x, y)

preconditions: $\text{likes}(x, y) \wedge \text{single}(x) \wedge \text{asked\_out}(y, x)$
effect: $\text{dating}(x, y)$ }
reject ← {
name: $\text{reject}(x, y)$
preconditions: $\neg\text{likes}(x, y) \wedge \text{asked\_out}(y, x)$
effect: $\text{sad}(y)$ } } ▷ Notice a state doesn't need to explicitly state false predicates, due to the closed world assumption.
Actions ← {
ask_out ← {
name: $\text{ask\_out}(Mark, Alice)$
preconditions: $\text{likes}(Mark, Alice) \wedge \text{single}(Mark)$
effect: $\text{asked\_out}(Mark, Alice)$ }
reject ← {
name: $\text{reject}(Alice, Mark)$
preconditions: $\neg\text{likes}(Alice, Mark) \wedge \text{asked\_out}(Mark, Alice)$
effect: $\text{sad}(Mark)$ } } ▷ For the accept action, preconditions don't hold.

*Note that there may be deviations in implementations, such as only describing what values changed since the last state, but those nuances are irrelevant to our use case.

The definitions of actions naturally lead to the question of action application. Actions would be useless without us being able to know when they can and can not be applied to a state and how they transform the state.

Definition 3 (Applicable action)**.** *An **action** $a = (\textit{preconditions}(o)^+, \textit{preconditions}(o)^-, \textit{effects}(o)^+, \textit{effects}(o)^-)$ in a planning state S is **applicable** if and only if $\textit{preconditions}(o)^+ \subseteq S \wedge (\textit{preconditions}(o)^- \cap S) = \emptyset$. If the applicable action is chosen, it transforms the state S in the following way: $(S \setminus \textit{effects}(o)^-) \cup \textit{effects}(o)^+$.*

A domain is simply a formalism giving the agent all the tools mentioned in the previous definitions in a unified manner. Note that it only describes what is possible in a given world, not who these possibilities apply to or what specific world the agent is working with.

Definition 4 (Domain)**.** *A **Domain** is typically defined as a tuple (Preds, Ops), where Preds are predicates and Ops are operators available within the domain.*

Example (**Domain**). *Domain* ← {
Predicates: { $\text{asked\_out}(y, x), \text{single}(x), \text{likes}(x, y), \text{dating}(x, y)$}
Operators: {
ask_out ← {
name: $\text{ask\_out}(x, y)$
preconditions: $\text{likes}(x, y) \wedge \text{single}(x)$
effect: $\text{asked\_out}(x, y)$ }
accept ← {

name: accept(x, y)
preconditions: likes$(x, y) \wedge$ single$(x) \wedge$ asked_out(y, x)
effect: dating(x, y) }
reject $\leftarrow$ {
name: reject(x, y)
preconditions: $\neg$likes$(x, y) \wedge$ asked_out(y, x)
effect: sad(y) }
give_roses $\leftarrow$ {
name: give_roses(x, y)
preconditions: likes(x, y)
effect: likes(y, x) } }
}

Finally, given a specific domain, the agent is able to tackle specific situations. He is provided with the possibilities (domain), a specific state of the world (initial state and objects), and a description of how the world should look after the agent is finished. Classical planning is then the act of an agent finding a series of steps (consecutive applications of actions) to achieve the desired goal state given all of this information.

Definition 5 (Problem)**.** *A* ***Problem*** *is a tuple of (Obj, Init, Goal), where Obj are objects (or constant symbols) to which predicates can be applied, Init is the initial state, and Goal is a set of predicates which need to hold true in a given state for the problem to be solved.*

Example (**Problem**). *Problem* $\leftarrow$ {
Objects: { $Mark, Alice$ } Init: { likes(Mark, Alice), single(Alice), single(Mark)}
Goal: { likes(Mark, Alice), likes(Alice, Mark), dating(Mark, Alice) }
}

Since both a problem and a domain are required in order for us to be able to utilize planning in a meaningful way, any planning task has to contain both.

Definition 6 (Planning task)**.** *A* ***Planning task*** *is then simply a tuple of our already defined (Domain, Problem) = (Preds, Acts, Obj, Init, Goal). The purpose of this planning task is to find a sequence of actions, such that by following this sequence, we can transform the Init State into a state S which satisfies the Goal. A given state S satisfies the goal if* $Goal^{+} \subseteq S \wedge (Goal^{-} \cap S) = \emptyset$.

During planning, the concept of reachable states becomes relevant. Sometimes, a planning problem may ask the impossible, meaning we need to be able to identify an impossible task.

Definition 7 (Reachable State)**.** *A* ***Reachable State*** *simply indicates whether there exists an ordered sequence of actions - a plan - which transforms the current State into this new Reachable State.*

The intended output of a planning agent is a specific sequence of steps complying with the logic defined in a domain solving the specific problem.

Definition 8 (Solution)**.** *A* ***solution*** *is simply an ordered sequence of actions that if executed, transforms the initial state into the desired goal state.*

Example (**Solution** to the example problem).
1. give_roses(Mark, Alice)
2. ask_out(Mark, Alice)
3. accept(Alice, Mark)

1.2 Motivating example

To better illustrate how ATB-DCK introduced later transforms the Domain and a problem, we will be using a running example. For this purpose, we shall be using the classic **Blocks world**[5] domain. This simple example typically models a robotic arm that is attempting to stack several blocks on top of a platform or on top of one another.

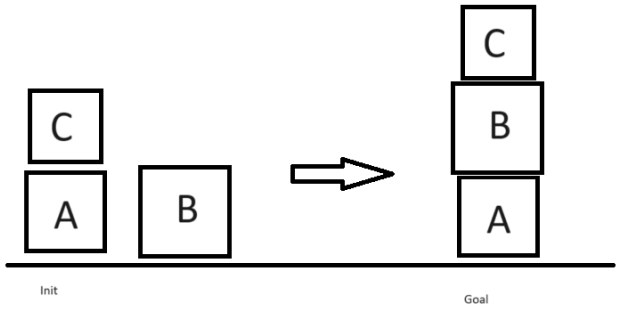


Figure 1.1 A simple blocks world example

Example (**Domain predicates** apply to specific objects at a given point in time).
free(x) ▷ The object x has no other object on top of it. This predicate could be considered redundant, but it eases checking for availability of x for stacking significantly.
on(x, y) ▷ Object x is on top of object y.
holding(x) ▷ The robotic arm is holding the object x.
arm_empty ▷ Similar to free(x), simply indicates if the arm is holding something.
on_table(x) ▷ Object x is placed on the table.

Example (**Planning problem** simply defines the initial state and the goal state).
initial ← on_table(a) ∧ on_table(b) ∧ on(c, a) ∧ arm-empty
goal ← on_table(a) ∧ on(b, a) ∧ on(c, b)

Example (**Planning operators**). *stack* ← {
name: stack(x, y)
preconditions: holding(x) ∧ free(y)
effect: arm_empty ∧ on(x, y) ∧ ¬holding(x) ∧ ¬free(y) ∧ free(x) } ▷ Places object x on top of object y, if and only if there isn't anything on top of y and our arm is holding x.
unstack ← {
name: unstack(x, y)
preconditions: arm_empty ∧ free(x)
effect: ¬arm_empty ∧ ¬on(x, y) ∧ free(y) ∧ holding(x) ∧ ¬on_table(x) } ▷ Pick up an object x which is on top of y if and only if the arm is empty and the object is free.
pickup ← {

name: pickup(x)
preconditions: arm_empty $\wedge$ on_table(x)
effect: $\neg$arm_empty $\wedge$ holding(x) $\wedge$ $\neg$on_table(x) } $\triangleright$ Take object x from the table if it is there.
putdown $\leftarrow$ {
name: putdown(x)
preconditions: holding(x)
effect: arm_empty $\wedge$ $\neg$holding(x) $\wedge$ on_table(x) } $\triangleright$ Place object x down on the table if we are currently holding it.

1.3 Planners in practice

For a proper understanding of the inner workings, weaknesses, and strengths of classical planning, we shall provide a brief overview of algorithmic techniques used by planners. We shall then proceed to provide context on the technology most often used to represent domains and problems in practice. Afterward, a brief look into the computational complexity associated with planning will be presented.

1.3.1 Algorithms for classical planning

Many techniques used to search for a solution are based on searching through the set of possible states reachable from the current state. Since most plans have multiple steps, this search can be represented using a tree-like graph. This so-called state space search often exhaustively applies all possible actions to the current state, until a desired state is found. Such approaches have the advantage of being guaranteed to find a solution if it exists but typically run into issues with computational achievability due to their large branching factors.

Forward state space search This is indeed the most straightforward state space search possible. It searches through states with grounded atoms, treating the goal state as a state with all atoms grounded and positive. The applicable actions are determined by testing if the preconditions of each available action schema hold in a given state. If preconditions hold, the action is applicable. The order of actions tried can then either be random or use heuristics. This often quickly leads to a combinatorial explosion in its branching factor.[3]

Backward state-space search Also known as regression search, it takes the opposite approach to Forward search. As the name would suggest, Backward search starts with the goal state. It then retroactively seeks out states that might have resulted in the current state by finding so-called relevant actions. Relevant actions differ from applicable actions in that it attempts to unify the effect of an action with one of the goal literals, instead of its preconditions. In practice, this approach generally reduces the branching factor in comparison to Forward search, seeing as it eliminates a greater number of irrelevant actions. However, it also makes finding a good heuristic more difficult, which is why it isn't universally favored.[3]

Planning as Boolean satisfiability Much like in many subfields of AI, classical planning problems are translatable into a propositional form. This translation propositionalizes actions by grounding each of its variables. It then adds axioms that make individual actions mutually exclusive and axioms corresponding to action preconditions. Finally, the translation propositionalizes the goal and adds succesor-state axioms, which represent the evolution of a plan over time. The obvious disadvantage of utilizing propositional logic is the significant increase in description size[3, 6]. This does, however, allow us to utilize SAT solvers. These solvers are excellent at finding the shortest possible plans, searching for the plans in a parallel manner, and in being highly efficient from a computational perspective[7]. SAT solvers in planning are now popular in both IPC[8, 9] and SAT competitons[10].

Planning graphs Planning graphs have originally been proposed along with the *Graphplan* algorithm[11]. This algorithm may have fallen out of favor today, but the planning graph data structure has proven useful in several other ways. The planning graph is fast to construct, polynomial in size, and excels at representing constraints inherent to the problem in an explicit manner[11]. These useful properties allow us to more easily identify and remove symmetries in search space, prune needlessly long branches, or develop smart heuristics[12]. Planning graphs are a useful tool for many modern planners and techniques[13, 14, 15].

Other planning approaches Other techniques utilized in classical planning, be it for algorithms, developing heuristics, or use of specific data structures include but are not limited to:

- Constraint satisfaction programming - A less popular formalism used in a wide variety of ways[16, 17, 15].
- Situation calculus - Similar to SAT planning, but relies on first-order logic instead[3].

Many other variations or combinations of existing approaches and emerging practices are constantly being developed. Categorizing these techniques is often highly speculative, as they belong to existing branches in the eyes of some, or into categories of their own in the eyes of others. The situation becomes even murkier once extensions to classical planning are introduced, which we will address in Chapter 2.

Heuristics While the mentioned algorithms are certainly capable of solving a planning task by themselves, any algorithms are bound to struggle with notable subsets of problems, with which any guidance of planning can lead to huge performance gains. This is achieved through heuristic functions. A heuristic function $h(s)$ estimates the distance from state s to the goal. The function needs to be **admissible**, meaning the function never overestimates the true distance from the goal. Naturally, a large number of different heuristics exist, many of which are domain-specific, but heuristics can notably improve domain-independent planning as well. Seeing as ATB-DCK isn't strictly speaking a heuristic technique, having more in common with the HTN approach, only a brief listing of some

popular heuristics and heuristic construction techniques will be provided with links to works containing more details.

- Ignore-preconditions is a very simple heuristic that simply ignores action preconditions[3, 18].
- Ignore-delete-lists deletes all negative literals from action effects, thus allowing the solution search to make monotonic progress[19, 3, 20].
- Pattern databases store optimal solutions for abstracted states, giving a sort of heuristic lookup table, rather than being a specific method to estimate the distance from a goal[21, 19].
- Symbolic Pattern Databases are an expansion of Pattern databases making them more memory-efficient through the use of boolean functions[22, 19].

Heuristics can be found by hand, but numerous methods for automated heuristic construction (sometimes directly implemented in specific planners) also exist. These include but are not limited to:

- Hill-climbing search attempts to optimize a function through only local changes improving an objective function[20, 23, 24].
- LPG is a local graph search utilizing Temporal Action Graphs to identify inconsistencies[25, 19, 26].

Needless to say, we are unable to mention all or even a representative fraction of available heuristic approaches and ideas. For the sake of brevity, we shall refer the reader to a paper from Macdonald and Enright[19], which goes into more detail.

1.3.2 Planning Domain Definition Language

Created over 20 years ago, the Planning Domain Definition Language (PDDL) has become the de-facto standard in domain-independent planning. The language is intended to only describe the physics of a domain. The authors went as far as describing the languages' commitment to neutrality in its features as 'perverse'[27]. Its roots lie in the STRIPS[28] language, which itself was massively influential in the early days of classical planning. The use of PDDL in the International Planning Competition and planners developed for it had made PDDL nearly synonymous with classical planning. Its syntax is nearly identical to the factored (classical) representation mentioned above. To allow for easier modification and increased customizability, PDDL is factored into individual subsets of features (also called requirements), which can be used or ignored at will. This clever feature allows for planners that have been created with only specific features in mind to simply skip any domains or problems using unfamiliar syntax.

Example (Comparison of classical representation with PDDL syntax). In classical representation: $stack(x, y) \leftarrow \{$

name: stack(x, y)
preconditions: holding$(x) \wedge$ free(y)
effect: arm_empty $\wedge$ on$(x, y) \wedge \neg$holding$(x) \wedge \neg$free$(y) \wedge$ free(x) }

In PDDL:

```
(: action stack
   :parameters (?x - block ?y - block)
   :precondition (and (holding ?x) (free ?y))
   :effect (and (not (holding ?x))
      (not (free ?y))
      (free ?x)
      (arm_empty)
      (on ?x ?y)
))
```

▷ This snippet was originally used in [1].

The extensions most relevant for this thesis are typing and STRIPS. These are perhaps the most basic requirements, as they only add very basic negation and typing of variables. These requirements are used by the most widely supported PDDL version - version 1.2. This support isn't strictly necessary for the ATB-DCK technique introduced later in this thesis. ATB-DCK can be used in any PDDL version, these requirements and version were simply chosen to present ATB-DCK in the most straightforward manner possible.

1.3.3 Planning complexity

A detailed description of planning complexity is presented in the book 'Automated Planning: theory and practice' by M. Ghallab, N. Dau, and P. Traverso[2]. In general, the issues of plan existence and plan length are relevant to the field of classical planning. As they aren't directly relevant to the subject of this thesis and serve more for context, let us simply mention that the issue of Plan existence falls into the *EXSPACE-complete* class, while the issue of plan length is *NEXPTIME-complete*.

2 Attributed Transition-Based Domain Control Knowledge

Let us briefly recall the simple example of a planning task where Mark likes Alice, both are single, and the planning agent is supposed to find a way to make them date using the operators $ask\_out, accept, reject$, and $give\_roses$.

Example (**Problem**). *Problem* ← {
Objects: { *Mark*, *Alice* } Init: { likes(Mark, Alice), single(Alice), single(Mark)}
Goal: { likes(Mark, Alice), likes(Alice, Mark), dating(Mark, Alice) }
}

The obvious solution to this problem would be to first give Alice roses, and then proceed to ask her out, followed by her inevitably accepting due to the definitions of our operators. This would indeed be the optimal solution, but notice that nothing is stopping the agent from asking out Alice before she likes Mark, possibly repeatedly. Such a solution is a valid solution, no matter how many rejections and pointless propositions it involves, it only needs to result in dating by the end.

For classical planning in its base form, the only line of defense against this kind of suboptimal action sequences are algorithms or heuristics implemented by individual planners, which can only do so much to prevent this from happening in more complicated problems. The key issue here is that the planning agent has no real way of guessing which sequences of actions might lead to a solution (or which sequences certainly won't lead to a solution) from just the domain and problem definitions. Knowing that an accepted date proposition can only occur after the proposer is already liked is not the same as knowing that there is no point in asking out someone who doesn't like the proposer in the first place(An unsolvable problem for all of teenagekind). It is precisely this shortcoming of classical planning (and PDDL), that ATB-DCK aims to remedy.

Transition-based DCK as introduced by Chrpa and Barták[29], was later expanded into Attributed Transition-based (ATB) DCK in a follow-up paper[1]. Using states and associating specific planning operators with transitions between states provides us with a way of representing the ordering of operators in any particular solution. Adding Attributes to these states then allows us to track specific objects, easing the elimination of actions that can not lead to a solution, or knowing when specific subgoals have been achieved. In terms of our running example, ATB-DCK could be used to keep track of which blocks do not need to be moved, and which do, or which blocks are already part of their intended structure, and which aren't. [1]

[1]Note that we shall be borrowing examples for our definitions from (Chrpa, Barták, Vodrážka, Vomelová 2020)[1]

2.1 Definitions

As mentioned above, for the blocksworld domain, we use Attributed states to prune computational branches causing the robot arm to destroy its previous correct work. Specifically, this takes the form of the state *goodtower*, which is used to prevent the agent from moving blocks that are already in their intended position, *badtower*, which indicates the block will need to be moved before a solution can be reached, and *dck_holding*, as a transition state between the two while the block is being manipulated. How these states are utilized will become clear by the end of this chapter.

Not to be confused with a planning State, an Attributed state is meant to be used as a predicate defined in the domain. It aims to capture certain attributes of one or more objects that are not strictly relevant from an action planning perspective but are relevant from an action ordering perspective.

Definition 9 (Attributed state)**.** *An **attributed state** s is composed of a unique identifier, and a set of variable symbols representing attributes. Formally:* $s = uid(var_1, ..., var_n)$

Just like a problem needs to specify which predicates are true for which objects in its initial state, attributed states (and later attributed memory) can also already be true for certain objects. Seeing as the initialization will need to be problem-specific, it is more practical to define initialization rules in the form of Horn clauses as proposed by the original paper[1]. These clauses (also initialization or inference rules) take the form

$$R_p(a_{k_1}, ..., a_{k_m}) \leftarrow F_p(a_1, ..., a_n)$$

where $R_p(a_{k_1}, ..., a_{k_m})$ can be either a memory predicate or an attributed state, while $F_p(a_1, ..., a_n)$ is a conjunction of literals which can contain:

- An initial state query I:$p(a_1, ..., a_z)$
- An goal state query G:$p(a_1, ..., a_z)$
- An initial DCK memory or state query $p(a_1, ..., a_z)$
- A cardinality query $p(a_1, ..., a_z)$

Initialization and goal queries simply check for information already present in the problem definition, looking at its goal state and initial state. DCK state and memory queries check if the queried atoms can be derived using their own initialization rules. A cardinality query is a special function counting the number of satisfied queries, which are provided to the function as arguments. The specifics of this function will be explored later while discussing the software created to complement this thesis. Once these rules are defined and a specific problem is provided, their truth values are determined by substituting variables present in the query for problem-specific objects. [2]

[2]Shortcut predicates (conjunctions of the above rules) will also ocassionally be used to enhance readability and shorten examples.

In a given problem, the initial attributed state (if any) of an object can be determined using a logical formula. This formula can use additional predicates not present in the original domain definition that can be deduced from problem-specific goal state, or even predicates found in the ATB-DCK definition itself. These goal predicates shall typically be denoted as starting with 'G'.

Example (**Attributed state**). Attributed States ← { DCK_holding(?x), goodtower(?x), badtower(?x) } ▷ Distinction between an object not needing to move for the rest of the plan, needing to move, or if it is currently being moved. Note that holding would be duplicit without the DCK prefix.
Initialization rule $goodtower(?x) \leftarrow (ontable(?x) \wedge (Gontable(?x) \vee \neg Gon(?x, ?y) \wedge \neg Gontable(?x))) \vee (on(?x?y) \wedge (Gon(?x?y) \wedge goodtower(?y)) \vee (\neg Gon(?x, ?y) \wedge \neg Gontable(?x) \wedge \neg Gon(?y, ?x) \wedge \neg Gclear(?x) \wedge goodtower(?y))))$
Initialization rule $badtower(?x) \leftarrow (ontable(?x) \wedge Gon(?x?y)) \vee (on(?x?y) \wedge (Gontable(?x) \vee (Gon(?x?z) \wedge ?y \neq ?z) \vee (Gon(?z?y) \wedge ?z \neq ?x) \vee badtower(?y)))$

DCK memory takes on a similar form to attributed states, only really being split to better distinguish the roles they take on in the planning task. Unlike Attributed states, DCK memory models certain facts about either landmarks we have achieved in the planning process, temporary attributes that can't be used to determine the ordering of actions by themselves or information about the goal.

Before defining transitions, we also need to define what a DCK memory is:

Definition 10 (DCK memory)**.** *A* ***DCK memory*** *for a domain is a set of predicates distinct from the domain predicates.*

Example (**DCK memory**). Memory predicates ← { Gontable(?x), Gon(?x ?y), Gclear(?x) } ▷ These can either be especially defined, or deduced from the information present in the problem

Before moving on, let us define a helper function $var(x)$, which denotes possible variants of some predicate p. A variant of predicate p is created by renaming the variable symbols of p. As an example, one possible variant of the predicate $p = on(?x, ?y)$ would be $var(p) = q = on(?z, ?g)$. Since Attributed states also take the form of predicates during planning, the *var* function can be extended to include them as well without change.

Attributed states are used in tandem with transitions. Together, they form the 2 fundamental building blocks of ATB-DCK. Transitions connect different states, are associated with planning operators, additional constraints, or modify the memory. They serve as a mechanism by which we can transfer between attributed states. During the encoding process, every individual transition is encoded as a separate operator, modifying the original associated operator in a specific way, which will be explored in a later section.

Definition 11 (DCK transition)**.** *An* ***DCK transition*** *is a tuple* (s, o, C, P, s') *where*

- *Let us denote domain predicates as D, DCK memory as M, and DCK states as S*
- $var(s), var(s') \in S$
- *s and s' are attributed states.*
- *$o \in$ planning operators $\cup$ empty operator*
- *C is a set of constraints of the form:*
  - *p such that $var(p) \in (D \cup M) \vee S$*
  - *$\neg p$ such that $var(p) \in D \cup M$*
- *P is a set of modifiers:*
  - *$+p \vee -p$ such that $var(p) \in M$*

Additionally, let us mention that transitions can also be associated with a dummy "empty operator" ($\perp$), which allows us to simplify ATB-DCK definitions by making it possible to modify DCK memory or move between attributed states. This operator will be removed from solutions, has no effects or preconditions, and serves only as a tool to make the definitions of ATB-DCK more compact. Let

us imagine a block ?x is currently in a tower from which it has to be moved in order to achieve the goal - a badtower. To solve this issue, we need to perform the action 'pickup(?x)' (operator 'unstack' needs a special transition instance). In addition to the action preconditions, ?x needs to be positioned on some object ?y in the goal, which is also free (or clear(?y) needs to hold true). Once the transition is used, the attributed state badtower(?x) will be replaced by DCK_holding(?x)

Example (**DCK transitions**). Initial state ← { badtower(?x) }
Associated operator(s) ← { pickup(?x) }
Constraint ← { Gon(?x, ?y), clear(?y), goodtower(?y) } ▷ Notice we utilize the DCK memory (Gon), the domain predicates (clear), and attributed states (goodtower)
No DCK memory is being modified in this instance
Target state ← { holding(?x) }

Finally, for Attributed Transition-based DCK (ATB-DCK) simply combines the above definitions into one structure.

Definition 12 (Attributed Transition-based DCK)**.** ***Attributed Transition-based DCK*** *is a tuple (Dom, S, M, T) where*

- *$Dom = (P, O)$ is a domain as defined earlier.*
- *S is a set of attributed states.*
- *M is DCK memory.*

- *T is a set of DCK transitions.*

There is nothing new about this definition, it simply couples all the information needed to define a planning task with ATB-DCK into one structure. ATB-DCK for the Blocksworld domain would be defined as follows:

Example (**ATB-DCK** for Blocksworld). *Dom* = the Blocksworld domain composed of predicates and operators we mentioned earlier
S ← { badtower(?x), goodtower(?x), dck_holding(?x) }
M ← { Gontable(?x), Gon(?x ?y), Gclear(?x) }
Transitions ← {

- { **Initial state** ← { badtower(?x) }, **Associated operator** ← { pickup(?x) }, **Constraint** ← { Gon(?x, ?y), clear(?y), goodtower(?y) }, **No DCK memory is being modified in this instance Target state** ← { dck_holding(?x) }, }
- { **Initial state** ← { dck_holding(?x) }, **Associated operator** ← { put_down(?x) }, **Constraint** ← { Mstacked(?y) }, **No DCK memory is being modified in this instance Target state** ← { badtower(?x) }, }
- ... (the other transitions follow the same schema as depicted in 2.1)

}

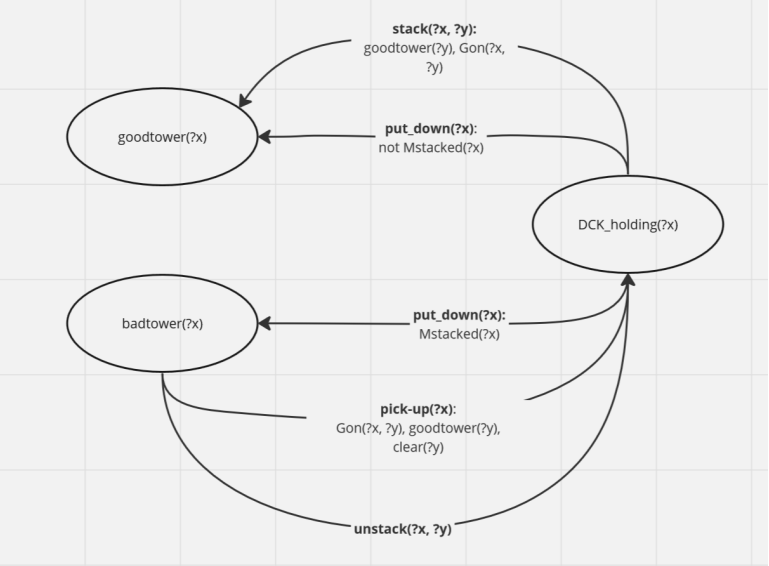


Figure 2.1 Blocksworld ATB-DCK

2.2 Encoding of ATB-DCK into Domains

If defined properly, an ATB-DCK can be added into any domain by simply adding predicates representing attributed memory or states, while properly redefining planning operators to behave as ATB-DCK transitions. Predicates are

only added, but if an operator is associated with a transition, it gets replaced, possibly with multiple copies if it is associated with multiple transitions. The redefinition extends preconditions by transition constraints, extends effects by the modifiers, adds the target state into the positive effects of the operator, and adds the origin state into the positive preconditions and negative effects of the operator. Naturally, in order to do this, individual attributed states and memory will need to be translated into unique predicates and the variables in the redefined operators will need to be correctly named and synchronized with what objects are having their state modified.

Much like the definition of a domain allows a problem to create specific grounded atoms to create a planning state in a given planning task, a configuration represents the grounded atoms created out of the defined attributed states and memory. In other words, a **configuration** can be thought of as an ATB-DCK equivalent of a state in an ordinary planning task. A configuration could be thought of as an extension of the definition of a state, since the planner indeed does not differentiate between the two once encoded, but it is important to make the distinction during the encoding process.

Definition 13 (Configuration)**.** *Let $D = (Dom, S, M, T)$ be ATB-DCK. A configuration is a pair $D^c = (S^c, M^c)$ such that $S^c \subseteq \{s^c | s^c$ is instance of $s \in S\}$ and $M^c \subseteq \{m^c | m^c$is instance of $m \in M\}$. Instances of both memory and attributed states are determined using problem-specific objects and DCK memory objects.*

Example (**Configuration** - For a problem outlined in Example 7). Instantiated attributed states ← { goodtower(a), badtower(b), badtower(c) } ▷ Object a will be able to stay where it is until the problem is solved, unlike the other objects. Instantiated attributed memory ← { Gon(b, a), Gon(c, b), Gontable(a) }

To better understand both the configuration and the following attributes of a transition, recall the state and the applicability of an action. A configuration could be considered an extension of a planning state. The applicability of a transition is an extension of the concept of action applicability. It demands a specific initial (attributed) state, and its associated operator (if any) to be applicable in its initial state, and additionally supports applicability constraints on other attributed states, ordinary atoms, or memory. The application of the transition then modifies the configuration in a way comparable to how an action modifies a state.

As eluded to at the start of this section, this definition formalizes how a transition behaves after being added to an operator. It highlights the added constraints on a planning state in the form of attributed memory and state predicates, and describes how an operator modifies the planning state, attributed state, and attributed memory after the operator with which the transition is associated is applied.

Definition 14 (Transition attributes)**.** *Let $D = (Dom, S, M, T)$ be ATB-DCK, $t = (s_t, o_t, C_t, P_t, s'_t)$ be a transition $t \in T$ and $f(s_t)$ be a substitution function from variables in T to objects in some problem of a planning task. Lastly, let s_p*

be a state in some planning task within a domain Dom and $D^c = (S^c, M^c)$ *be a configuration of* D.

An applicable transition t*:*

- $f(s_t) \in S^c$
- $f(o_t)$ *is applicable in* s_p
- $\forall p \in C_t : f(p) \in s_p \cup M^c$
- $\forall \neg p \in C_t : f(p) \notin s_p \cup M^c$
- $\forall p \in C_t, var(p) \in S : f(p) \in S^c$

Result of transition t *is* s'_p *and* $D'^c = (S'^c, M'^c)$*:*

- $s'_p = (s_p \setminus f(effects(o_t)^-)) \cup f(effects(o_t)^+)$
- $S'^c = (S^c \setminus \{f(s_p)\}) \cup \{f(s'_p)\}$
- $M'^c = (M^c \setminus \{f(p)| - p \in P_t\}) \cup \{f(p)| + p \in P_t\}$

This allows us to extend the definition of a planning task into an ATB-DCK planning task.

Definition 15 (ATB-DCK planning task)**.** *The current task definition of* $(Domain, Problem)$ *simply gets extended by an ATB-DCK* C *and its initial configuration* C^I *into* $(Domain, Problem, C, C^I)$.

For brevity, we shall not describe an entire planning task with ATB-DCK, but an illustration of how the unstack operator might change after its associated transition is encoded will be provided.

Example (**ATB-DCK unstack operator**). *old_stack* $\leftarrow$ {
name: stack(x, y)
preconditions: holding$(x) \wedge$ free(y)
effect: arm_empty $\wedge$ on$(x, y) \wedge \neg$holding$(x) \wedge \neg$free$(y) \wedge$ free(x) } $\triangleright$ Places object x on top of object y, if and only if there isn't anything on top of y and our arm is holding x.
new_stack $\leftarrow$ {
name: new_stack(x, y)
preconditions: holding$(x) \wedge$free$(y) \wedge$goodtower$(y) \wedge$Gon$(x, y) \wedge$dck_holding(x)
effect: arm_empty$\wedge$on$(x, y)\wedge\neg$holding$(x)\wedge\neg$free$(y)\wedge$free$(x)\wedge\neg$dck_holding$(x)\wedge$ goodtower(x) } $\triangleright$ Places object x on top of object y, if and only if there isn't anything on top of y and our arm is holding x.

2.2.1 The encoding algorithm

The algorithm used for extending a domain is then fairly straightforward. Domain predicates are extended by the new memory and ATB state predicates with appropriate amounts of variables, the operator definitions are extended by new positive or negative preconditions and effects (Possibly creating multiple copies of an operator if it is associated with multiple transitions, or creating a new operator in case of an empty transition).

Algorithm 1 Encoding ATB-DCK to domain

Require: $Dom = (P, O), ATB = (Dom, S, M, T)$
Ensure: $Dom^c = (P^c, O^c)$
procedure EncodeDckToDomain(P, O, S, M, T)
$P^c \leftarrow P \cup M \cup encodePreds(S)$
$O^c \leftarrow \emptyset$
for all $t = (s_t, o_t, C_t, P_t, s'_t) \in T$ **do**
$preconditions(o_t)^- \leftarrow preconditions(o)^- \cup GetNegPrecs(s, C_t)$
$preconditions(o_t)^+ \leftarrow preconditions(o)^+ \cup GetPosPrecs(C_t)$
$effects(o_t)^- \leftarrow effects(o)^- \cup GetNegativeEffects(s', P_t)$
$effects(o_t)^+ \leftarrow effects(o)^+ \cup GetPositiveEffects(s, P_t)$
$O^c \leftarrow O^c \cup \{o_t\}$
end for
end procedure
procedure GetPosPrecs($initial\_state, Constraint$)
return $\{p|p \in Constraint\} \cup encodePreds(initial\_state) \cup encodePreds(Constraint)$
end procedure
procedure GetNegPrecs($Constraint$)
return $\{p|\neg p \in Constraint\}$
end procedure
procedure GetPositiveEffects($target\_state, Memory$)
return $\{p|p \in Constraint\} \cup encodePreds(target\_state) \cup \{m| + m \in Memory\}$
end procedure
procedure GetNegativeEffects($initial\_state, Memory$)
return $\{p|p \in Constraint\} \cup encodePreds(initial\_state) \cup \{m| - m \in Memory\}$
end procedure
procedure encodePreds($AttributedStates$)
Converts all input attributed states or attributed states in a constraint into predicates of the desired format ▷ Assume the input can be both a single state and a set of states
end procedure

As the algorithm is essentially just a single loop over all transitions within which it simply unifies preconditions and effects in constant time, it is easy to see

that the algorithm runs in $O(|T|)$ time where T are transitions.

Example (**Encoding of predicates**). Predicates before encoding ← { free(x), on(x, y), holding(x), arm_empty, on_table(x) }
Predicates after encoding ← { free(x), on(x, y), holding(x), arm_empty, on_table(x), goodtower(x), badtower(x), DCK_holding(x), Gontable(x), Gon(x, y), Gclear(x) }

Example (**Encoding to operators** - We list only a single operator for brevity).
Operator *pickup* before encoding ← {
name: $pickup(x)$
preconditions: $arm\_empty \wedge on\_table(x)$
effect: $\neg arm\_empty \wedge holding(x) \wedge \neg on\_table(x)$ }
Operator *pickup* after encoding ← {
name: $badtower\_pickup\_DCK\_holding(x, y)$
preconditions: $arm\_empty \wedge on\_table(x) \wedge badtower(x) \wedge Gon(x, y) \wedge goodtower(y) \wedge clear(y)$
effect: $\neg arm\_empty \wedge holding(x) \wedge \neg on\_table(x) \wedge DCK\_holding(x) \wedge \neg badtower(x)$
}

It can be observed that the change in the operator illustrates DCK-ATB utility perfectly. Rather than letting the planner attempt to find a sequence of actions including picking up a block we cannot possibly put in the correct place and subsequently having to drop the entire branch of computation, ATB-DCK prevents it entirely by modifying the definition of operators.

2.3 Encoding of ATB-DCK into Problems

One area in which the original work could be expanded for better clarity is a clear and explicit discussion on the topic of encoding ATB-DCK into problem instances, or more accurately, finding initial configuration. In its short form, the process can indeed be summarized as follows:

$$Prob' = (Obj \cup Obj', Init \cup M^I \cup \{encPred(s) | s \in S^I\}, Goal)$$

where Obj' are problem-specific objects, S^I and M^I come from some initial configuration, and the original problem is defined as $(Obj, Init, Goal)$. In symbolic terms, this description shall suffice, yet for implementation purposes, one major question remains unanswered - How exactly are problem-specific objects obtained?

Recall our definition of initialization rules for individual attributed memory or state predicates. Once given a problem instance with an initial state and objects, we can use these objects and possibly initial predicates to substitute any variables present in a given initialization rule for objects. We refer to these substitutions as variable assignments. Once a variable assignment that satisfies a given initialization rule (taking the form of a Horn clause) is found, a new predicate with found objects can be encoded into the initial state. The problem arises when we want to allow problem-specific objects.

Problem-specific objects arise as a natural consequence of one of the possible initialization rules - a cardinality query. A cardinality query takes the form

$$count(c, b_i, F(b_1, ..., b_q)), (1 \leq i \leq q)$$

where c represents the counted amount of instances of the formula $F(b_1, ..., b_q)$ which hold true for any fixed variable b_i. This rule can result in a variable assignment containing numbers, which weren't present in the original problem definition. Depending on the rule specifics, these new numbers can result in new objects for which some of our new predicates hold in the initial state. As such, the definition of ATB-DCK and its initialization rules allows attributed states or memory to take the form of counters or utilize numbers in some way. For some problems, modeling counters may greatly improve efficiency, such as making sure a truck only moves with a certain amount of objects, so as to prevent inefficient trips.Similarly, these rules may also contain a user-defined constant. For both of these cases, problem-specific objects need to be created.

The process of finding problem-specific objects can be combined with finding the initial states of attributed states or memory. In short, existing problem objects along with initial state predicates of a problem are treated as a knowledge base. The inference rules of our states and memory are then added to this knowledge base, after which we can trigger the inference mechanism by querying all of our attributed state and memory predicates. The inference process generates all possible variable assignemnts for a given query. These variable assignments to a given predicate (represented by the query) can then be encoded into the problem. In case a number or a constant previously not present in the list of problem objects appears in the variable assignment, they are encoded into the list of objects as well. Logical programming[30] are suited particularly well to this inference.

It can be observed that, unlike objects in regular problem definition, problem-specific objects in enhanced problems can only appear if and only if they occur in the initial state for some predicates. Allowing ones that don't occur provides no guarantee that these new objects will only be used by attributed memory or states (or at all for that matter), thus changing the problem, rather than just providing control information to be removed from the plan later.

To illustrate how a knowledge base might be assembled, our blocksworld planning problem from the first chapter1.2 might have a knowledge base that looks as in the figure 2.2 after implemented in Prolog.

This knowledge base would then infer goodtower(a), badtower(b), badtower(c), gon(b, a), gon(c, b), mStacked(b), mStacked(c) (We omit possible freeBot and freeTop, since those predicates only serve as helper predicates to shorten the rules. Prefixes like 'init_rule' were also omitted).

2.4 Extracting a solution

Finally, it remains to show that these algorithms and modifications will not only result in a valid planning task, but also generate only valid solutions, which can then be decoded into a valid solution for the original planning task.

```
% Initialization rules of states and memory for encoding to problems
init_rule_atb_obj(a).
init_rule_atb_obj(b).
init_rule_atb_obj(c).
init_rule_on_table(a).
init_rule_on_table(b).
init_rule_on(c, a).
init_rule_arm-empty.
goal_rule_on(b, a).
goal_rule_on(c, b).
init_rule_goodtower(V0O0) :- init_rule_on-table(V0O0),goal_rule_on-table(V0O0).
init_rule_goodtower(V0O1) :- init_rule_on-table(V0O1),init_rule_freeBot(V0O1).
init_rule_goodtower(V0O2) :- init_rule_on(V0O2,O2A0V1),init_rule_goodtower(V0O2),init_rule_gon(V0O2,O2A0V1).
init_rule_goodtower(V0O3) :- init_rule_on(V0O3,O3A0V1),init_rule_goodtower(V0O3),init_rule_freeBot(V0O3),init_rule_freeTop(O3A0V1).
init_rule_badtower(V0O0) :- init_rule_on-table(V0O0),goal_rule_gon(V0O0,_).
init_rule_badtower(V0O1) :- init_rule_on(V0O1,O1A0V1),goal_rule_on-table(V0O1).
init_rule_badtower(V0O2) :- init_rule_on(V0O2,O2A0V1),init_rule_gon(O2A1V0,O2A0V1),V0O2\= O2A1V0.
init_rule_badtower(V0O3) :- init_rule_on(V0O3,O3A0V1),init_rule_gon(V0O3,O3A1V1),O3A0V1 \= O3A1V1.
init_rule_badtower(V0O4) :- init_rule_on(V0O4,O4A0V1),goal_rule_clear(V0O4).
init_rule_badtower(V0O5) :- init_rule_on(V0O5,O5A0V1),init_rule_badtower(O5A0V1).
init_rule_dck_holding(V0O0) :- init_rule_holding(V0O0).
init_rule_gon(V0O0,V1O0) :- goal_rule_on(V0O0,V1O0).
init_rule_mStacked(V0O0) :- init_rule_gon(V0O0,_).
init_rule_freeBot(V0O0) :- \+ goal_rule_clear(V0O0),\+ init_rule_gon(_,V0O0).
init_rule_freeTop(V0O0) :- \+ goal_rule_on-table(V0O0),\+ init_rule_gon(V0O0,_).

```

Figure 2.2 Blocksworld Knowledge Base

Definition 16 (ATB-DCK planning task solution)**.** *Let* $PT^C = (Dom_{PT}, Prob_{PT}, C, C^I)$ *be an enhanced ATB-DCK planning task with initial state* I *and* T *being a set of transitions defined in* C*. A sequence of grounded transitions* $t_1, ..., t_n \in T$ *solves* PT^C *if and only if their consecutive application starting in* I *and* C^I *results in a planning state satisfying the goal defined in* $Prob_{PT}$*.*

Furthermore, any such solution to an ATB-DCK planning task can be transformed into a solution using only the original operators by simply removing any steps involving empty operators, replacing transitions with their associated operators, and possibly reducing the number of arguments to their original amount for corresponding operators (Any such reduction will need to either know the variable mapping or use the ordering of variables to determine which should be cut). Seeing as transitions can only expand the preconditions of actions, while only modifying the effects by memory or attributed state predicates, it then follows that such a solution will be a valid solution to the original problem. In short, ATB-DCK planning task may not always entail all valid solutions to the original problem, but any solution valid in the ATB-DCK planning task is also a valid solution to the original problem.

3 ATB-DCK editor

The core objective of this thesis was to create a tool that allows the user to design and programmatically encode ATB-DCK into domains and problems using a visual interface. The associated work is currently available online at `https://aiplanner-29862.web.app/`. This chapter shall describe the used technologies, user options, and data structure of the application.

3.1 Vue.js

The entire user interface of the web app utilizes the javascript[31] framework Vue.js[32]. A popular framework allowing us to organize code into views and components, as well as significantly easing many rendering, responsivity, and reactivity issues. Where possible, we also utilized Typescript[33], a Javascript syntax extension allowing for better type safety, making projects more readable and less error-prone. Lastly, The visual library Vuetify[34] and the local development server Vite[35] proved invaluable during development as well.

3.2 Developer dependencies

During development, Node.js[36] runtime for javascript was used. To manage our dependencies, we used the npm[37] registry and CLI tools. Other than that, prettier[38], mitt[39], lodash[40], eslint[41], and pinia[42] were all used to either ease development, or create various utilities invisible to the user, but nonetheless crucial.

3.3 Firebase

Firebase[43] is a cloud service created by Google. The development platform is responsible for our database - a document database called Firestore - our API endpoints, their secure exposure, and calling. Furthermore, a Google-provided cloud backend provides the project with easy and secure access to the user's Google account necessary for the management of multiple domains and problems in the database. Another invaluable feature provided by Firebase are backend emulators, which allow us to locally run all functionality provided by the Firebase servers with only minor differences. A feature that significantly sped up development. Lastly, as of today, Firebase provides all server functionality with a sizable free tier, making the development and maintenance costs non-existent.

3.4 The encoding process

Luckily for us, the definition of ATB-DCK is already mostly usable by an application, requiring minimal changes. This section will briefly highlight where our application-specific implementation differs from the definition and where it stays the same.

3.4.1 The data structure

During the definition of ATB-DCK, a single root object associated with one specific domain is created. This root object closely resembles the definition of ATB-DCK in that it contains an array of initialization rule objects and a string representing these rules in the Prolog language, as well as an object representing the domain for which it was assembled. Attributed state and memory objects contain

```
export interface AttributedDCK {
  domain: Domain,
  states: AttributedState[],
  memory: AttributedMemory[],
  transitions: AttributedTransition[],
  initRules: AttributedInitRule[],
  prologDomainInit: string,
}
```

Figure 3.1 ATB-DCK structure

nothing peculiar, they simply ask the user to specify the number of variables and name, possibly also holding a list of specific variable names to ease the autofilling of names during ATB-DCK definition process. Attributed Transitions are somewhat

```
export interface AttributedState {
  name: string,
  numOfVars: number,
  specificVars?: string[],
}
```

Figure 3.2 Attributed state structure

more complicated, seeing as they contain objects representing both the origin and the target state, the associated action, and an array of constraints. These constraints simply hold the name of a predicate, possibly its negation, variables to be used with the predicate, and a flag indicating the predicate being an effect of a transition or a precondition.

```
export interface AttributedTransition {
  originState: AttributedState,
  targetState: AttributedState,
  operator: Action,
  constraints: AttributedConstraint[],
}
```

Figure 3.3 Attributed transition structure

The design of Attributed Initialization rules was without a doubt one of the greater challenges during the development process, seeing as it needed to

encompass all the expressivity of Horn-clauses, pair the information up to actual predicates and variables, allow for the use of constants or cardinality rules, all while shielding the user from the need to utilize what could essentially be the complexity of programming those rules by hand in some language. The resulting

```
export interface AttributedInitRule {
  rulePredicate: RulePredicate,
  orClause: LogicalOrRule[],
  hasSimpleValue: boolean,
  simpleDefaultValue?: "True" | "False" | "Constant",
}
export interface LogicalOrRule {
  anyVariables: string[],
  andClause: RulePredicate[],
  prologFunctions: PrologFunction[],
}
```

Figure 3.4 Initialization rules structure

structure is fairly simple. It contains a possible simple default value for the predicate (constant, true for all objects, false for all objects), Predicate being initialized, and a nested object structure of logical or rules containing arrays of and rules, possibly allowing the user to add predefined logical functions (Such as the cardinality rule). These rules are subsequently compiled into a representation of rules in the Prolog language, which is offered to the user for editing and debugging (In a one-way manner), since some users might get confused by large chunks of logic being hidden behind a series of adaptive menus, preferring to utilize their existing Prolog expertise instead.

3.4.2 Encoding to domains

A very straightforward process that can mostly be summed up using the earlier-defined algorithm of encoding ATB-DCK to the domain. Once our language parser detects relevant parts of the domain raw text, these parts are either extended (the predicates section) or replaced (actions for one or multiple transitions with corresponding associated operator). Any precondition or effect modifications are solved by simply wrapping the existing rules in an "and" rule and adding transition-specific ones.

3.4.3 Encoding to problems

A more difficult process driven mostly by the steps of our algorithm for encoding into problems defined earlier. It only uses initialization rules and their prolog representation. The key is to continually generate all possible solutions using an in-built Prolog engine included in the trealla[44] package. These solutions are then encoded in the PDDL format into their corresponding sections. This encoding also keeps track of newly generated variables, so as to not create duplicate objects.

3.5 User interface

In designing the visual interface, an emphasis was put on keeping it as simple as possible. As such, the interface can be broken down into only a few distinct components.

3.5.1 Management of files

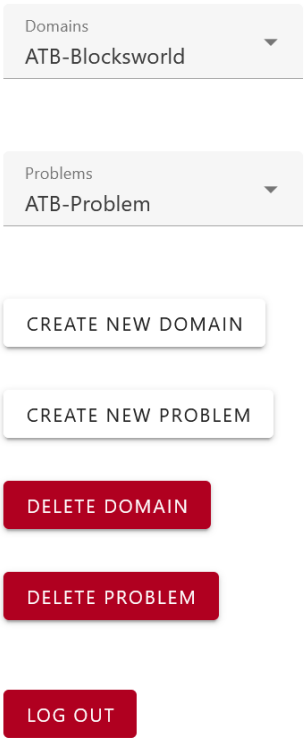


Figure 3.5 Domain and problem selection/creation/deletion

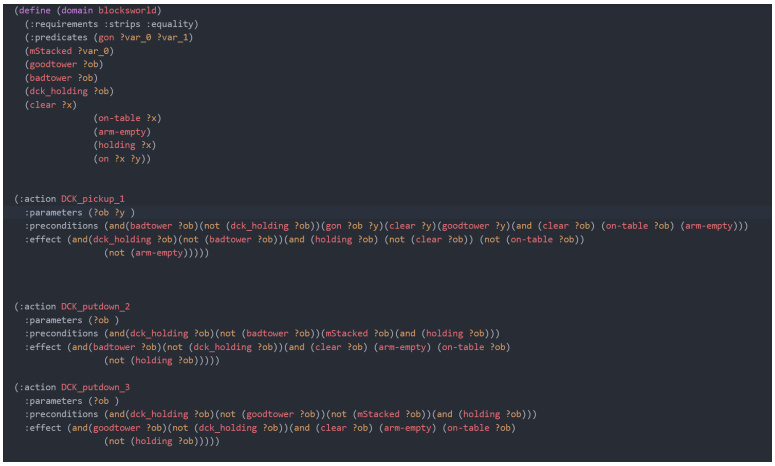
Should the user choose to log in using his Google account, a simple interface allows him to create and edit multiple different domains, delete them at will, associate problems with them, and keep a separate ATB-DCK for each of them. The provided functionality is fairly basic, only emulating the bare necessities of working with multiple files, but it allows the user to easier experiment with and change different ATB-DCK for different domains. Features of reverting file contents to one currently stored in a database or branching into a new domain/problem are also supported using the top bar.

Figure 3.6 Domain utility bar

Figure 3.7 Problem utility bar

3.5.2 File editor

Powered by codemirror[45] and its associated lezer parser[46], a code editor for the PDDL language is a key component of the user interface. Custom-written grammar for the language allows the application to decompose domains and problems into individual relevant parts, respond to user changes by changing selection options in ATB-DCK, and output a complete domain/problem with ATB-DCK encoded as needed. Syntax highlighting is supported for all basic features of PDDL[27].



```
(define (domain blocksworld)
  (:requirements :strips :equality)
  (:predicates (gon ?var_0 ?var_1)
  (mStacked ?var_0)
  (goodtower ?ob)
  (badtower ?ob)
  (dck_holding ?ob)
  (clear ?x)
              (on-table ?x)
              (arm-empty)
              (holding ?x)
              (on ?x ?y))


(:action DCK_pickup_1
  :parameters (?ob ?y )
  :preconditions (and(badtower ?ob)(not (dck_holding ?ob))(gon ?ob ?y)(clear ?y)(goodtower ?y)(and (clear ?ob) (on-table ?ob) (arm-empty)))
  :effect (and(dck_holding ?ob)(not (badtower ?ob))(and (holding ?ob) (not (clear ?ob)) (not (on-table ?ob))
               (not (arm-empty)))))


(:action DCK_putdown_2
  :parameters (?ob )
  :preconditions (and(dck_holding ?ob)(not (badtower ?ob))(mStacked ?ob)(and (holding ?ob)))
  :effect (and(badtower ?ob)(not (dck_holding ?ob))(and (clear ?ob) (arm-empty) (on-table ?ob)
               (not (holding ?ob)))))

(:action DCK_putdown_3
  :parameters (?ob )
  :preconditions (and(dck_holding ?ob)(not (goodtower ?ob))(not (mStacked ?ob))(and (holding ?ob)))
  :effect (and(goodtower ?ob)(not (dck_holding ?ob))(and (clear ?ob) (arm-empty) (on-table ?ob)
               (not (holding ?ob)))))
```

Figure 3.8 Code editor

3.5.3 ATB-DCK form

A series of adaptive menus corresponding to the data structure mentioned in one of the previous sections. The options offered to the user in every menu change dynamically once the source domain file is changed.

Attributed states form simply asks for a name and the number of variables of the attributed state. More can be defined, existing ones can be deleted, and any changes made can become available in all other relevant forms upon saving.

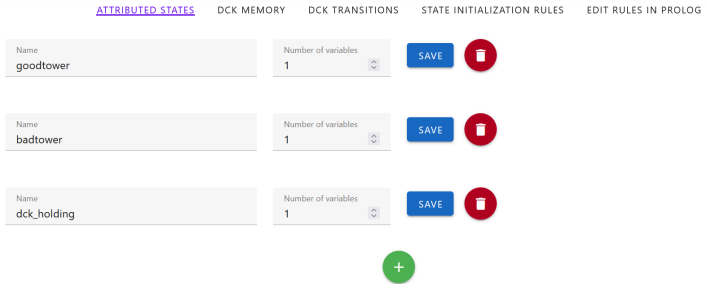


Figure 3.9 Attributed state form

Attributed memory form differs from the attributed state form only mildly. One important difference is the option to not encode a predicate to a problem. Such predicates simply serve to enhance the expressiveness of initialization rules, seeing as the user would need to define them in a purely DNF form composed of only basic predicates otherwise. These extra predicates may be necessary to significantly shorten the length of individual rules and increase readability.

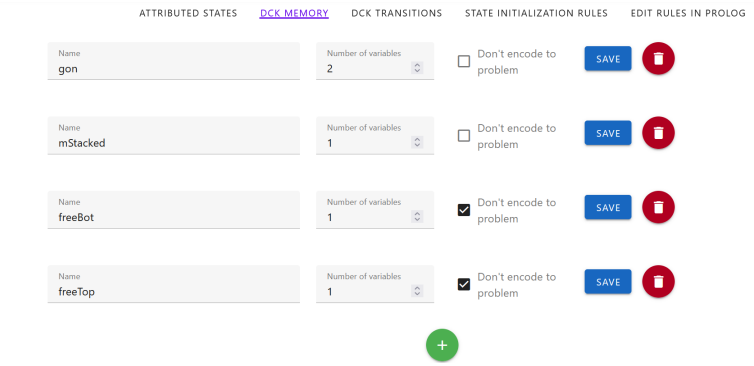


Figure 3.10 Attributed memory form

Attributed transitions form is slightly more complicated. It allows the selection of the origin and target state, along with the associated operator. The user can further add individual constraints specifying preconditions or effects of the transition, which can also be negated. It is up to the user to specify the names of individual variables and synchronize them with the associated action predicates. A dummy empty operator is also added to the list of possible operators. Constraints can come in the form of attributed states or memory, or ordinary domain predicates.

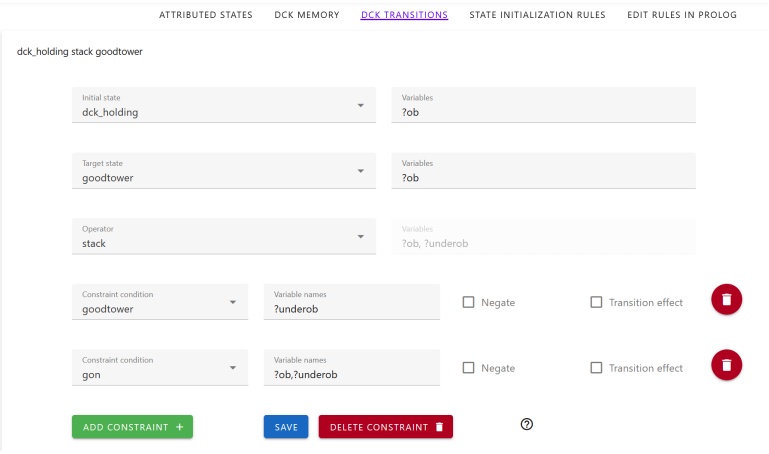


Figure 3.11 Attributed transition form

State initialization rules form can be somewhat obscure at first glance. There is exactly one rule for each attributed state or memory predicate. Each rule can either hold a simple value (constant/true for all objects/false for all objects), or value derived from some logical formula in a DNF form. This logical formula can

contain attributed states, memory predicates, ordinary predicates, queries for goal states or negations can contain 'any' variables (variables whose assignment does not matter, it is enough that an assignment exists), all of which can be added into individual 'AND' clauses. These 'AND' clauses are enclosed within 'OR' clauses, thus mimicking the structure of DNF. Individual rules can also contain function predicates, which is a set of predefined functions to which the user simply needs to provide a series of arguments (possibly nested predicates) and extract the result. These functions include min, max, and count.

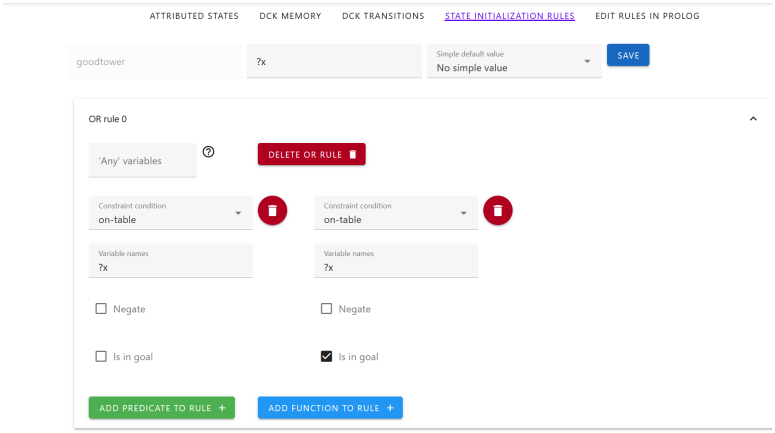


Figure 3.12 Initialization rule form

Prolog editor is primarily meant to provide the user with a debugging tool. Any changes made in the above-mentioned forms will rewrite the contents of this Prolog code, but the contents of this page will be used as rules for generating possible answers for queries when encoding to a problem, allowing for quick changes, as well as simply giving those versed in Prolog a better insight into how their initialization rules are used to encode into problems.

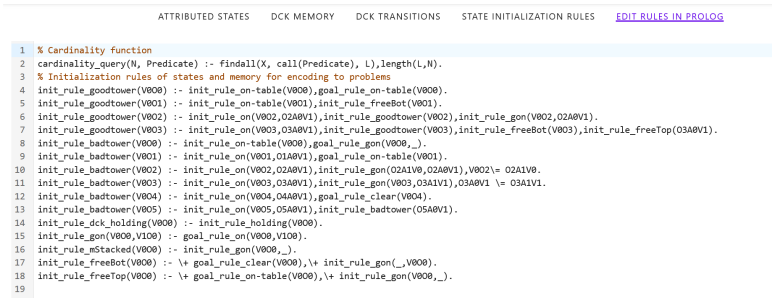


Figure 3.13 Prolog rule editor

3.6 Workflow example

In this section, we will demonstrate a basic workflow while using the application. As the usage of the tool expands, account and file management features might need to be used, but we will stick to only the bare minimum workflow for now.

- The first step will always be to input a domain. This can be done either on the domain tab (if not logged in), or by selecting the domain of choice from a dropdown in the menu (possibly creating a new one). This can be done using the left menu bar as seen in figure 3.5.
- Once the domain is defined in the code editor from figure 3.8, the top utility bar allows us to edit the ATB-DCK using a toggle button.
- The ATB-DCK editor offers several tabs. In general, it is best to first edit attributed states and memory as can be seen in figures 3.9 and 3.10, then proceed to editing the transitions as seen in figure 3.11. Once this is done, a valid ATB-DCK domain can be encoded, if only the domain is needed, the next step can be skipped.
- Editing the rules in the 'State initialization rules' tab can be somewhat tricky, as it's easy to make mistakes. Variable names need to be synchronized across predicates, and the rules need to be defined in the correct order, since they are translated into Prolog code after. Individual or (disjunction) rules can be added, which consist of and (conjunction) rules. 'Any' variables can be added to a special field, which correspond to Prolog '_' symbol. Variables start with '?', while constants don't. An example of rule definitions can be seen in figure 3.12.
- The code compiled from the rules can be viewed in the Prolog editor tab. Whatever is in this tab once we leave to encode will be used for inference.
- Once our ATB-DCK is done, we can switch back over to the editor, where a button in the upper right corner allows us to encode ATB-DCK into a domain/problem.

4 Experimental evaluation

This chapter aims to design an ATB-DCK for a new domain and then test its effects on the performance of plans. The effects would be demonstrated on the Blocksworld domain, but it had already been tested in the original paper, along with several other domains[1]. For this reason, we shall refer the reader to the original paper in case a more detailed analysis of performance is desired. An important finding to highlight is that ATB-DCK is only effective for some domains, seeing as it can potentially introduce too many new predicates, causing additional preprocessing overhead which might outweigh potential performance improvements. Additionally, many different variations of ATB-DCK and associated initialization rules may exist for any particular domain, only some of which may be effective.

4.1 Domain description

The chosen domain for our new ATB-DCK is called Transport, which was taken from IPC 2014[9], where it served to evaluate competing planners on the sequential planning track. The domain was chosen for its similarity to Blocksworld in its abstract model, but it holds one unique feature over Blocksworld - its planning tasks attempt to minimize the cost of traveled roads.

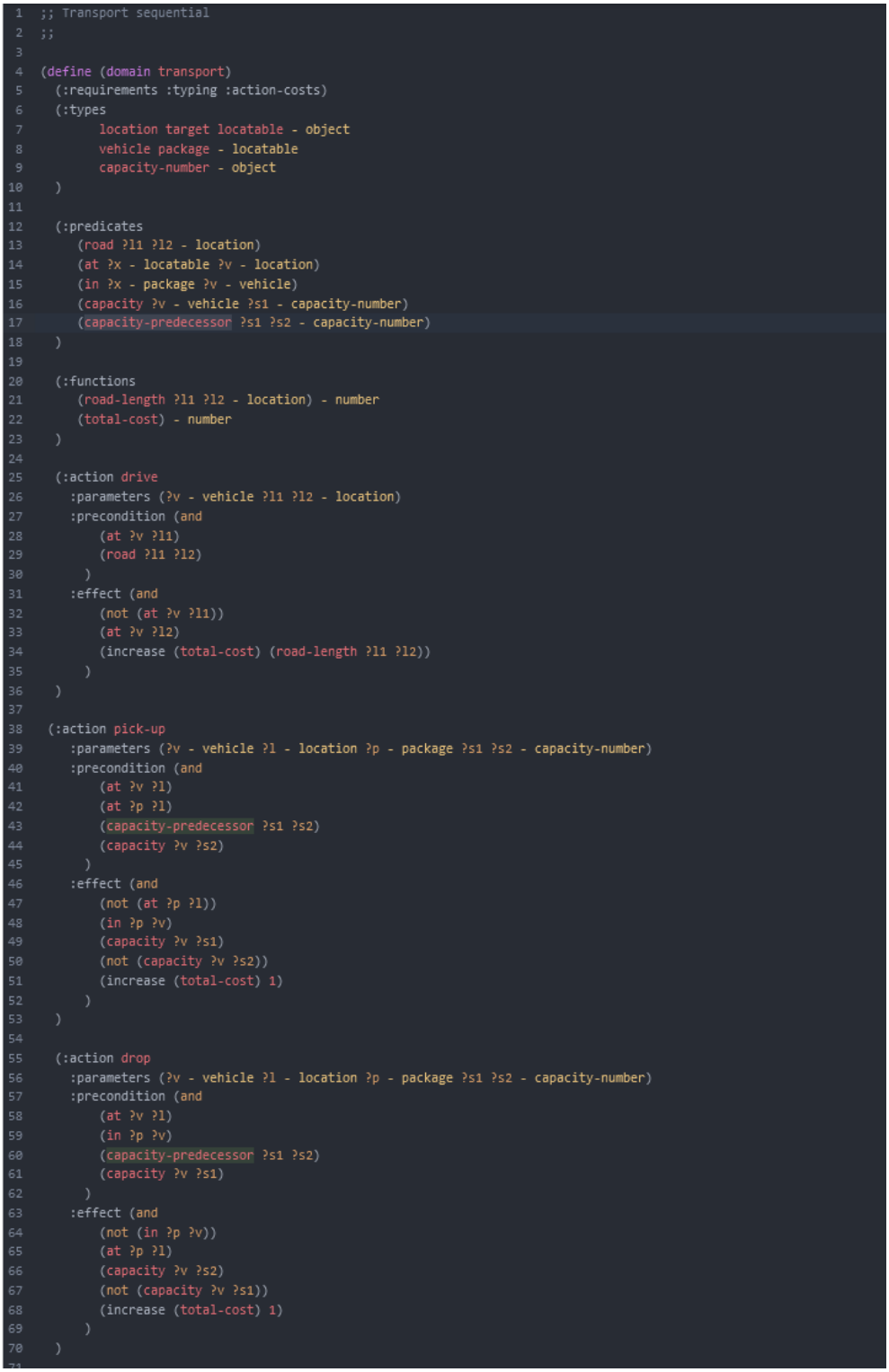
The domain uses even fewer operators than Blocksworld:

- drive
- pick-up
- drop

As the names of operators suggest, the domain concerns itself with finding a plan for one or more vehicles to transport a number of objects between different locations. In addition, the domain contains several predicates:

- road - which connects different locations
- at - informing us of which object is at what location
- in - is a version of at, only telling us the presence of objects in vehicles, rather than locations
- capacity - a predicate indicating which vehicle can carry how many objects
- capacity-predecessor - a numeric predicate serving as a counter for increasing or decreasing the occupied capacity of vehicles

Summarizing the preconditions and effects of actions is then very simple. Drive requires a road and a vehicle to be in one of the road-connected locations before moving both the vehicle and any object it's carrying in between locations, pick-up requires the car to not be at capacity and for an object to be present at a location before loading it to the car, and drop simply leaves the specified object held by a vehicle at its current location. Full PDDL description of the domain can be seen in the figure 4.5.



```
;; Transport sequential
;;

(define (domain transport)
  (:requirements :typing :action-costs)
  (:types
        location target locatable - object
        vehicle package - locatable
        capacity-number - object
  )

  (:predicates
     (road ?l1 ?l2 - location)
     (at ?x - locatable ?v - location)
     (in ?x - package ?v - vehicle)
     (capacity ?v - vehicle ?s1 - capacity-number)
     (capacity-predecessor ?s1 ?s2 - capacity-number)
  )

  (:functions
     (road-length ?l1 ?l2 - location) - number
     (total-cost) - number
  )

  (:action drive
    :parameters (?v - vehicle ?l1 ?l2 - location)
    :precondition (and
        (at ?v ?l1)
        (road ?l1 ?l2)
      )
    :effect (and
        (not (at ?v ?l1))
        (at ?v ?l2)
        (increase (total-cost) (road-length ?l1 ?l2))
      )
  )

 (:action pick-up
    :parameters (?v - vehicle ?l - location ?p - package ?s1 ?s2 - capacity-number)
    :precondition (and
        (at ?v ?l)
        (at ?p ?l)
        (capacity-predecessor ?s1 ?s2)
        (capacity ?v ?s2)
      )
    :effect (and
        (not (at ?p ?l))
        (in ?p ?v)
        (capacity ?v ?s1)
        (not (capacity ?v ?s2))
        (increase (total-cost) 1)
      )
  )

  (:action drop
    :parameters (?v - vehicle ?l - location ?p - package ?s1 ?s2 - capacity-number)
    :precondition (and
        (at ?v ?l)
        (in ?p ?v)
        (capacity-predecessor ?s1 ?s2)
        (capacity ?v ?s1)
      )
    :effect (and
        (not (in ?p ?v))
        (at ?p ?l)
        (capacity ?v ?s2)
        (not (capacity ?v ?s1))
        (increase (total-cost) 1)
      )
  )
```

Figure 4.1 Transport domain

Overall, the abstract model of the Transport domain is simpler than that of blocksworld, do note, however, that it utilizes counters and the domain specification as used by IPC 2014 measures and minimizes the total cost of the solution, which is why it will be interesting to see not only how the runtime is affected, but also whether ATB-DCK can somehow help or harm the overall metric.

4.2 ATB-DCK design

In the process of designing ATB-DCK for this domain, several variants were attempted. In the end, only a simplified version of our Blocksworld ATB-DCK had proven viable. Attempts were made to create an ATB-DCK counting the numbers of packages at specific locations and to make car trips more efficient, but any use of such counters for both cars and objects that had been tried proved to be unrealistically large after being encoded to a problem. Since problems for this particular domain tend to contain a fair amount of objects and a large number of predicates in the initial state, most of the tried ATB-DCK bloated the problem description exponentially, which was a problem for both our encoder (which at the time of writing runs on only a free-tier of hardware while being subjected to browser memory and processing limitations), and planners, for which viable ATB-DCK using location or object counters either negated any performance improvements due to size or worsened it. It may be possible such an ATB-DCK exists, but none was found during our attempts.

After moving back to simpler designs, an ATB-DCK very similar to our Blocksworld running example naturally arose. Any package can either be at a bad location, currently loaded, or at a good location. The only way for a package to be at a good location is for it to either already be there, or to be dropped off by a vehicle. A package can never move from a good location. Packages can get away from a bad location by being unloaded, but since we might want to allow the passing of packages in between vehicles or temporary dropoffs, it can also be dropped at a bad location.

In order to drop a package at a good location, the only additional constraint is for the package to have it as a goal location, which we shall encode using ATB-memory. Both other transitions (To and from bad location) will contain the same constraint, only negated. In addition to the new goal location memory predicate,

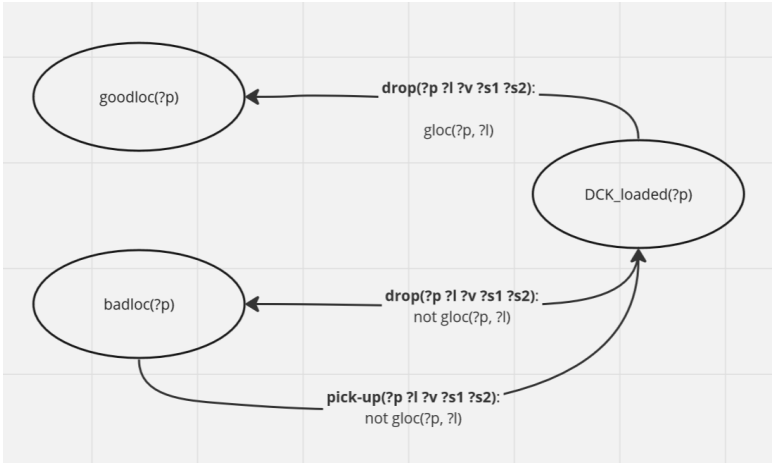


Figure 4.2 ATB-DCK 1

we also introduce additional shortcut predicate for use in the initialization rules. Notably the 'iscar' predicate, whichprevents the encoder from assigning badloc to vehicles. The initialization rules shall be as follows:

- $gloc(?p, ?l) \leftarrow goal\_at(?p, ?l)$

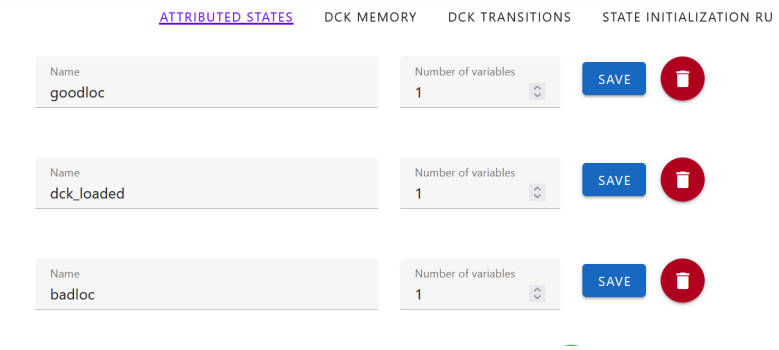


Figure 4.3 Transport state definition

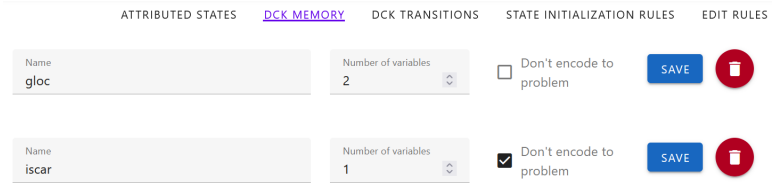


Figure 4.4 Transport memory definition

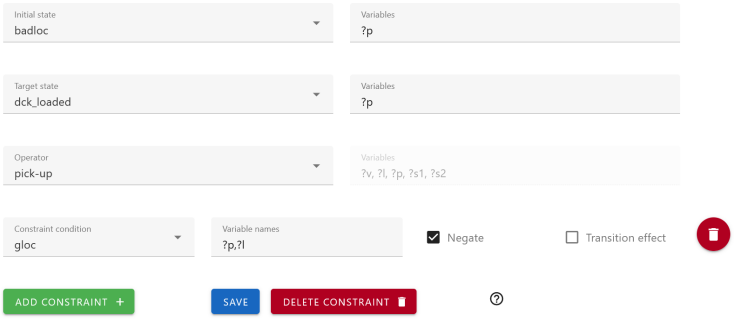


Figure 4.5 A transition definition example

- $goodloc(?p) \leftarrow gloc(?p, ?l) \wedge at(?p, ?l)$
- $DCK\_loaded(?p) \leftarrow in(?p, ?v)$
- $iscar(?p) \leftarrow capacity(?p, ?x)$
- $badloc(?p) \leftarrow gloc(?p, ?l) \wedge \neg at(?p, ?l) \wedge \neg iscar(?p)$

We shall refer to this ATB-DCK as ATB-DCK 1. After performing the experiments, a variation of this ATB-DCK was tried and to astonishing results. Simply deleting the option of dropping a package at a badloc - denoted as ATB-DCK 2 - had drastically pruned the search tree. For design of our second ATB-DCK, see figure 4.7.

4.3 ATB-DCK experimental evaluation

For experimentation, a publicly available docker image of the lapkt[47] planning toolkit had been used. The toolkit had been selected for its variety of supported

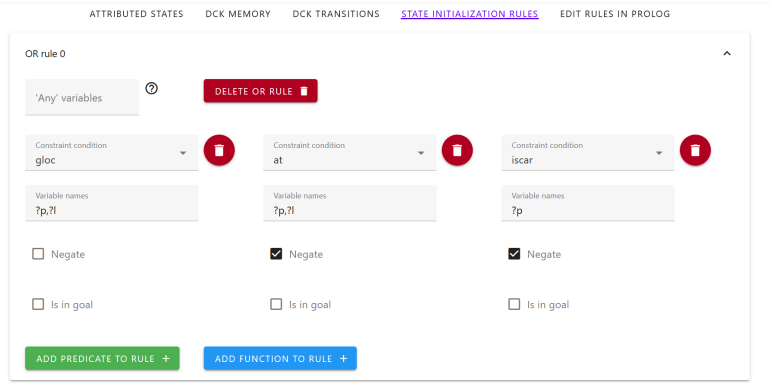


Figure 4.6 A badloc rule example

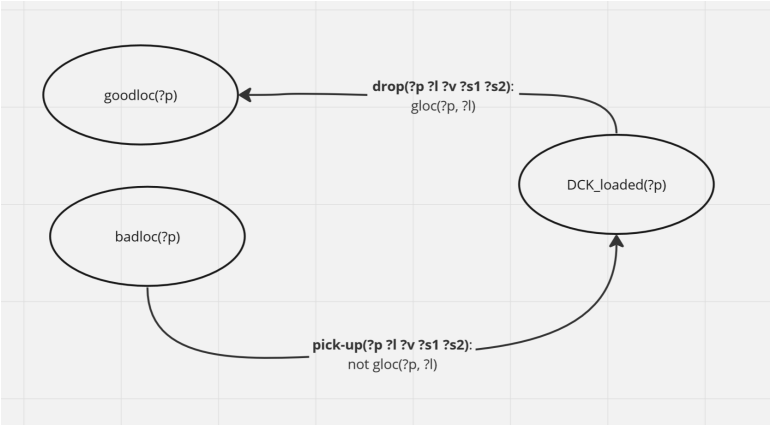


Figure 4.7 ATB-DCK 2

planners and supporting updated versions of planners which were used in the IPC 2014[47], from which our domain and problems were taken as well. Most notably, the planner BFS_F had been selected for both its good overall performance and support of the solution cost metric. BFS_F stands for Greedy Best First Search algorithm with f(n) = novelty(n) as described in [48]. Experiments were run on a 12th Gen Intel(R) Core(TM) i7-12700H 2.70 GHz processor with 32GB RAM. A total of 10 problem instances were measured with degrees of nodes being 4, minimal distance equal to 100, size of 1000, and 4 trucks. The problem instances vary in size, but mostly in how the nodes are arranged, both within a city (a cluster of connected nodes), and among cities.

As can be observed from the presented results, ATB-DCK can have an impact

Packages	Cities	Nodes	Original time	Original cost	ATB-DCK 1 time	ATB-DCK 1 cost	ATB-DCK 2 time	ATB-DCK 2 cost
25	1	50	25.13s	246	29.85s	202	0.49s	184
30	1	53	66.36s	294	72.82s	271	0.96s	260
25	1	53	35.49s	242	39.85s	232	0.61	221
25	2	68	1297.38s	563	452.99s	532	1.53s	524
30	2	67	360.55s	763	1421.97s	826	2.12s	627
25	2	68	1293.43s	563	457.09s	532	1.56s	524
25	3	68	426.57s	379	519.91s	393	1.68s	363
30	3	67	805.40s	502	897.74s	431	2.49s	411
25	3	68	434.39s	379	525.26s	393	1.59s	363
25	3	66	261.98s	472	756.55s	390	1.80s	357

Table 4.1 Comparison of runtimes and costs

on both performance and solution cost. Paradoxically, while ATB-DCK 1 is primarily meant as a technique to improve the performance of planners, it mostly introduces a minor slowdown compared to the original encoding, but interestingly, a fairly consistent improvement in solution cost can be observed. This starkly contrasts with ATB-DCK 2, which majorly speeds up planning time and minimizes solution costs. For every problem instance, ATB-DCK 2 improves the planning time more than ten times, which brings us to an interesting observation.

Designing an ATB-DCK is difficult and might not yield any results. Even minor changes might lead to significant speedups, performance losses, or even prevent finding a solution at all, as reinforced by the observations in the original paper[1]. One area of research waiting to be explored is ATB-DCK design. At the moment, any design relies on the creativity of a domain expert. Much like the field of heuristic research, finding more general ATB-DCK designs or ways of automatic ATB-DCK generation could prove vital in improving classical planning as a whole. Our results also contain one other interesting finding.

At the time of writing, very little research had been done in regards to improving quality of plans or other planning metrics using ATB-DCK, but as our results demonstrate the plan cost (which is a metric to be minimized during planning for this domain) had been improved in seven out of ten cases for ATB-DCK 1, and all cases for ATB-DCK 2. A well-designed ATB-DCK for the right domain might have a significant impact on not only its performance during planning.

Conclusion

After defining and introducing the topic of classical planning along with touching upon historical and contemporary developments in the field, this thesis reitorated the original definition of ATB-DCK. It then expanded this definition by more detailed description of encoding ATB-DCK into problems, and proceeded to briefly describe the software created to supplement this thesis. The described project can be found in the following GitHub repository: `https://github.com/Neathac/AIPlanner` and available at the web address `https://aiplanner-29862.web.app/`. The project intends to provide a simple text editor allowing users to try out the process of creating and encoding ATB-DCK for themselves without any development setup needed. It also aims to illustrate how practical ATB-DCK encoders might work for possible future projects. Finally, we designed a new ATB-DCK and illustrated its effects on a few experiments, which led to some interesting results, pathing the way to possible future research.

Bibliography

[1] Lukáš Chrpa et al. "Attributed Transition-Based Domain Control Knowledge for Domain-Independent Planning". In: *IEEE Transactions on Knowledge and Data Engineering* 34.9 (2022), pp. 4089–4101. DOI: 10.1109/TKDE.2020.3037058.

[2] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. The Morgan Kaufmann Series in Artificial Intelligence. Elsevier Science, 2004. ISBN: 9780080490519. URL: https://books.google.cz/books?id=uYnpze57MSgC.

[3] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson series in artificial intelligence. Pearson, 2021. ISBN: 9781292401133. URL: https://books.google.cz/books?id=B4xczgEACAAJ.

[4] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Always learning. Pearson, 2016. ISBN: 9781292153964. URL: https://books.google.cz/books?id=XS9CjwEACAAJ.

[5] John Slaney and Sylvie Thiébaux. "Blocks World revisited". In: *Artificial Intelligence* 125.1 (2001), pp. 119–153. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(00)00079-5. URL: https://www.sciencedirect.com/science/article/pii/S0004370200000795.

[6] Henry A Kautz, Bart Selman, et al. "Planning as Satisfiability." In: *European Conference on Artificial Intelligence*. Vol. 92. Citeseer. 1992, pp. 359–363.

[7] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. "Planning as satisfiability: parallel plans and algorithms for plan search". In: *Artificial Intelligence* 170.12-13 (2006), pp. 1031–1080.

[8] Jussi Rintanen. "Madagascar: Scalable planning with SAT". In: *Proceedings of the 8th International Planning Competition (IPC-2014)* 21 (2014), pp. 1–5.

[9] Mauro Vallati et al. "The 2014 international planning competition: Progress and trends". In: *Ai Magazine* 36.3 (2015), pp. 90–98.

[10] Nils Froleyks et al. "SAT Competition 2020". In: *Artificial Intelligence* 301 (2021), p. 103572. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2021.103572. URL: https://www.sciencedirect.com/science/article/pii/S0004370221001235.

[11] Avrim L Blum and Merrick L Furst. "Fast planning through planning graph analysis". In: *Artificial intelligence* 90.1-2 (1997), pp. 281–300.

[12] Alfonso Gerevini and Ivan Serina. "LPG: A Planner Based on Local Search for Planning Graphs with Action Costs." In: *AI Planning & Scheduling*. Vol. 2. 2002, pp. 281–290.

[13] Adam Amos-Binks, Colin Potts, and R Young. "Planning graphs for efficient generation of desirable narrative trajectories". In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 13. 2. 2017, pp. 146–153.

[14] Stephen Ware and R Michael Young. "Glaive: a state-space narrative planner supporting intentionality and conflict". In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 10. 1. 2014, pp. 80–86.

[15] Pavel Surynek. *Constraint Programming in Planning*. 2008.

[16] Roman Barták and Daniel Toropila. "Reformulating Constraint Models for Classical Planning." In: *The Florida AI Research Society Conference*. 2008, pp. 525–530.

[17] Rina Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

[18] Malte Helmert. "A Planning Heuristic Based on Causal Graph Analysis." In: *International Conference on Automated Planning and Scheduling*. Vol. 16. Citeseer. 2004, pp. 161–170.

[19] Leo Macdonald and Jess Enright. *Heuristic Search in Planning*. 2004.

[20] Blai Bonet and Héctor Geffner. "Planning as heuristic search". In: *Artificial Intelligence* 129.1-2 (2001), pp. 5–33.

[21] Stefan Edelkamp. "Planning with Pattern Databases". In: *Proceedings of the 6th European Conference on Planning (ECP-01)* (Jan. 2002).

[22] Stefan Edelkamp. "Symbolic Pattern Databases in Heuristic Search Planning." In: *AI Planning & Scheduling*. 2002, pp. 274–283.

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

[24] Bart Selman and Carla P Gomes. "Hill-climbing search". In: *Encyclopedia of cognitive science* 81 (2006), p. 82.

[25] Alfonso Gerevini et al. "Local search techniques for temporal planning in LPG". In: *Proceedings of the Thirteenth International Conference on International Conference on Automated Planning and Scheduling*. 2003, pp. 62–71.

[26] Minh Binh Do and Subbarao Kambhampati. "Planning Graph-based Heuristics for Cost-sensitive Temporal Planning." In: *AI Planning & Scheduling*. 2002, pp. 3–12.

[27] Constructions Aeronautiques et al. "Pddl| the planning domain definition language". In: *Technical Report, Tech. Rep.* (1998).

[28] Richard E Fikes and Nils J Nilsson. "STRIPS: A new approach to the application of theorem proving to problem solving". In: *Artificial intelligence* 2.3-4 (1971), pp. 189–208.

[29] Lukáš Chrpa and Roman Barták. "Guiding planning engines by transition-based domain control knowledge". In: *Fifteenth International Conference on the Principles of Knowledge Representation and Reasoning*. 2016.

[30] Robert Kowalski. "Logic programming". In: *Handbook of the History of Logic*. Vol. 9. Elsevier, 2014, pp. 523–569.

[31] *Javascript*. Accessed: 2024. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript`.

[32] *Vue.js - The Progressive JavaScript Framework v3.0*. Version 3.4.21. Accessed: 2024. 2014. URL: https://vuejs.org/.

[33] *Typescript*. Version 4.7.4. Accessed: 2024. URL: https://www.typescriptlang.org/.

[34] *Vuetify*. Version 3.5.7. Accessed: 2024. URL: https://vuetifyjs.com/en/.

[35] *Vite*. Version 4.0.0. Accessed: 2024. URL: https://vitejs.dev/.

[36] *Node.js*. Version 20.11.1. Accessed: 2024. URL: https://nodejs.org/en.

[37] *NPM*. Version 9.5.0. Accessed: 2024. URL: https://www.npmjs.com/.

[38] *Prettier*. Version 2.7.1. Accessed: 2024. URL: https://prettier.io/.

[39] *mitt*. Version 3.0.0. Accessed: 2024. URL: https://github.com/developit/mitt.

[40] *Lodash*. Version 4.17.21. Accessed: 2024. URL: https://lodash.com/.

[41] *Eslint*. Version 8.22.0. Accessed: 2024. URL: https://eslint.org/.

[42] *Pinia*. Version 2.0.28. Accessed: 2024. URL: https://pinia.vuejs.org/.

[43] *Firebase*. Version 9.17.1. Accessed: 2024. URL: https://firebase.google.com/.

[44] *Trealla*. Version 0.17.36. Accessed: 2024. URL: https://github.com/guregu/trealla-js.

[45] *Codemirror*. Version 6.1.1. Accessed: 2024. URL: https://codemirror.net/.

[46] *Lezer parser*. Version 1.2.2. Accessed: 2024. URL: https://lezer.codemirror.net/.

[47] Miquel Ramirez, Nir Lipovetzky, and Christian Muise. *Lightweight Automated Planning ToolKiT*. http://lapkt.org/. Accessed: 2024. 2015.

[48] Nir Lipovetzky and Hector Geffner. "Width and serialization of classical planning problems". In: *European Conference on Artificial Intelligence 2012*. IOS Press, 2012, pp. 540–545.

List of Figures

List of Tables