Master Thesis

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Graphical Editor and Visualization of ATB-DCK for domain-independent automated planning

Martin Hruška

Supervisor: doc. RNDr. Lukáš Chrpa, Ph.D.
Field of study: Artificial Intelligence
Subfield: Automated Planning
May 2023

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Hruška** Jméno: **Martin** Osobní číslo: **474759**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Grafický editor a vizualizace ATB-DCK pro doménově nezávislé automatické plánování

Název diplomové práce anglicky:

Graphical Editor and Visualization of ATB-DCK for domain-independent automated planning

Pokyny pro vypracování:

Design a language that represents ATB-DCK (Attributed Transition-based Domain Control Knowledge) and implement its compilation to PDDL [1] (a language for representing planning problems [3]).
Design and implement text editor for ATB-DCK (with visual distinction of particular syntactical elements).
Design and implement graphical editor for editing and visualizing ATB-DCK in a "schematical" form (like finite automata).
Integrate at least two planners (e.g. LAMA, BFWS) into the editor
Specify ATB-DCK in several domains and evaluate it by comparing ATB-DCK enhanced problems to original one (CPU time, plan quality).

Seznam doporučené literatury:

[1] Lukás Chrpa, Roman Barták, Jindrich Vodrázka, Marta Vomlelová:
Attributed Transition-Based Domain Control Knowledge for Domain-Independent Planning. IEEE Trans. Knowl. Data Eng. 34(9): 4089-4101 (2022)
[2] Lukás Chrpa, Roman Barták:
Guiding Planning Engines by Transition-Based Domain Control Knowledge. KR 2016: 545-548
[3] Maria Fox, Derek Long:
PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. J. Artif. Intell. Res. 20: 61-124 (2003)

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. RNDr. Lukáš Chrpa, Ph.D. optimalizace CIIRC

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **14.09.2022** Termín odevzdání diplomové práce: **26.05.2023**

Platnost zadání diplomové práce: **19.02.2024**

doc. RNDr. Lukáš Chrpa, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání	Podpis studenta

© VUT v Praze, Design: VUT v Praze, VIC

Acknowledgements

I would like to thank my thesis supervisor doc. RNDr. Lukáš Chrpa, Ph.D., for his guidance throughout the entire thesis preparation.

Declaration

I declare that I have prepared the submitted thesis independently and that I have cited all the literature used.

Prague, May 26, 2023

ctuthesis t1606152353

Abstract

In domain-independent automated planning, the specification of the planning task is separated from its implementation, which is provided by domain-independent planning engines. This approach offers a high degree of flexibility because planners are thus able to solve any computational problem that can be converted into an automated planning task. However, the downside of this approach is that planning tasks' specifications often lack important contextual information based on the unique properties of the corresponding domains, which might negatively affect the performance of domain-independent planners in comparison to the performance of domain-specific algorithms designed for the same problems. To at least partially bridge the performance gap between planning engines and algorithms, the additional domain-specific information known as Domain Control Knowledge (DCK) could be constructed and encoded directly into the specifications of planning tasks.

Attributed Transition-Based Domain Control Knowledge (ATB-DCK) is a type of DCK that is represented by a finite-state automaton. Its states and transitions impose additional constraints on the applicability of actions defined in planning tasks' specifications and thus limit the number of permissible actions the planning engine can take at each step of the planning process. The main advantage of ATB-DCK is its simplicity and readability even for non-technical users.

The aim of this thesis is to simplify the process of encoding ATB-DCK into the planning task specification represented in the PDDL description language. The thesis first formulates a language to represent ATB-DCK in JSON data format that can be easily processed by computer programs. Then, it introduces a graphical editor for creating and visualizing ATB-DCK as a finite state automaton, which offers an intuitive way of constructing data representation of ATB-DCK. The graphical editor includes a text editor to modify ATB-DCK directly in its JSON data form. To improve the text editor's readability, its particular syntactical elements are visualized in different colors. The thesis also implements the process of ATB-DCK compilation into the PDDL planning task specification. Finally, the thesis specifies ATB-DCK in three characteristically unique domains and evaluates its effectivness for one specific NP-hard task scheduling problem.

Keywords: domain-independent automated planning, PDDL, Domain Control Knowledge, ATB-DCK

Supervisor: doc. RNDr. Lukáš Chrpa, Ph.D.
Czech Institute of Informatics, Robotics and Cybernetics (CIIRC),
Jugoslávských partyzánů 1580/3,
Prague 6

Abstrakt

V doménově nezávislém automatickém plánování je specifikace plánovací úlohy oddělena od její implementace, kterou poskytují doménově nezávislé plánovače. Tento přístup nabízí vysoký stupeň flexibility, protože plánovače jsou tak schopny řešit jakýkoli výpočetní problém, který lze převést na úlohu automatického plánování. Nevýhodou této přístupu však je, že specifikace plánovacích úloh často postrádají důležité kontextové informace vycházející z unikátních vlastností příslušných domén, což může negativně ovlivnit výkonnost doménově nezávislých plánovačů ve srovnání s výkonností doménově specifických algoritmů navržených pro stejné problémy. Alespoň částečného překlenutí výkonnostní propasti mezi plánovači a algoritmy by mohlo být dosaženo zkonstruováním dodatečných doménově specifických informací známých jako Domain Control Knowledge (DCK), které by mohly být zakódovány přímo do specifikací plánovacích úloh.

ATB-DCK (Attributed Transition-Based Domain Control Knowledge) je typ DCK, který je reprezentován konečným stavovým automatem. Jeho stavy a přechody ukládají dodatečná omezení na použitelnost akcí definovaných ve specifikacích plánovacích úloh, a tím omezují počet přípustných akcí, které může plánováč v každém kroku plánovacího procesu provést. Hlavní výhodou ATB-DCK je jeho jednoduchost a čitelnost i pro netechnické uživatele.

Cílem této práce je zjednodušit proces kompilace ATB-DCK do specifikace plánovací úlohy reprezentované popisným jazykem PDDL. Práce nejprve formuluje jazyk pro reprezentaci ATB-DCK v datovém formátu JSON, který lze snadno zpracovávat počítačovými programy. Poté představuje grafický editor pro vytváření a vizualizaci ATB-DCK jako konečného stavového automatu, který nabízí intuitivní způsob konstrukce datové reprezentace ATB-DCK. Součástí grafického editoru je textový editor, který umožňuje upravovat ATB-DCK přímo v jeho JSON datové podobě. Pro zlepšení čitelnosti textového editoru jsou jeho jednotlivé syntaktické prvky vizualizovány různými barvami. V práci je rovněž implementován samotný proces kompilace ATB-DCK do PDDL specifikace plánovací úlohy. Nakonec práce specifikuje ATB-DCK ve třech charakteristicky unikátních doménách a vyhodnocuje jeho efektivitu pro jeden konkrétní NP-obtížný problém rozvrhování úloh.

Klíčová slova: doménově nezávislé automatické plánování, PDDL, Domain Control Knowledge, ATB-DCK

Contents

1 Introduction 1

2 Attributed Transition-Based Domain Control Knowledge (ATB-DCK) 5

2.0.1 Planning Task Definition 5

2.0.2 ATB-DCK Definition 9

2.0.3 Planning Task With ATB-DCK 14

3 ATB-DCK Data Representation 21

3.0.1 JSON Data Format 21

3.0.2 DCK Memory Predicates . . . 22

3.0.3 ATB-DCK Variables 23

3.0.4 ATB-DCK States 24

3.0.5 ATB-DCK Transitions 24

4 ATB-DCK Graphical Editor 27

4.0.1 PyQt 27

4.0.2 User Interface 27

4.0.3 Communication Between UI Components 34

4.0.4 Scene Geometry 35

5 Compilation Of ATB-DCK Into PDDL Planning Task 41

5.0.1 Tarski 42

5.0.2 Data Representation Of Inference Rules 42

5.0.3 Compilation Process 44

6 Experimental evaluation 47

6.0.1 Childsnack 47

6.0.2 Reconfigurable Machines 51

6.0.3 Performance Evaluation 56

7 Conclusion 59

A Bibliography 61

Figures

2.1 PDDL planning domain model constructed for the Blocksworld domain. 8

2.2 PDDL planning problem constructed for the Blocksworld domain. 9

2.3 ATB-DCK for the Blocksworld domain. 10

3.1 Example of the JSON data format. 22

3.2 JSON representation of DCK memory predicates defined for the Blocksworld domain. 23

3.3 JSON representation of ATB-DCK variables defined for the Blocksworld domain. 23

3.4 JSON representation of ATB-DCK states defined for the Blocksworld domain. 24

3.5 JSON representation of the ATB-DCK transition defined for the Blocksworld domain. 25

4.1 Main window of the ATB-DCK graphical editor. 28

4.2 Toolbar of the ATB-DCK graphical editor. 28

4.3 Scene of the ATB-DCK graphical editor. 29

4.4 Scene hierarchy of the ATB-DCK graphical editor. 30

4.5 Inspector of the ATB-DCK graphical editor when nothing is selected. 30

4.6 Inspector of the ATB-DCK graphical editor when a state is selected. 31

4.7 Inspector of the ATB-DCK graphical editor when a transition is selected. 31

4.8 Error popup message - invalid input. 32

4.9 UI component "Variables" of the ATB-DCK graphical editor. 33

4.10 JSON text editor of the ATB-DCK graphical editor. 34

4.11 Scene coordinate system. 36

4.12 Geometry of a state. 36

4.13 Geometry of a line transition. . 37

4.14 Geometry of a curved transition. 38

4.15 Geometry of a self-transition. . . 39

5.1 ASP data representation of inference rules defined for the Blocksworld domain. 43

5.2 ATB-DCK-enhanced PDDL planning domain model constructed for the Blocksworld domain. 44

5.3 ATB-DCK-enhanced PDDL planning problem constructed for the Blocksworld domain. 45

5.4 Mapping of ATB-DCK-enhanced planning operators to original ones defined for the Blocksworld domain. 46

5.5 Plan extraction from the ATB-DCK-enhanced solution generated for the planning task of the Blocksworld domain. 46

6.1 ATB-DCK defined for the Childsnack domain. 48

6.2 DCK memory predicates defined for the Childsnack domain. 49

6.3 ATB-DCK-enhanced planning operator defined for the Childsnack domain. 50

6.4 Inference rules defined for the Childsnack domain. 51

6.5 ATB-DCK-enhanced PDDL planning problem defined for the Childsnack domain. 51

6.6 Planning operator defined for the Reconfigurable Machines domain. . 53

6.7 Planning problem defined for the Reconfigurable Machines domain. . 54

6.8 ATB-DCK defined for the Reconfigurable Machines domain. . 55

Tables

6.1 Test results from solving original and ATB-DCK-enhanced instances of the Reconfigurable Machines problem. 56

ctuthesis t1606152353

Introduction

Domain-independent automated planning is a branch of artificial intelligence that is concerned with finding a sequence of actions that takes a given domain from a defined initial state to a goal state, i.e., a state or set of states that fulfill the defined goal criteria. In order to solve a problem using automated planning, it must first be defined as an automated planning task and then provided to a domain-independent automated planning engine that can solve it [1].

While automated planning engines for solving automated planning tasks grow in popularity, their performance still pales in comparison to algorithmic methods specifically designed to solve the same types of problems. However, while algorithmic methods are often restricted to addressing only specific subsets of conceptually similar problems, domain-independent automated planning engines are capable of solving any problem that can be defined as an automated planning task. The flexibility of automated planning engines has given rise to ongoing research efforts aimed at further improving their efficiency and effectiveness in solving various computational problems.

One of the key reasons behind the inefficiency of automated planning engines is the lack of context information they get from planning task specifications. Unlike algorithmic methods, automated planning engines are often restricted to working with a limited amount of context information, making it difficult for them to develop effective problem-solving strategies. As a result, automated planning researchers are actively exploring ways to provide domain-independent automated planning engines with additional context information to help them solve problems more efficiently and to further bridge

the performance gap between them and algorithms.

A common approach to providing additional context information to automated planning engines is to directly encode it into the planning task specification. This additional context information can be called Domain Control Knowledge (DCK). The aim of many researchers is to make the DCK planner-independent, so it may be exploited by any standard domain-independent automated planning engine. DCK can take many forms, but this thesis focuses on one specific action-centric planner-independent DCK called Attributed Transition-Based Domain Control Knowledge (ATB-DCK), recently introduced by Chrpa and Barták [2] who were inspired by their previous work of Transition-based DCK [3]. Action-centric DCKs provide more information about defined actions and the relationships between them. Typically, the aim of action-centric DCK is to limit the number of permissible actions the automated planning engine can take at each step of the planning process by modifying actions' definitions (i.e., planning operators).

ATB-DCK can be represented as a finite-state automaton. Each state of the automaton is represented by an attribute that holds additional context information about specific objects substituted for the attribute's arguments, e.g., an attribute $dck_state(?x)$ tells the automated planning engine that the object $?x$ lies in the state dck_state. Each automaton transition is represented by a source and a destination ATB-DCK state, a planning operator (that is typically taken from the original planning domain model), and sets of constraints and modifiers representing the additional preconditions on the transition's applicability and its additional effects, respectively. Apart from the ATB-DCK states themselves, ATB-DCK defines an additional set of unique predicates known as DCK memory. Similarly to ATB-DCK states, DCK memory predicates hold additional context information about specific objects substituted for their arguments and can be added to ATB-DCK transitions' constraints and modifiers. ATB-DCK can be encoded into the planning task specification by adding newly defined predicates (DCK memory predicates and predicates encoded from ATB-DCK states) to the planning domain model, by replacing the original operators with operators encoded from ATB-DCK transitions, and by modifying the initial state of the planning problem. The main advantage of ATB-DCK is its simplicity and readability (doesn't have to be always true as some domains might produce quite complex ATB-DCKs) even for non-technical users.

The primary objective of this thesis is to simplify the process of encoding ATB-DCK into the automated planning task specification represented in the PDDL description language [4]. The objective can be divided into four sub-objectives:

1. To design a language for ATB-DCK data representation.
2. To design and implement a graphical editor for editing and visualizing ATB-DCK in the form of a finite-state automaton.
3. To design and implement a text editor for ATB-DCK data representation as a part of the graphical editor.
4. To implement the compilation of ATB-DCK data representation into the PDDL planning task specification.

The secondary objective of this thesis is to specify ATB-DCK in several domains and compare the performance of automated planning engines in solving ATB-DCK-enhanced and original planning tasks.

Chapter 2

Attributed Transition-Based Domain Control Knowledge (ATB-DCK)

In order to correctly design a language for ATB-DCK data representation and to implement the encoding of ATB-DCK into the automated planning task specification, it is first necessary to properly define both the planning task and ATB-DCK. All definitions found in this chapter are directly taken from the already mentioned paper that introduced ATB-DCK [2].

2.0.1 Planning Task Definition

Below is the definition of the classical planning task. While ATB-DCK should be theoretically usable even in non-classical automated planning (e.g., temporal planning), current research is primarily focused on its use in classical planning.

Definition 2.1 (Planning task). *A **planning task** is a pair* $\Pi = (Dom_\Pi, Prob_\Pi)$ *where a **planning domain model*** $Dom_\Pi = (P_\Pi, O_\Pi)$ *is a pair consisting of a finite set of predicates* P_Π *and planning operators* O_Π*, and a **planning problem*** $Prob_\Pi = (Obj_\Pi, I_\Pi, G_\Pi)$ *is a triple consisting of a finite set of objects* Obj_Π*, initial planning state* I_Π *and goal* G_Π*.*

Let ats_Π *be the set of all **atoms** that are formed from the predicates* P_Π *by substituting the objects* Obj_Π *for the predicates' arguments. In other words, an atom is an **instance** of a predicate (in the rest of the paper when we use the term instance, we mean an instance that is fully grounded, i.e., all the variables are substituted by objects). A **planning state** is a subset of* ats_Π*,*

*and the **initial planning state** I_Π is a distinguished state. The **goal** G_Π is a non-empty subset of ats_Π, and a **goal planning state** is any state that contains the goal G_Π.*

Each predicate of the planning domain model consists of a unique name and a list of predicate arguments. Together, they define all possible states of objects (and all possible relationships between objects) that can be substituted for their arguments. In some cases, the list of predicate arguments might be empty to represent a state of the corresponding domain that is not tied to any objects (e.g., the *handempty*() predicate from the well-known Blocksworld domain defines the state where the robotic hand manipulating blocks doesn't hold any block at the moment). In those special cases, both the predicate and its instance (atom) have identical forms.

Each planning state fully describes the overall state of the domain. Only atoms that are present in the given planning state are considered to be true in the domain's overall state, all other atoms are assumed to be false.

Planning Operator

As was already mentioned, ATB-DCK is an action-centric DCK which means it is primarily focused on actions and the relationships between them. The action's applicability in each step of the planning process can be controlled by modifying its definition. The action's definition is called the planning operator.

Definition 2.2 (Planning operator). *A **planning operator** $o = (name(o), pre^+(o), pre^-(o), eff^-(o), eff^+(o))$ is specified such that $name(o) = op_name(x_1, ..., x_k)$, where op_name is a unique identifier and $x_1, ..., x_k$ are all the variable symbols (arguments) appearing in the operator, $pre^+(o)$ and $pre^-(o)$ are sets of predicates representing an operator's positive and negative preconditions, respectively, $eff^-(o)$ and $eff^+(o)$ are sets of predicates representing an operator's negative and positive effects, respectively. **Actions** are instances of planning operators that are formed by substituting objects, which are defined in a planning problem, for operators' arguments as well as for corresponding variable symbols in operators' preconditions and effects. An action $a = (pre^+(a), pre^-(a), eff^-(a), eff^+(a))$ is **applicable** in a planning state s if and only if $pre^+(a) \subseteq s$ and $pre^-(a) \cap s = \emptyset$. Application of a in s, if possible, results in a planning state $(s \setminus eff^-(a)) \cup eff^+(a)$.*

Similarly to the predicate, the planning operator consists of a unique name and a list of operator arguments that can be substituted by concrete objects

defined in the planning problem or by constants defined in the planning domain model. In addition, it consists of operator preconditions and effects, each represented by a set of predicates or predicate negations. Negated predicates represent negative preconditions or negative effects. When a planning operator is instantiated and turned into the action, all its preconditions and effects are instantiated (by substituted operator arguments) as well. Therefore, preconditions and effects of action consist of atoms (or atom negations) instead of predicates.

Actions play a crucial role in the planning process as they allow automated planning engines to alter the current planning state in an attempt to reach a defined goal. For an action to be applicable in the current step of the planning process, the current planning state must contain all the action's positive preconditions and lack all its negative ones. When the action is applied, the current planning state is stripped of all the action's negative effects (if they are present in the current planning state) and extended with all its positive ones.

Planning Task Solution

Definition 2.3 (Planning task solution). *A sequence of actions is a **solution**, or a **solution plan** of a planning task* Π *if and only if a consecutive application of the actions from the plan starting in the initial planning state of* Π *results in the goal planning state of* Π*.*

Most planning tasks have more than one solution or even an infinite number of solutions. For example, in the well-known Blocksworld domain, if the robotic hand, which is manipulating the blocks, grabs one specific block and then puts it back in the same place, then the hand performed two actions (grabbing the block and putting it down), but the current planning state is exactly the same as it was before these two actions were performed. In other words, one specific planning state can be reached by different sequences of actions which means that the goal planning state can be reached by different solution plans. Naturally, the aim of automated planning is to find the most optimal solution.

Similarly, most planning tasks have more than one possible goal planning state. Again, let's use the example from the Blocksworld domain. The goal of the planning task might be that all blocks defined in the planning problem need to be stacked in a single tower. In this case, each unique configuration of blocks forming the single tower satisfies the given goal, thus leading to

different goal planning states in one planning task.

PDDL Planning Task Specification

This thesis works with automated planning tasks that have their specifications defined in the PDDL description language [4]. PDDL planning task specifications are divided into two main parts, a planning domain model and a planning problem. The planning domain model defines object types, predicates, and planning operators (actions) used in planning tasks corresponding to the given domain. Sometimes, the planning domain model may even define new constants. A constant is a concrete object which is present in all planning tasks constructed for the represented domain. Figure 2.1 shows the truncated version of the PDDL planning domain model defined for the well-known Blocksworld domain.

```
(define (domain blocksworld)

    (:types
        block
    )

    (:predicates
        (handempty)
        (on ?x - block ?y - block)
        (ontable ?x - block)
        (clear ?x - block)
        (holding ?x - block)
    )

    (:action pick_up
        :parameters (?x - block)
        :precondition (and (clear ?x) (ontable ?x) (handempty))
        :effect (and (not (ontable ?x)) (not (clear ?x)) (not (handempty)) (holding ?x))
    )

    ; rest omitted

)
```

Figure 2.1: PDDL planning domain model constructed for the Blocksworld domain.

Sets of preconditions and effects of a planning operator are specified as conjunctions of predicates represented by *and* connectors followed by sequences of predicates. The keyword *not* represents a negation of the predicate that comes after it.

The planning problem defines an initial planning state, a goal, and concrete objects of a specific planning task. Both the initial planning state and the goal are formed from atoms that are created from predicates defined in the planning domain model by substituting their arguments with objects defined in the planning problem. An example of the PDDL planning problem defined

for the Blocksworld domain is shown in Figure 2.2.

```
(define (problem blocksworld_instance)

    (:domain blocksworld)

    (:objects
        b1 b2 b3 - block
    )

    (:init
        (clear b1)
        (clear b3)
        (on b1 b2)
        (ontable b3)
        (ontable b2)
        (handempty)
    )

    (:goal
        (and (on b1 b2) (on b2 b3) (clear b1))
    )

)
```

Figure 2.2: PDDL planning problem constructed for the Blocksworld domain.

The goal of the classical planning task is specified as a conjunction of atoms represented by the *and* connector followed by a sequence of predicates.

2.0.2 ATB-DCK Definition

This section formally defines ATB-DCK and all its components. Below is the definition of ATB-DCK as a whole.

Definition 2.4 (ATB-DCK). *An* ***Attributed Transition-based Domain Control Knowledge (ATB-DCK)*** *is a tuple* $\mathcal{K} = (Dom_{\Pi}, S, M, T)$*, where*

- $Dom_{\Pi} = (P_{\Pi}, O_{\Pi})$ *is a planning domain model, where* P_{Π} *is a set of predicates and* O_{Π} *is a set of planning operators.*
- S *is a set of attributed states denoted as* ***ATB-DCK states****.*
- M *is a set of predicates distinct from* P_{Π} *representing a* ***DCK memory****.*

- *T is a set of transitions between ATB-DCK states denoted as **ATB-DCK transitions**.*

For its full representation, ATB-DCK draws data (defined object types, planning operators and predicates) from the planning domain model of the original planning task. ATB-DCK also defines its own predicates known as DCK memory predicates to represent additional context information about specific objects substituted for the DCK predicates' arguments (e.g. information about open goals of substituted objects or achieved milestones of the planning process). As previously mentioned, ATB-DCK can be represented as a finite-state automaton consisting of ATB-DCK states and transitions. Both the ATB-DCK state and the ATB-DCK transition are properly defined later in this section. Figure 2.3 shows the example of ATB-DCK created for the well-known Blocksworld domain.

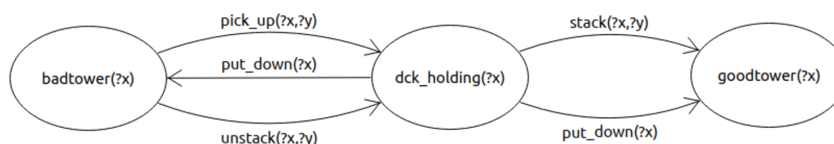

Figure 2.3: ATB-DCK for the Blocksworld domain.

ATB-DCK is domain-specific, so the same ATB-DCK can be encoded into different planning tasks that share the same planning domain model.

ATB-DCK State

ATB-DCK state is a core component of ATB-DCK. It defines additional context information about specific objects substituted for its arguments, e.g., the ATB-DCK state $goodtower(?x)$ from the popular Blocksworld domain informs the automated planning engine that the block $?x$ is exactly where it is supposed to be (according to the defined goal of the corresponding planning task) and thus, it doesn't have to be further manipulated. The formal definition of the ATB-DCK state is provided below.

Definition 2.5 (ATB-DCK state). *An **ATB-DCK state** $s = s_id(attr_1, ..., attr_k)$ is specified via a unique state identifier s_id and a list of variable symbols (attributes) $attr_1, ..., attr_k$.*

When encoding ATB-DCK into the planning task specification, each ATB-DCK state is encoded into the predicate and added to the planning domain model as a new predicate. Therefore, the ATB-DCK state identifier must be unique even among the identifiers (names) of predicates originally defined in the domain model of the given planning task. The same applies to DCK memory predicates.

ATB-DCK Transition

ATB-DCK transitions represent new planning operators replacing the original operators in the ATB-DCK-enhanced planning task specification. The formal definition of the ATB-DCK transition is provided below.

Definition 2.6 (ATB-DCK transition). *An **ATB-DCK transition** is a tuple $t = (s, o, \mathcal{C}, \mathcal{M}, s')$, where $var(s), var(s') \in S$, $o \in O_\Pi \cup \{\perp\}$, $\mathcal{C}$ is a set of constraints, where each constraint takes one of the forms:*

- *p such that $var(p) \in P_\Pi \cup M$*
- *$\neg p$ such that $var(p) \in P_\Pi \cup M$*
- *st:q such that $var(q) \in S$*

and $\mathcal{M}$ is a set of modifiers, where each modifier takes one of the forms:

- *$+p$ such that $var(p) \in M$*
- *$-p$ such that $var(p) \in M$*

Predicates used in the ATB-DCK transition have their arguments substituted with identifiers of the transition's arguments, but they might be defined using different arguments' identifiers. To take this into account, the term *variant* of a predicate was defined. A predicate q is a *variant* of a predicate p, denoted as $q = var(p)$, if q is formed from p by renaming its arguments while preserving their order according to the arguments' object types.

The transition's source and destination ATB-DCK states are denoted as s and s', respectively. The planning operator assigned to the ATB-DCK

transition and denoted as o can be either an existing planning operator taken from the domain model of the original planning task or the so-called *empty* operator (denoted as $\bot$), which is the operator with no defined arguments, preconditions, or effects. The set of the transition's constraints $\mathcal{C}$ represents additional preconditions on the transition's applicability. Constraints are represented by predicates (or predicate negations in case of negative constraints) either from the planning domain model of the given planning task or from the DCK memory. They could be even represented by variants of ATB-DCK states distinct from s and s'. Similarly to transition's constraints, the set of transition's modifiers $\mathcal{M}$ represents additional effects of the given transition. Modifiers can be represented by DCK memory predicates or by their negations (in case of negative effects).

The same planning operator (may it be an operator from the planning domain model of the given planning task or the *empty* operator) can be assigned to multiple ATB-DCK transitions. In other words, the ATB-DCK-enhanced planning domain model might contain several modified versions of one planning operator. Also, there is no limitation on number of transitions between two ATB-DCK states (or even one state, because self-transitions with both endpoints being the same state are permitted), so multiple transitions with exactly the same source and destination states may occur.

ATB-DCK Configuration

Just like the atoms created from predicates defined in the planning domain model form a planning state of the given planning task, the atoms created from ATB-DCK states and DCK memory predicates form an ATB-DCK configuration.

Definition 2.7 (ATB-DCK configuration). *Let $\Pi = (Dom_\Pi, Prob_\Pi)$ be a planning task, $\mathcal{K} = (Dom_\Pi, S, M, T)$ be an Attributed Transition-based DCK, and $Obj^{\mathcal{K}}$ be a set of problem-specific DCK memory objects. A **configuration** $\mathcal{K}^c$ of $\mathcal{K}$ is a pair $\mathcal{K}^c = (S^c, M^c)$ such that $S^c \subseteq \{s^c \mid s^c$ an instance of $s \in S\}$ and $M^c \subseteq \{m^c \mid m^c$ an instance of $m \in M\}$. Instantiation of both ATB-DCK states and ATB-DCK memory predicates is done by substituting objects (defined in $Prob_\Pi$ and $Obj^{\mathcal{K}}$) for all corresponding variable symbols.*

ATB-DCK may use additional object types distinct from the object types defined in the original planning task. These new object types can be used in arguments of newly defined predicates (i.e., ATB-DCK states and DCK memory predicates) or to create new problem-specific DCK memory objects.

ATB-DCK Transition Applicability

Similarly to planning operators, ATB-DCK transitions have preconditions on their applicability and effects that can modify both the current planning state and the current ATB-DCK configuration.

Definition 2.8 (ATB-DCK transition applicability). *Let $\Pi = (Dom_\Pi, Prob_\Pi)$ be a planning task, $\mathcal{K} = (Dom_\Pi, S, M, T)$ be an Attributed Transition-based DCK, $Obj^\mathcal{K}$ be a set of problem-specific DCK memory objects, s_Π be a current planning state and $\mathcal{K}^s = (S^s, M^s)$ be a current configuration of $\mathcal{K}$. We say a transition $t \in T$, where $t = (s_t, o_t, \mathcal{C}_t, \mathcal{M}_t, s'_t)$, is **applicable** in s_Π and K^s with respect to Θ, a substitution from variable symbols appearing in the elements of t into objects (defined in $Prob_\Pi$ and $Obj^\mathcal{K}$), if and only if:*

- $\Theta(s_t) \in S^s$
- *$\Theta(o_t)$ is applicable in s_Π ($\perp$ is always applicable)*
- *for each $p \in \mathcal{C}_t$ it is the case that $\Theta(p) \in s_\Pi \cup M^s$*
- *for each $\neg p \in \mathcal{C}_t$ it is the case that $\Theta(p) \notin s_\Pi \cup M^s$*
- *for each st:$q \in \mathcal{C}_t$ it is the case that $\Theta(q) \in S^s$*

*The **result** of a transition application (if possible) is a new planning state s'_Π and a new configuration $\mathcal{K}^{s'} = (S^{s'}, M^{s'})$. In particular:*

- $s'_\Pi = (s_\Pi \setminus \Theta(eff^-(o_t))) \cup \Theta(eff^+(o_t))$ *(for $o_t = \perp$, $s'_\Pi = s_\Pi$)*
- $S^{s'} = (S^s \setminus \{\Theta(s_t)\}) \cup \{\Theta(s'_t)\}$
- $M^{s'} = (M^s \setminus \{\Theta(p) \mid -p \in \mathcal{M}_t\}) \cup \{\Theta(p) \mid +p \in \mathcal{M}_t\}$

*We also say that a planning state s'_Π and a configuration $\mathcal{K}^{s'}$ are **reachable** from a planning state s_Π and a configuration $\mathcal{K}^s$ if and only if there exists a sequence of transitions (from T) whose consecutive application in $s_\Pi, \mathcal{K}^s$ (must always be possible) results in $s'_\Pi, \mathcal{K}^{s'}$.*

For the instantiated ATB-DCK transition to be applicable in the current step of the planning process, the transition's source ATB-DCK state must be part of the current ATB-DCK configuration, the instantiated planning

operator assigned to the transition must be applicable in the current planning state, the transition's positive constraints must be present either in the current planning state or in the current ATB-DCK configuration, and the transition's negative constraints must not be present in the current planning state nor in the current ATB-DCK configuration.

After the instantiated ATB-DCK transition is applied, the transition's source ATB-DCK state is replaced in the current ATB-DCK configuration by the transition's destination ATB-DCK state, the instantiated planning operator assigned to the transition is applied in the current planning state, and the current ATB-DCK configuration is stripped of all transition's negative modifiers and extended with all the positive ones.

2.0.3 Planning Task With ATB-DCK

This section defines the ATB-DCK-enhanced planning task and all its components. Below is the definition of the planning task with ATB-DCK as a whole.

Definition 2.9 (Planning Task With ATB-DCK). *A* ***planning task with ATB-DCK*** *is a tuple* $\Pi^{\mathcal{K}} = (Dom_\Pi, Prob_\Pi, \mathcal{K}, \mathcal{K}^I)$ *where* Dom_Π *is a planning domain model,* $Prob_\Pi$ *is a planning problem,* $\mathcal{K} = (Dom_\Pi, S, M, T)$ *is an ATB-DCK and* $\mathcal{K}^I$ *is an initial configuration of* $\mathcal{K}$*.*

Both the planning task and ATB-DCK have their initial configuration, which fully describes their initial state over the concrete objects at the start of the automated planning process. The already established and defined initial configuration of the planning task is called the initial planning state, which is represented by atoms formed from predicates defined in the domain model of the planning task. Similarly, the initial configuration of ATB-DCK is represented by atoms formed from ATB-DCK states and DCK memory predicates.

Derivation Of Initial ATB-DCK Configuration

The paper that introduced ATB-DCK [2] presents the method for deriving the initial configuration of ATB-DCK. Specific atoms forming the ATB-DCK initial configuration are derived from the so-called **inference rules**. Inference

rules are represented by Horn clauses in their implicative form as follows:

$$R_p(a_{k_1}, ..., a_{k_n}) \Leftarrow F_p(a_1, ..., a_m)$$

The head of inference rules $R_p(a_{k_1}, ..., a_{k_n})$ represents either a DCK memory predicate or an ATB-DCK state, e.g., $goodtower(?x)$. The body of inference rules $F_p(a_1, ..., a_m)$ is a conjunction of literals that can be specified in the following forms:

- initial state query: I:$p(a_1, ..., a_n)$, e.g., I:$on(?x, ?y)$
- goal query: G:$p(a_1, ..., a_n)$, e.g., G:$on(?x, ?y)$
- initial DCK memory or ATB-DCK state query: $p(a_1, ..., a_n)$, e.g., $goodtower(?x)$
- cardinality query: $count(c, b_i, F(b_1, ..., b_q))$, $(1 \leq i \leq q)$, e.g., $count(?c, ?p,$I:$waiting(?ch, ?p))$ determines how many children are initially waiting for a sandwich at the fixed place $?p$, the desired number is here denoted as $?c$

Initial state and goal queries test the existence of corresponding atoms in the initial state or goal of the given planning task, respectively. Initial DCK memory or ATB-DCK state queries test whether the corresponding atoms can be derived from the defined inference rules. Finally, cardinality queries are used to determine the number of satisfied formulas (literals or inference rules) for each instance of their fixed argument b_i. In classical automated planning, numbers (integers) are typically represented by special objects additionally defined for each used number. Arithmetic operations over the integer objects can be performed by additionally defined special predicates.

Attributes of inference rule heads, as well as the literals present in the inference rule bodies, are substituted with concrete objects that are either defined in the original planning problem or newly defined by ATB-DCK. If the instantiated inference rule is evaluated as true (i.e., all literals present in its body are evaluated as true), the rule is deemed satisfied and its head is encoded into a new DCK memory atom (using the head's substituted arguments). All DCK memory atoms encoded from satisfied inference rules form together the initial ATB-DCK configuration.

If the DCK memory atom can be derived from multiple formulas (inference rule bodies), i.e., the inference rule bodies can be formed into a single body represented in a disjunctive normal form (DNF), then the inference rule can

be split into multiple inference rules sharing the same head while having different bodies. Below is an example of split inference rules for the derivation of the $goodtower(?x)$ ATB-DCK state defined for the Blocksworld domain:

$$goodtower(?x) \Leftarrow \text{I:}ontable(?x) \wedge \text{G:}ontable(?x)$$
$$goodtower(?x) \Leftarrow \text{I:}ontable(?x) \wedge freeBot(?x)$$
$$goodtower(?x) \Leftarrow \text{I:}on(?x, ?y) \wedge \text{G:}on(?x, ?y) \wedge goodtower(?y)$$
$$goodtower(?x) \Leftarrow \text{I:}on(?x, ?y) \wedge goodtower(?y) \wedge freeBot(?x) \wedge freeTop(?y)$$

Not all inference rules need to be encoded into atoms and added to the initial configuration of ATB-DCK. Some inference rules, e.g., $freeBot(?x)$ and $freeTop(?x)$, exist only to simplify bodies of other inference rules as their possible atoms are not used in any ATB-DCK-enhanced action, so adding them to the initial ATB-DCK configuration would be redundant.

The initial configuration of ATB-DCK is denoted as $K^I = (S^I, M^I)$ where S^I is a set of initial atoms formed from ATB-DCK states and M^I is a set of initial atoms formed from DCK memory predicates.

ATB-DCK-Enhanced Planning Task Specification

In order to exploit ATB-DCK by domain-independent automated planning engines, it must be first encoded into the planning task specification. The paper that introduced ATB-DCK [2] presents the algorithm 1 for encoding ATB-DCK into the planning domain model of the given planning task.

To summarize the algorithm, the original predicates of the planning task are extended with predicates defined by ATB-DCK (i.e., DCK memory predicates and predicates encoded from ATB-DCK states). Additionally, all original operators are replaced with ones encoded from ATB-DCK transitions. The algorithm also extends the original object types with object types defined by ATB-DCK as they are possibly used in arguments of new predicates and operators. The extension of object types is not specifically shown in the algorithm, but it is a crucial step of the ATB-DCK encoding process. The *encPred* function used in the algorithm encodes an ATB-DCK state into a uniquely identifiable predicate. ATB-DCK transitions are encoded into new planning operators by extending preconditions and effects of associated operators with transitions' defined constraints and modifiers, respectively. In addition, a source state of the transition is added to the associated operator's preconditions, while a destination state is added to its effects. Arguments of

Algorithm 1 Encoding ATB-DCK into domain model

Require: $Dom_\Pi = (P_\Pi, O_\Pi)$, $\mathcal{K} = (Dom_\Pi, S, M, T)$
Ensure: $Dom_\Pi^\mathcal{K} = (P_\Pi^\mathcal{K}, O_\Pi^\mathcal{K})$
1: **procedure** ENCODE_ATB-DCK(P_Π, O_Π, S, M, T)
2: $S' \leftarrow \{encPred(s) \mid s \in S\}$
3: $P_\Pi^\mathcal{K} \leftarrow P_\Pi \cup M \cup S'$
4: $O_\Pi^\mathcal{K} \leftarrow \emptyset$
5: **for all** $t = (s, o, \mathcal{C}, \mathcal{M}, s') \in T$ **do**
6: $\mathcal{C}1 \leftarrow \{p \mid p \in \mathcal{C}\}$
7: $\mathcal{C}2 \leftarrow \{p \mid \neg p \in \mathcal{C}\}$
8: $\mathcal{C}3 \leftarrow \{encPred(q) \mid st{:}q \in \mathcal{C}\}$
9: $\mathcal{M}1 \leftarrow \{p \mid +p \in \mathcal{M}\}$
10: $\mathcal{M}2 \leftarrow \{p \mid -p \in \mathcal{M}\}$
11: $pre^+(o^t) \leftarrow pre^+(o) \cup \{encPred(s)\} \cup \mathcal{C}1 \cup \mathcal{C}3$
12: $pre^-(o^t) \leftarrow pre^-(o) \cup \mathcal{C}2$
13: $eff^+(o^t) \leftarrow eff^+(o) \cup \{encPred(s')\} \cup \mathcal{M}1$
14: $eff^-(o^t) \leftarrow eff^-(o) \cup \{encPred(s)\} \cup \mathcal{M}2$
15: $args(o^t) \leftarrow collectArgs(o^t)$
16: $O_\Pi^\mathcal{K} \leftarrow O_\Pi^\mathcal{K} \cup \{o^t\}$
17: **end for**
18: $Dom_\Pi^\mathcal{K} \leftarrow (P_\Pi^\mathcal{K}, O_\Pi^\mathcal{K})$
19: **end procedure**

newly formed operators are collected from arguments of all predicates used in preconditions and effects of given operators, this step is represented by the *collectArgs* function. A result of the algorithm is an ATB-DCK-enhanced planning domain model, denoted as $Dom_\Pi^\mathcal{K} = (P_\Pi^\mathcal{K}, O_\Pi^\mathcal{K})$, where $P_\Pi^\mathcal{K}$ is a set of extended predicates and $O_\Pi^\mathcal{K}$ is a set of newly created operators from ATB-DCK transitions.

The compilation of ATB-DCK into the original planning problem, denoted as $Prob_\Pi = (Obj_\Pi, I_\Pi, G_\Pi)$, where Obj_Π is a set of original problem-specific objects, I_Π is an original initial planning state, G_Π is a defined goal, and $K^I = (S^I, M^I)$ is an initial ATB-DCK configuration which may use newly defined ATB-DCK objects denoted as $Obj^\mathcal{K}$, is performed as follows:

$$Prob_\Pi^\mathcal{K} = (Obj_\Pi \cup Obj^\mathcal{K}, I_\Pi \cup M^I \cup \{encPred(s) \mid s \in S^I\}, G_\Pi)$$

In other words, original problem-specific objects are extended with additional objects defined by ATB-DCK, and the original initial planning state is extended with atoms from the initial ATB-DCK configuration. The goal remains the same. $Prob_\Pi^\mathcal{K}$ represents an ATB-DCK-enhanced planning problem.

Because the ATB-DCK-enhanced planning task, denoted as $\Pi' = (Dom_{\Pi}^{\mathcal{K}}, Prob_{\Pi}^{\mathcal{K}})$, shares the same goal with the corresponding original planning task, it can be provided to domain-independent automated planning engines to hopefully reach the given goal in a more efficient manner. It is worth noting that ATB-DCK doesn't guarantee an improvement in the efficiency of solving automated planning tasks as it may perform poorly for some specific domains or specific planning engines.

Solution Of Planning Task With ATB-DCK

After solving the ATB-DCK-enhanced planning task, the automated planning engine generates the ATB-DCK-enhanced solution plan represented by the sequence of actions encoded from ATB-DCK transitions. The solution of the original planning task, represented by the sequence of original actions, can be extracted from the corresponding ATB-DCK-enhanced solution plan. Below is the definition of the solution of a planning task with ATB-DCK.

Definition 2.10 (Solution Of Planning Task With ATB-DCK). *Let $\Pi^{\mathcal{K}} = (Dom_{\Pi}, Prob_{\Pi}, \mathcal{K}, \mathcal{K}^I)$ be an ATB-DCK planning task, I_{Π} be the initial state of $\Pi^{\mathcal{K}}$ and T be the set of transitions defined in $\mathcal{K}$. We say that a sequence of grounded transitions $\langle t_1, ..., t_n \rangle$ (from T) **solves** $\Pi^{\mathcal{K}}$ if and only if its consecutive application starting in I_{Π} and $\mathcal{K}^I$ results in a planning state satisfying the goal defined in $Prob_{\Pi}$.*

Grounded ATB-DCK transitions are associated with actions from the original planning task (or with *empty* actions). The ATB-DCK transition is applicable only if the associated action is applicable (an *empty* action is always applicable) and if all its additional constraints (including the transition's source ATB-DCK state) are satisfied. The additional constraints can significantly limit the number of applicable actions in each step of the planning process and thus make the whole planning process much more effective. Furthermore, the ATB-DCK transition can alter the current planning state with the effects of its associated action and with its additional modifiers (including the transition's destination ATB-DCK state). However, the transition's additional modifiers are formed only from predicates that are not part of the original planning task, so the transition's alteration of the current planning state doesn't affect the applicability of original actions. Therefore, the sequence of actions, representing the solution of the original planning task, can be obtained from the solution of the ATB-DCK-enhanced planning task by simply extracting the original actions associated with grounded transitions (encoded into ATB-DCK-enhanced actions) forming the ATB-DCK-enhanced solution. *Empty* actions are omitted from the final solution of the original

planning task as they are not defined in the original planning domain model and as their application has no effect on the domain environment.

Chapter 3

ATB-DCK Data Representation

The aim of this chapter is to design a language for ATB-DCK data representation, so it can be used by computer programs. Specifically, this chapter includes data representations of DCK memory predicates, ATB-DCK variables, ATB-DCK states, and ATB-DCK transitions. Data representations of inference rules and the planning task specification are described in the later chapter about the compilation of ATB-DCK into the PDDL planning task.

3.0.1 JSON Data Format

To simplify the process of parsing ATB-DCK data representation and transforming it into data structures of chosen programming language, it was decided to use the popular JSON (JavaScript Object Notation) text format [5]. Apart from its simplicity and readability even for non-technical users, the main advantage of the JSON data format is its broad support across many programming languages, which typically possess optimized built-in libraries for its handling.

The JSON data format can be used to represent JSON objects (map data structures consisting of key-value pairs), arrays (list of values with allowed duplicates), strings, integer or floating point numbers, boolean values (true/false), and null values representing unknown or undefined values of any data types. Both the JSON object and the JSON array may contain values of any supported data types, including other objects and arrays, which allows the possible representation of complex hierarchical data structures.

Figure 3.1 shows the example of JSON representation. All data must be encapsulated into a single *root* JSON object and thus enclosed by a pair of curly braces. The keys of a JSON object are represented by unique (in the context of a given object) strings and followed by colons separating keys from their corresponding values. Items of JSON arrays are enclosed in square brackets, and similarly to key-value pairs of JSON objects, they must be separated by commas.

```
{
  "object": {
    "string_key": "string_value",
    "number_int_key": 100,
    "number_float_key": 100.0,
    "boolean_key": true,
    "null_key": null,
    "object_key": {},
    "array_key": ["string_value", 100, 100.0, true, null, {}, []]
  }
}
```

Figure 3.1: Example of the JSON data format.

3.0.2 DCK Memory Predicates

DCK memory predicates represent additional predicates defined by ATB-DCK and used in planning operators of the ATB-DCK-enhanced planning task. Predicates consist of a unique identifier (name) and a list of predicate arguments. In JSON representation, DCK memory predicates can be represented by key-value pairs where keys are formed from predicates' identifiers (names), and values are specified by arrays of object types corresponding to predicates' arguments.

Figure 3.2 shows the JSON representation of DCK memory predicates defined for the popular Blockworld domain. The $gon(?x-block, ?y-block)$ predicate provides the planning process with additional information on whether the given planning task's goal specifies if the block $?x$ must be stacked on the block $?y$. The $mStacked(?x-block)$ predicate indicates whether the goal specifies if the block $?x$ has to be stacked on any other block at all.

It is possible to represent (define) DCK memory predicates separately from the actual ATB-DCK data representation, where they are only referenced using their identifiers.

```
{
    "gon": ["block", "block"],
    "mStacked": ["block"]
}
```

Figure 3.2: JSON representation of DCK memory predicates defined for the Blocksworld domain.

3.0.3 ATB-DCK Variables

ATB-DCK variables are used to correctly map arguments of ATB-DCK states, ATB-DCK transitions, and transitions' additional constraints and modifiers to corresponding objects. In JSON representation, ATB-DCK variables can be represented as key-value pairs where keys are represented by their unique identifiers (symbols), and values are specified by identifiers of their object types.

Figure 3.3 shows the JSON representation of ATB-DCK variables defined for the Blockworld domain. To define ATB-DCK for the Blocksworld domain, only two unique *block* variables are needed.

```
{
    "vars": {
        "x": "block",
        "y": "block"
    }
}
```

Figure 3.3: JSON representation of ATB-DCK variables defined for the Blocksworld domain.

3.0.4 ATB-DCK States

ATB-DCK states are encoded into predicates and added to preconditions and effects of planning operators formed from ATB-DCK transitions. Therefore, similarly to DCK memory predicates, ATB-DCK states can be represented by key-value pairs where keys are represented by unique predicate identifiers (names), and values are specified by arrays of predicate arguments already substituted with ATB-DCK variables.

Figure 3.4 shows the JSON representation of ATB-DCK states defined for the Blockworld domain. The $goodtower(?x)$ and the $badtower(?x)$ states indicate whether the placement of the block x satisfies the goal of the corresponding planning task, and thus, whether the block $?x$ should be further manipulated. The $dck_holding(?x)$ state is an intermediate state between the $badtower(?x)$ and the $goodtower(?x)$ states in which the block $?x$ is currently held by the robotic hand manipulating objects. The goal of the given ATB-DCK-enhanced planning task is satisfied only if all blocks defined in the original planning problem lie in the ATB-DCK state $goodtower(?x - block)$. Only one of ATB-DCK states can be *active* (i.e., part of the current planning state) for the specific block $?x$ at the same time.

```json
{
    "S": {
        "goodtower": ["x"],
        "badtower": ["x"],
        "dck_holding": ["x"]
    }
}
```

Figure 3.4: JSON representation of ATB-DCK states defined for the Blocksworld domain.

3.0.5 ATB-DCK Transitions

ATB-DCK transitions consist of their source and destination ATB-DCK states, associated planning operators, and sets of transitions' additional constraints and modifiers. In the data representation of ATB-DCK transitions, already

defined transitions' source and destination states can be referenced by their unique identifiers. Associated operators may also be specified by their unique identifiers, with an empty string referring to an *empty* planning operator, and by arrays of operators' arguments substituted with ATB-DCK variables. Similarly to associated operators, transitions' additional constraints and modifiers, stored in JSON arrays, can be represented by unique identifiers of corresponding predicates and by arrays of their arguments substituted with ATB-DCK variables. Identifiers of constraints' and modifiers' corresponding predicates might be prefixed by a tilde character ( ) representing the negation of given predicates. All ATB-DCK transitions are stored in a JSON array.

Figure 3.5 shows the JSON representation of the specific ATB-DCK transition defined for the Blockworld domain. The transition between the *dck_holding* and the *goodtower* ATB-DCK states is associated with the *put_down* planning operator having the ATB-DCK variable $?x$ as its only argument. It has only one additional constraint and no modifiers. The additional constraint represents the negation of the DCK memory predicate *mStacked* with the $?x$ ATB-DCK variable as its argument.

```
{
    "T": [
        {
            "src": "dck_holding",
            "dest": "goodtower",
            "operator": {
                "name": "put_down",
                "params": ["x"]
            },
            "C": [
                {
                    "pred": "~mStacked",
                    "attrs": ["x"]
                }
            ],
            "M": []
        }
    ]
}
```

Figure 3.5: JSON representation of the ATB-DCK transition defined for the Blocksworld domain.

The transition specifies that the block $?x$ can be moved from the *dck_holding* state (a state in which the block is held by a robotic hand) to the *goodtower* state (a state in which the block's placement satisfies the

planning task's defined goal) by applying the associated operator *put_down* only if the planning task's goal doesn't specify whether the block $?x$ must be stacked on any other block.

Chapter 4

ATB-DCK Graphical Editor

This chapter describes the design and implementation of a graphical editor for editing and visualizing ATB-DCK, both in the form of a finite-state automaton and its JSON data representation.

4.0.1 PyQt

The graphical editor was developed using PyQt [6]. PyQt is a popular Python binding for the cross-platform C++ framework called Qt. The Qt framework is used for creating multiplatform graphical user interfaces, and PyQt provides Python API (Application Programming Interface) to allow the usage of the Qt framework through Python programming language [7].

4.0.2 User Interface

This section describes all main UI (User Interface) components of the ATB-DCK graphical editor. Figure 4.1 shows the main window of the graphical editor with almost all described UI components being visible (except for the JSON text editor). Most UI components of the ATB-DCK graphical editor can be resized by grabbing and dragging their borders with a mouse.

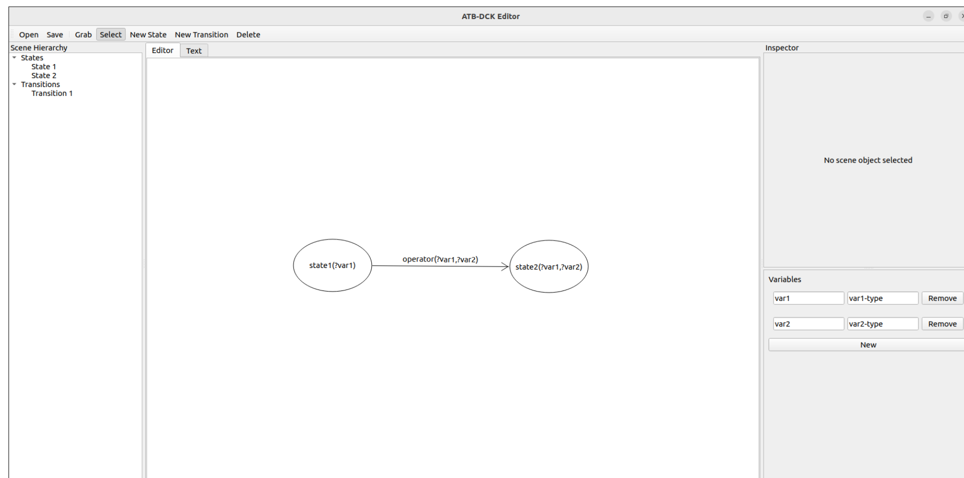

Figure 4.1: Main window of the ATB-DCK graphical editor.

Toolbar Options

The toolbar of the ATB-DCK graphical editor, shown in Figure 4.2, offers several actions. The "Open" action allows the user to import the existing JSON representation of ATB-DCK into the graphical editor and visualize the imported ATB-DCK as a finite-state automaton. The "Save" action allows the user to save the JSON representation of ATB-DCK extracted from the editor's data to a chosen location. The rest of the toolbar actions are used to interact with a scene and are further specified during the description of the UI component representing the scene.

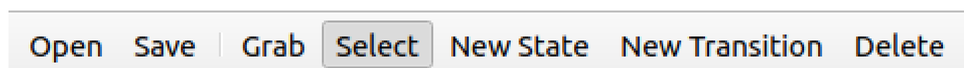

Figure 4.2: Toolbar of the ATB-DCK graphical editor.

Scene

The scene is the main UI component of the ATB-DCK graphical editor. Its purpose is to design and visualize the representation of ATB-DCK as a finite-state automaton. The scene UI component is shown in Figure 4.3.

The user can interact with the scene through modes (actions) defined in the graphical editor's toolbar. The "Grab" mode allows the user to move the viewport of the scene (visible part of the scene), so the user can reach the whole scene no matter the size of the viewport. The "Select" mode can be used to select scene objects to edit their properties or to move them across

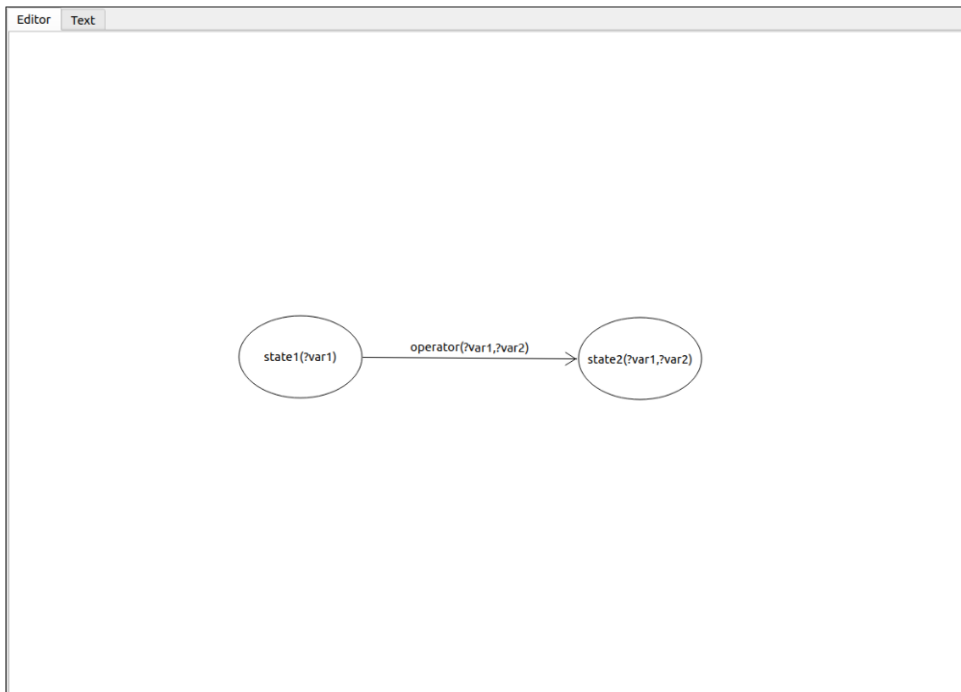

Figure 4.3: Scene of the ATB-DCK graphical editor.

the scene. The "New State" mode allows the user to create a new state on a position of the scene where the user clicked with his mouse. The "New Transition" mode can be used to create a new transition between two states (or one state in case of self-transitions) the user clicked on. The direction of a new transition is determined by the order in which the user clicked on two states. Finally, the "Delete" mode allows the user to remove scene objects from the scene by clicking on them.

The scene can be toggled into the JSON text editor, which is described later in this section.

Scene Hierarchy

The scene hierarchy, illustrated in Figure 4.4, displays all states and transitions currently present in the scene in a tree-like structure. Its main purpose is to allow the user to select scene objects from the simple menu without having to find them in the scene.

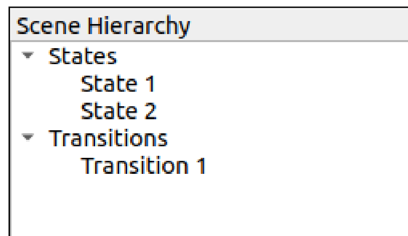

Figure 4.4: Scene hierarchy of the ATB-DCK graphical editor.

Inspector

The inspector displays editable properties of selected scene objects. Its layout is changed according to the current selection. Its default view, shown in Figure 4.5, is displayed when nothing is selected.

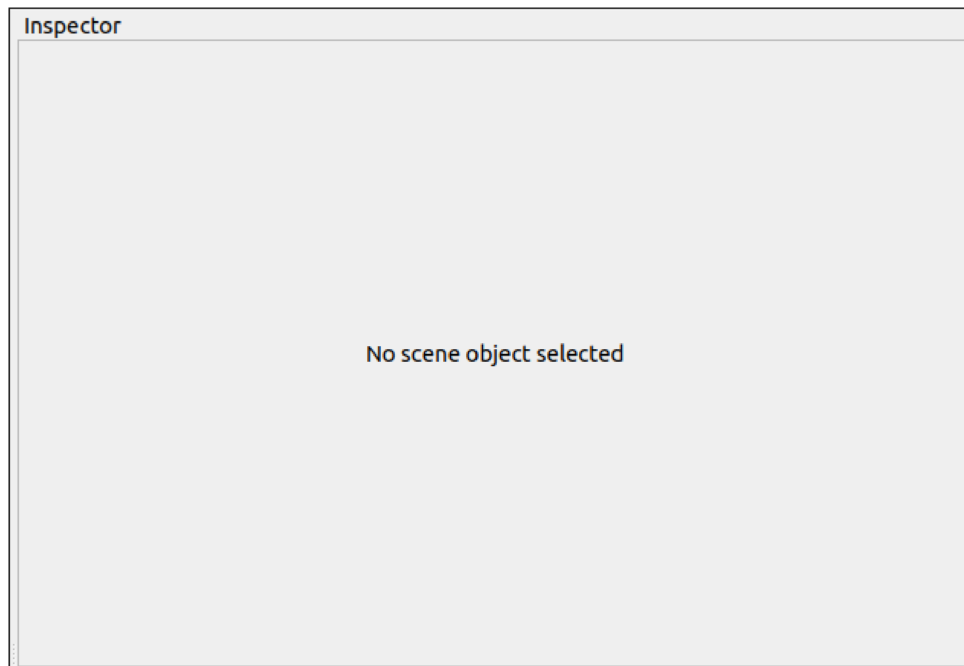

Figure 4.5: Inspector of the ATB-DCK graphical editor when nothing is selected.

Figure 4.6 shows the appearance of the inspector when the state is selected. The state properties consist of the state name, which is used to represent the state in the scene hierarchy, and the state attribute, which represents the predicate associated with the given ATB-DCK state. The state attribute consists of a unique identifier (name) and a list of its arguments represented by identifiers of ATB-DCK variables defined in the "Variables" UI component, which is described later in this section.

Figure 4.7 shows the appearance of the inspector when the transition is

Figure 4.6: Inspector of the ATB-DCK graphical editor when a state is selected.

selected. Similarly to states, the transition has the name property representing the given transition in the scene hierarchy. Additionally, the transition properties include the property of the associated operator and properties corresponding to the ATB-DCK transition's constraints and modifiers. The transition's associated operator consists of a unique identifier and a list of ATB-DCK variables representing its arguments. The transition's constraints and modifiers are specified by identifiers of their corresponding predicates, by lists of ATB-DCK variables representing arguments of their corresponding predicates, and by boolean values indicating whether the corresponding predicates are negated or not.

Figure 4.7: Inspector of the ATB-DCK graphical editor when a transition is selected.

All text fields are validated towards corresponding regular expressions.

E.g., the regular expression for text fields representing either predicate or operator identifiers is structured as follows: `^[a-zA-Z]+([-]*\w+)*$`. This regular expression allows non-empty values that contain only alphanumeric characters, hyphens (-), or underscores (_). Moreover, values matching this regular expression must start with a letter from the alphabet and can't end with a hyphen. The use of such regular expressions makes all values defined in the ATB-DCK graphical editor syntactically compatible with PDDL description language [4] that is used to represent the planning task specification. Text fields representing arguments of either predicates or operators accept sequences of ATB-DCK variable identifiers separated by commas as their values. In this case, apart from the validation of text field values with the corresponding regular expression, each variable identifier must correspond to the existing ATB-DCK variable defined in the "Variables" UI component.

The validation process for the specific text field is performed every time the text field is edited, and the UI widget corresponding to the given text field loses focus (i.e., the user clicks away from the widget). If the validation is deemed unsuccessful, all changes to the text field are reverted to its last valid value, and the message informing the user about the invalid input pops up on the main screen. An example of such unsuccessful validation is shown in Figure 4.8.

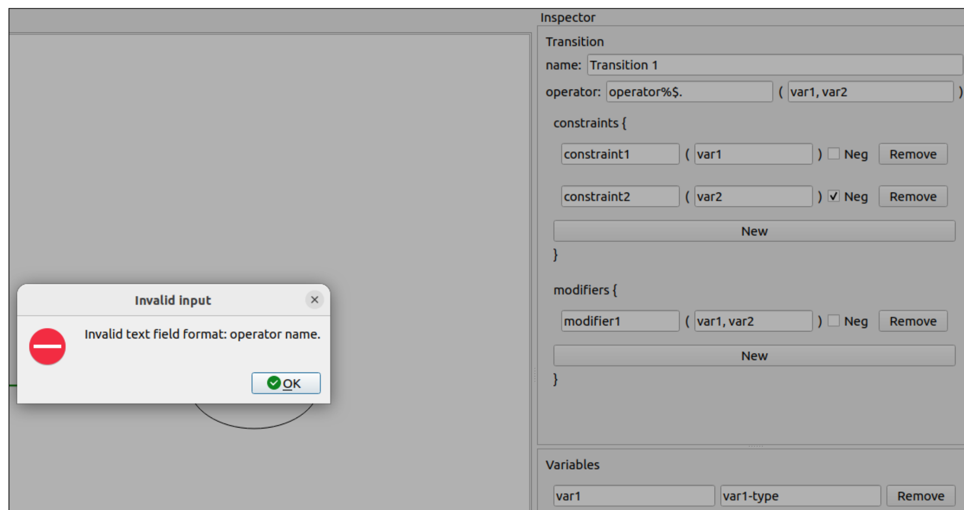

Figure 4.8: Error popup message - invalid input.

Variables

The "Variables" UI component, shown in 4.9, is where ATB-DCK variables are defined. ATB-DCK variables consist of unique identifiers and object types.

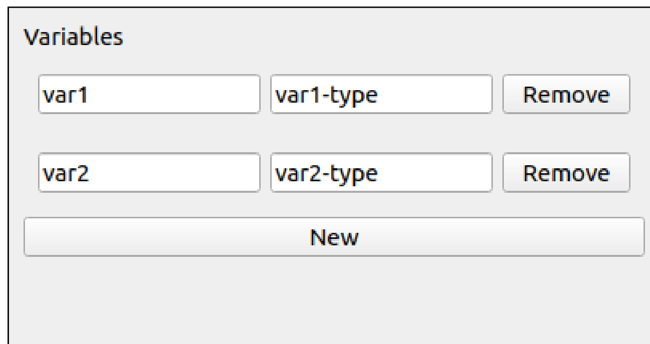

Figure 4.9: UI component "Variables" of the ATB-DCK graphical editor.

All text fields corresponding to properties of ATB-DCK variables use the same validation process already specified in the description of the inspector UI component. To ensure the uniqueness of variable identifiers, the validation process can be extended with an additional validation step that checks whether the ATB-DCK variable with the newly defined identifier already exists.

JSON Text Editor

The JSON text editor displays the ATB-DCK data representation. The user can access the text editor via the tab menu located on the widget containing the scene UI component. When the text editor is accessed and visible, all other UI components described in this section (except the toolbar) have their visibility toggled off (i.e., they become hidden). Because the scene itself is no longer visible, all toolbar options interacting with it are also hidden. The UI component of the JSON text editor is shown in Figure 4.10.

The text editor displays an enhanced version of ATB-DCK. This enhanced version includes the ATB-DCK graphical editor's data, which are not required during the compilation of ATB-DCK into the PDDL planning task but are used to correctly reconstruct ATB-DCK as a finite-state automaton inside of the graphical editor. For example, these data may include positions (x and y coordinates) of individual states in the scene or names of states and transitions representing corresponding objects in the scene hierarchy. The ATB-DCK graphical editor exports the enhanced version of ATB-DCK data representation but is able to import data and recreate finite-state automaton even from unenhanced versions. The recreation of the finite-state automaton from the unenhanced version of ATB-DCK data representation places all states at the same position in the middle of the scene and lets the user rearrange them himself. To easily include or exclude the graphical editor's data from the actual ATB-DCK data representation, they are specified separately under

Figure 4.10: JSON text editor of the ATB-DCK graphical editor.

a new key ("editor_data") of the JSON root object instead of being injected directly into data representations of ATB-DCK states and transitions.

The visual distinction of JSON syntactical elements is achieved by using different colors for different groups of elements. Elements are sorted into such groups according to regular expressions they match. E.g. there might be different syntactical groups for curly braces, square brackets, strings, and numbers.

4.0.3 Communication Between UI Components

In the implementation, the root of the ATB-DCK graphical editor is its main window which displays and manages all UI components. It also serves as the mediator in communication between particular UI components. Such communication occurs when changes in one UI component affect other components. For example, if the user selects a scene object, the scene hierarchy should highlight the menu item corresponding to the given object, and the inspector should display the selected object properties. Simply put, the edited UI component informs the main window about the changes that occurred, the main window redirects this information to affected UI components, and said components then perform relevant actions.

For communication between individual UI components (widgets), the Qt framework introduces so-called signals and slots [8]. Signals are emitted in reaction to particular events, e.g., the selection of a scene object or the editing of a text field, and can be intercepted by connected functions called slots. The Qt framework offers built-in signals for many events but also allows the creation of custom ones. For communication between main UI components, the ATB-DCK graphical editor uses a combination of built-in and custom signals, which are intercepted by slots defined in the main window and connected to corresponding signals. E.g., when a scene object is selected, the corresponding signal is emitted and intercepted by the connected slot defined in the main window. This connected slot receives a reference (from the emitted signal) to the selected object and passes it to both the scene hierarchy and the inspector UI components. The scene hierarchy highlights the menu item corresponding to the selected object (represented by its reference), and the inspector displays its properties.

4.0.4 Scene Geometry

This section describes the geometry used to represent ATB-DCK as a finite-state automaton.

Scene Coordinate System

The Qt framework encloses all widgets, including the scene and all the scene items, with so-called bounding rectangles. The bounding rectangle determines the dimensions of an enclosed widget. The origin of its local coordinate system lies in its top-left corner. From there, the x coordinate increases towards the right, and the y coordinate increases downward, so the x-axis of the bounding rectangle's local coordinate system can be represented by a line connecting its top-left corner with the top-right corner, and the y-axis by a line connecting its top-left corner with the bottom-left corner [9]. However, it's worth noting that the widget's size and position are ultimately managed by the Qt layout system, which can adjust them independently of the bounding rectangle. As a result, the axes of the widget's local coordinate system may not always align with the axes of the corresponding bounding rectangle's local coordinate system [10]. Let's assume that for the scene of the ATB-DCK graphical editor, they align. The scene coordinate system, which is basically the two-dimensional Cartesian coordinate system with the x-axis oriented to the right and the y-axis oriented downward, is shown in Figure 4.11.

Figure 4.11: Scene coordinate system.

State Geometry

States of the finite-state automaton are represented by ellipses. Same as the scene itself, each ATB-DCK state is enclosed with its own bounding rectangle. The geometry of a state is illustrated in Figure 4.12.

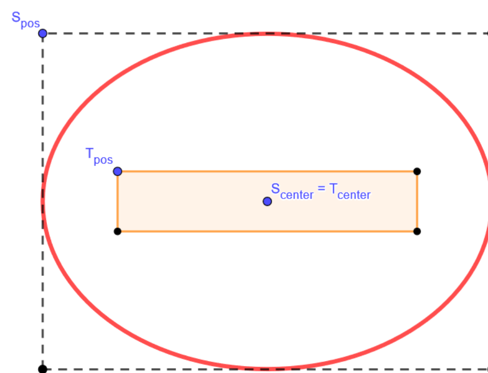

Figure 4.12: Geometry of a state.

The state's position in the scene is determined by the position of its local coordinate system's origin, denoted as S_{pos}. In the middle of the ATB-DCK state lies the text formed from the state's associated attribute. The text position, denoted as T_{pos}, can be computed by subtracting half the text's width from the x coordinate of the state center, denoted as S_{center}, and half the text's height from its y coordinate. The text position is recomputed every time the text is changed or the corresponding state is moved. If the text doesn't fit the ellipse representing the ATB-DCK state, it is displayed shortened and suffixed with three dots to indicate a truncated text.

Transition Geometry

In the implementation, individual transitions of the finite-state automaton are assigned with their shape. This shape determines how they are drawn between their source and destination states. The main advantage of this approach is that all ATB-DCK transitions can share a single implementation while being drawn differently according to their assigned shape, which can be dynamically swapped during runtime.

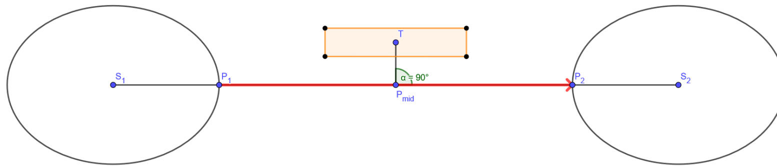

Figure 4.13: Geometry of a line transition.

The line shape is represented by a directed line segment connecting two ATB-DCK states. Figure 4.13 visualizes the construction of a line transition. The line segment connecting the centers of two ATB-DCK states (denoted as S_1 and S_2) has exactly one intersection point with each state. These two intersection points (denotes as P_1 and P_2) are the endpoints of a newly created line transition. To visualize the direction of the given transition, an arrowhead is drawn at its computed destination endpoint P_2. The text formed from the transition's associated operator is placed above the center of the transition line denoted as P_{mid}. The text center T lies on a perpendicular line passing through the point P_{mid} at the chosen distance from the transition line. Positions and rotations of both the transition and the text are constantly updated whenever corresponding states are moved. The text can also be manually repositioned if the user is not fully satisfied with its placement.

Line transitions are sufficient only if a finite-state automaton can be represented as a simple directed graph (i.e., a graph with at most one edge between the same pair of vertices and no self-loops on a single vertex). However, ATB-DCK supports multiple transitions (edges) between the same pair of states (vertices) as well as multiple self-transitions (self-loops) on a single state (vertex). Therefore, to avoid overlapping transitions between two states, a curved transition shape illustrated in Figure 4.14 is introduced.

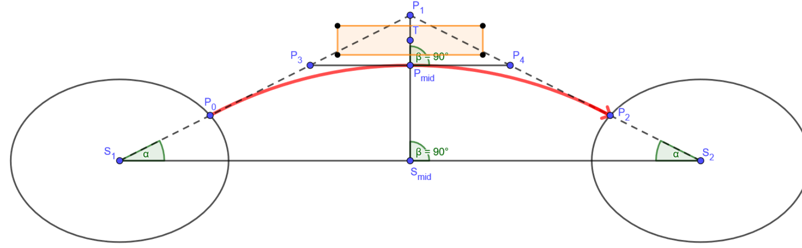

Figure 4.14: Geometry of a curved transition.

To construct a curved transition, an isosceles triangle (i.e., a triangle with at least two equally long sides), defined by the centers of two ATB-DCK states (denoted as S_1 and S_2) and some point P_1, is created. The dimensions of the triangle are determined by the size of the angle α representing the triangle's two internal angles located at points S_1 and S_2. The size of the α angle can be arbitrarily chosen. An intersection point of the transition's source state and the triangle's side between vertices S_1 and P_1 is denoted as P_0 and represents the start of the curved transition. The end of the curved transition, denoted as P_2, is represented by an intersection point of the transition's destination state and the triangle's side between vertices S_2 and P_1. The curved transition is then constructed as a quadratic Bézier curve defined by points P_0, P_1, and P_2, with an arrowhead representing the curve direction drawn at its destination point P_2. Such quadratic Bézier curve can be defined by the following function:

$$B(t) = (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2,\ 0 \le t \le 1$$

The function B returns a point on the curve according to its parameter t. The center of the Beziér curve, denoted as P_{mid}, equals $B(0.5)$. The text center T can lie anywhere on the line passing through points P_1 and P_{mid}. The described construction of a curved transition allows two more non-overlapping transitions between the same pair of ATB-DCK states. It means that the finite-state automaton representing ATB-DCK in the ATB-DCK graphical editor can be constructed with at most three transitions (one line transition and two curved transitions) between each pair of different states. In the future development, to allow the representation of more than three different

transitions between the same ATB-DCK states, a "super transition" visually specified by a single transition but holding properties of several ATB-DCK transitions could be introduced.

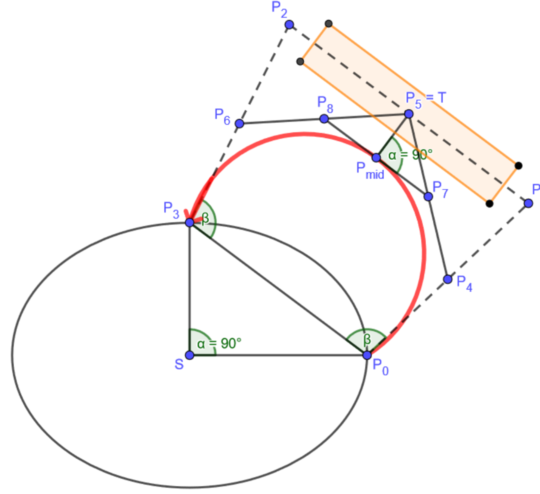

Figure 4.15: Geometry of a self-transition.

The self-transition shape is used by self-transitions specified on a single ATB-DCK state. Its construction is illustrated in Figure 4.15. The self-transition's source and destination endpoints can be found by creating a right triangle defined by points S, P_0, and P_3. Line segments between points S and P_0 and points S and P_3 represent the semi-major and semi-minor axes of the ellipse representing the corresponding state, respectively. Point S is the center of the ellipse, point P_0 defines the start of the self-transition, and point P_3 represents its end. The self-transition is constructed as a cubic Bézier curve (with an arrowhead drawn at its destination point P_3) which allows rounder shapes than its quadratic counterpart. So far, only two of the four points that define the cubic Bézier curve have been specified. Rest can be found by creating an isosceles trapezoid with the line segment between points P_0 and P_3 as one of its bases (parallel sides of the trapezoid). An isosceles trapezoid is any trapezoid with equally long legs (non-parallel sides of the trapezoid). The dimensions of the constructed trapezoid are determined by the size of the angle β representing the trapezoid's two internal angles located at points P_0 and P_3 and by the length of the trapezoid's legs. Both these values can be arbitrarily chosen. A cubic Bézier curve defined by all vertices of the trapezoid denoted as P_0, P_1, P_2, and P_3 can be represented by the following function:

$$B(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t) t^2 P_2 + t^3 P_2,\ 0 \le t \le 1$$

As with the quadratic Bézier curve, the center of the cubic Bézier curve, denoted as P_{mid}, is equal to $B(0.5)$. The text center T can lie anywhere on the line passing through the point P_{mid} and the center of the trapezoid's base between its vertices P_1 and P_2. Self-transitions could be theoretically visualized in each "corner" of the ATB-DCK state, so one state could potentially hold four different self-transitions. For better readability of the finite-state automaton, the number of allowed self-transitions on a single ATB-DCK state is limited to three. The already mentioned concept of "super transitions" could be used for the specification of more than four self-transitions.

Chapter 5

Compilation Of ATB-DCK Into PDDL Planning Task

This chapter describes the implementation of the compilation process for encoding ATB-DCK into the PDDL planning task. The compilation process accepts the following components as input:

- planning task specification represented in the PDDL description language
- ATB-DCK data representation specified in JSON data format
- list of DCK memory predicates specified in JSON data format
- list of inference rules for the derivation of new initial atoms enriching the initial planning state of the planning task

The aim of the compilation process is to merge all these components into an ATB-DCK-enhanced planning task specification represented in the PDDL description language and extract a solution plan for the originally defined planning task. To achieve this, all components of the compilation process must first be data-represented so they can be parsed by computer programs and transformed into data structures of the corresponding programming language. During the compilation process, those data structures are merged and modified into a desired form which is then transformed into a PDDL data representation of the ATB-DCK-enhanced planning task specification.

5.0.1 Tarski

Tarski [11] is a Python library designed to represent and manipulate automated planning tasks. Tarski even possesses a parser for a PDDL language which transforms the PDDL planning task into Tarski's internal data structures, which can be modified through the Python programming language [7]. The parser can't handle most features introduced in newer PDDL versions (e.g., numeric fluents) but works well for classical planning tasks. To encode ATB-DCK into a PDDL planning task, the compilation process can modify Tarski's data structures using data from Python representations of ATB-DCK, DCK memory predicates, and satisfied inference rules.

5.0.2 Data Representation Of Inference Rules

The remaining component of the compilation process whose data representation wasn't yet specified is the component that defines inference rules.

Answer Set Programming

Answer Set Programming (ASP) [12], semantically based on the logic programming [13], defines a set of parametrizable rules and returns only the satisfied ones as a solution (answer set). Moreover, simple ASP rules can be specified as Horn clauses in their implicative forms represented by their head and a body formed from a conjunction of literals. If both the representation of ASP rules and the ASP solution are compared to the definition and the purpose of inference rules, it could be observed that ASP perfectly aligns with the representation and the satisfiability determination of inference rules. Figure 5.1 shows the ASP data representation of inference rules defined for the Blocksworld domain.

Literals prefixed with "i_" correspond to atoms defined in the initial planning state of the original planning task, and literals prefixed with "g_" correspond to atoms defined in the planning task's goal. A satisfied rule head is turned into an atom and added to the initial planning task if it corresponds either to the ATB-DCK state (*goodtower*, *badtower*) or to the DCK memory predicate (*gon*, *mStacked*). Otherwise, it is ignored and serves only to simplify bodies of other inference rules. Inference rule bodies of $freeBot$ and $freeTop$ consist of two negative literals and one undefined literal *block*. The

```
goodtower(X) :- i_ontable(X), gontable(X).
goodtower(X) :- i_ontable(X), freeBot(X).
goodtower(X) :- i_on(X,Y), goodtower(Y), gon(X,Y).
goodtower(X) :- i_on(X,Y), goodtower(Y), freeBot(X), freeTop(Y).

badtower(X) :- i_ontable(X), gon(X,_).
badtower(X) :- i_on(X,Y), gontable(X).
badtower(X) :- i_on(X,Y), gon(Z,Y), Z != X.
badtower(X) :- i_on(X,Y), gon(X,Z), Y != Z.
badtower(X) :- i_on(X,Y), gclear(X).
badtower(X) :- i_on(X,Y), badtower(Y).

gontable(X) :- g_ontable(X).
gclear(X) :- g_clear(X).
gon(X,Y) :- g_on(X,Y).
mStacked(X) :- gon(X,_).
freeBot(X) :- not gontable(X), not gon(X,_), block(X).
freeTop(X) :- not gclear(X), not gon(_,X), block(X).
```

Figure 5.1: ASP data representation of inference rules defined for the Blocksworld domain.

block literal represents a fact (i.e., a rule with no body that always evaluates to true). Facts are created for each object defined in the original planning task from the object's type and identifier as follows: $object_type(object_id)$. All facts are added to the given ASP problem during the compilation process programmatically, so they are not part of the original ASP problem, not even if they are referenced from bodies of other inference rules. In this example, the presence of *block* literals is logically redundant but syntactically required, because all ASP rule bodies must be specified with at least one non-negative literal.

Clingo

Clingo is a solver for ASP written in C++ [14], which provides APIs for different programming languages, including Python, via a Python package called "clingo" [15] that provides bindings to the C++ library. The "clingo" Python API can be used to create and solve ASP problems using the Python programming language. The API can be further simplified by the "clorm" Python library [16] which provides developers with an intuitive Object Relational Mapping (ORM) style interface for communication with the Clingo solver. With the "clorm" library, developers can manage ASP problems using Python objects and methods.

5.0.3 Compilation Process

Once all components of the compilation process are transformed into data structures of chosen programming language (Python in this case), the original PDDL planning task specification can be programmatically modified into its ATB-DCK-enhanced version using data from other components.

```
(define (domain blocksworld)

    (:types
        block
    )

    (:predicates
        (handempty)
        (on ?x - block ?y - block)
        (ontable ?x - block)
        (clear ?x - block)
        (holding ?x - block)

        ; ATB-DCK states and DCK memory predicates:
        (gon ?x1 - block ?x2 - block)
        (mStacked ?x1 - block)
        (goodtower ?x1 - block)
        (badtower ?x1 - block)
        (dck_holding ?x1 - block)
    )

    (:action dck_pick_up
        :parameters (?x - block)
        :precondition (and
            ; original preconditions:
            (clear ?x) (ontable ?x) (handempty) (badtower ?x)
            ; new precoditions:
            (badtower ?x) (gon ?x ?y) (goodtower ?y) (clear ?y)
        )
        :effect (and
            ; original effects:
            (not (ontable ?x)) (not (clear ?x)) (not (handempty)) (holding ?x)
            ; new effects:
            (not (badtower ?x)) (dck_holding ?x)
        )
    )

    ; rest omitted

)
```

Figure 5.2: ATB-DCK-enhanced PDDL planning domain model constructed for the Blocksworld domain.

The compilation process starts with parsing the original PDDL planning task specification using the Tarski framework. Next, it parses data representations of ATB-DCK and DCK memory predicates, both specified in the JSON data format, using Python's built-in library for encoding and decoding the JSON texts [17]. The PDDL planning domain model is programmatically modified with data from ATB-DCK and DCK memory representations according to the algorithm 1, defined in the theoretical section 2.0.3 about the compilation of ATB-DCK into the planning task specification, for encoding ATB-DCK into the planning domain model of the corresponding planning

task. Figure 5.2 illustrates a truncated version of the ATB-DCK-enhanced PDDL planning domain model defined for the Blocksworld domain.

All objects and atoms defined in the planning problem are turned into facts and programmatically added to the representation of inference rules. Facts formed from atoms of the initial planning state are prefixed with "i_" while facts formed from atoms defined in the goal are prefixed with "g_". The corresponding ASP problem yields the sequence of satisfied rules (answer set) as a solution. All relevant satisfied rules are turned into atoms and added to the initial state of the original planning task. The planning problem might even be extended with some additional objects which can be yet undefined but present in arguments of some atoms formed from satisfied rules. An example of the ATB-DCK-enhanced PDDL planning problem defined for the Blocksworld domain is shown in Figure 5.3.

```
(define (problem blocksworld_instance)

    (:domain blocksworld)

    (:objects
        b1 b2 b3 - block
    )

    (:init
        ; original initial state atoms:
        (clear b1) (clear b3) (on b1 b2) (ontable b3) (ontable b2) (handempty)

        ; new initial state atoms formed from satisfied inference rules:
        (goodtower b3) (gon b2 b3) (gon b1 b2) (badtower b1) (badtower b2) (mStacked b1) (mStacked b2)
    )

    (:goal
        (and (on b1 b2) (on b2 b3) (clear b1))
    )

)
```

Figure 5.3: ATB-DCK-enhanced PDDL planning problem constructed for the Blocksworld domain.

Solution Plan Extraction

Finally, the fully modified PDDL planning task specification can be provided to chosen domain-independent automated planning engine that solves the ATB-DCK-enhanced planning task and generates its solution plan. The solution plan of the ATB-DCK-enhanced planning task is a sequence of actions that are enhanced either from original or empty actions. Extraction of the original planning task solution from the ATB-DCK-enhanced solution can be performed by mapping all enhanced actions to their original counterparts. Empty actions are naturally omitted from the final solution plan of the original planning task. A recipe for such mapping can be constructed during the process of encoding ATB-DCK into the planning domain model of the

original planning task. Figure 5.4 shows an example of mapping between planning operators defined for the Blocksworld domain.

```
b_h_unstack unstack 2
b_h_pick_up pick_up 1
h_b_put_down put_down 1
h_g_put_down put_down 1
h_g_stack stack 2
```

Figure 5.4: Mapping of ATB-DCK-enhanced planning operators to original ones defined for the Blocksworld domain.

Each row of the planning operators' mapping consists of three values. The first and the second value are identifiers of the ATB-DCK-enhanced planning operator and the corresponding original operator, respectively. The third value is the number of the original operator's arguments. This number determines how many arguments (taken from the beginning of the argument list) of the ATB-DCK-enhanced planning operator, which often has a higher number of arguments, need to be mapped to arguments of the corresponding original operator. It means that both the enhanced and the original planning operator must have the same order of arguments (apart from additional arguments of the enhanced operator appended to the end of the argument list) to be correctly mapped. An example of the solution plan extraction for the original planning task of the Blocksworld domain is shown in Figure 5.5.

```
(b_h_unstack b1 b2) -> (unstack b1 b2)
(h_b_put_down b1)   -> (put_down b1)
(b_h_pick_up b2 b3) -> (pick_up b2)
(h_g_stack b2 b3)   -> (stack b2 b3)
(b_h_pick_up b1 b2) -> (pick_up b1)
(h_g_stack b1 b2)   -> (stack b1 b2)
```

Figure 5.5: Plan extraction from the ATB-DCK-enhanced solution generated for the planning task of the Blocksworld domain.

Chapter 6

Experimental evaluation

The thesis has already established ATB-DCK for the Blocksworld domain and utilized it as a reference example. The aim of this chapter is to specify ATB-DCK for two more domains with unique characteristics and to evaluate the performance of solving ATB-DCK-enhanced planning tasks as compared to solving the original ones.

6.0.1 Childsnack

The Childsnack domain describes an environment in which sandwiches are made and served to children waiting on them. Some children might be allergic to gluten, so the sandwiches served to gluten-allergic children must be made from gluten-free ingredients. Sandwiches are made in the kitchen, put on a tray, and delivered to children waiting on them in groups at specific tables. There are no imposed constraints on the maximum capacity of trays, so all sandwiches, no matter their number, could be theoretically placed on a single tray. Therefore, the most efficient way of solving this task would be to first make all ordered sandwiches, then put them all on a single tray, and finally deliver them to all groups of children waiting at specific tables. The guidance for domain-independent planning engines toward the optimal solution could provide ATB-DCK defined for the Childsnack domain and shown in Figure 6.1.

The process of solving the Childsnack problem using this ATB-DCK first chooses a single tray (it makes defining more than one tray redun-

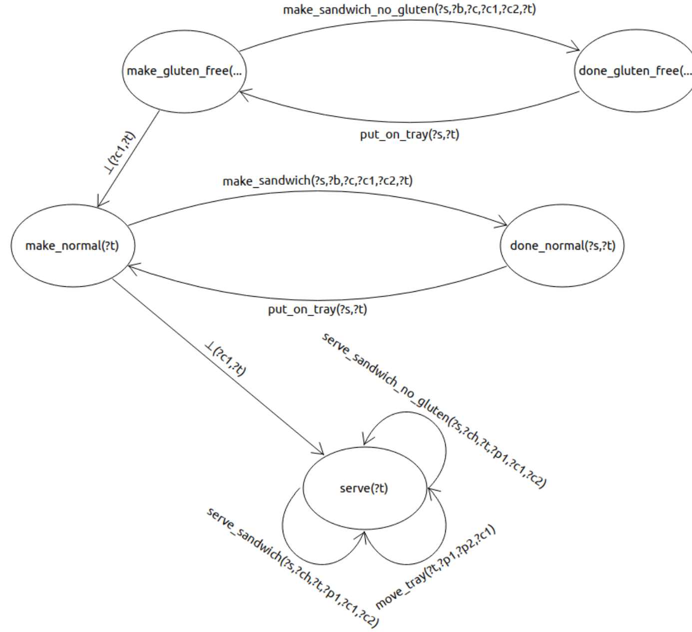

Figure 6.1: ATB-DCK defined for the Childsnack domain.

dant) that will be used to transport all prepared sandwiches to the waiting children. This tray is put into the $make_gluten_free(?t)$ ATB-DCK state where $?t$ corresponds to the substituted tray. In this state, the process checks the number of ordered and yet unprepared gluten-free sandwiches. If the number is higher than zero, then a new gluten-free sandwich is made, the number of unprepared sandwiches is decreased, and the tray is moved to the $done_gluten_free(?s, ?t)$ ATB-DCK state where $?s$ corresponds to a newly created sandwich. From there, the prepared sandwich can only be put on the tray, which moves the tray back to the $make_gluten_free(?t)$ state. When all ordered gluten-free sandwiches are finished (i.e., the number of unprepared gluten-free sandwiches is equal to zero) and put on the tray, then the tray is moved through an empty transition to the $make_normal(?t)$ ATB-DCK state. The process of preparing normal sandwiches is equivalent to the process of preparing gluten-free sandwiches. Instead of states $make_gluten_free(?t)$ and $done_gluten_free(?s, ?t)$, states $make_normal(?t)$ and $done_normal(?s, ?t)$ are used. After the preparation of all ordered normal sandwiches, the tray is moved through another empty transition into the final ATB-DCK state $serve(?t)$. In this state, the tray is being moved between tables, and all sandwiches from the tray are served among the waiting children. The process of solving the Childsnack problem ends once all children are served.

As is apparent from its description, ATB-DCK defined for the Childsnack domain works with numbers (e.g., a number of ordered gluten-free sandwiches) which influence the applicability of certain ATB-DCK-enhanced actions. The

lowest number is obviously zero and the highest number is the maximum from the number of ordered gluten-free sandwiches, the number of ordered normal sandwiches, and the number of waiting children at specific tables. The process of solving the ATB-DCK-enhanced Childsnack problem uses all integers between the lowest and the highest defined numbers as all numbers are continuously decreased until they are equal to zero (e.g., the number of unprepared gluten-free sandwiches is decreased by one every time a new gluten-free sandwich is made, and once it is equal to zero, the solving process is moved into the step of preparing normal sandwiches). Therefore, the only arithmetic operation used over these numbers is subtraction by one. Because the usage of numbers in the ATB-DCK-enhanced Childsnack problem is very simple, they can be handled in a classical planning fashion where each number is represented by a new object of a newly defined object type (e.g., *count* or *int*), and where all operations over integer objects are performed using newly defined predicates. DCK memory predicates defined for the Childsnack domain are shown in Figure 6.2.

```
{
    "mGlutenFreeS": ["count"],
    "mNormalS": ["count"],
    "mToServeS": ["place", "count"],
    "next": ["count", "count"],
    "min": ["count"]
}
```

Figure 6.2: DCK memory predicates defined for the Childsnack domain.

The predicates $mGlutenFreeS$ and $mNormalS$ specify numbers (represented by objects of the *count* object type) of ordered but yet unprepared gluten-free and normal sandwiches, respectively. The $mToServeS$ predicate specifies the number of children waiting to be served at the specific table substituted for the predicate's argument of the *place* object type. The *next* predicate specifies all pairs of adjacent integers (e.g., $next(0, 1)$, $next(1, 2)$, $next(2, 3)$, ...) from the lowest to the highest number used in the corresponding ATB-DCK-enhanced planning task. The *next* predicate is used for the arithmetic operation of subtraction by one. Finally, the *min* predicate specifies the lowest defined number, which is zero. The usage of these predicates can be observed in the ATB-DCK-enhanced planning operator shown in Figure 6.3.

The planning operator defines an action for making the gluten-free sandwich. It is constructed from the ATB-DCK transition between *make_gluten_free* and *done_gluten_free* ATB-DCK states. This transition is applicable only if the number of yet unprepared sandwiches (held by the $mGlutenFreeS$ predicate) is higher than zero. That is represented by two planning operator

```
(:action mgf_dgf_make_sandwich_no_gluten
    :parameters (
        ; original parameters:
        ?s - sandwich ?b - bread-portion ?c - content-portion
        ; new parameters:
        ?c1 - count ?c2 - count ?t - tray
    )
    :precondition (and
        ; original preconditions:
        (at_kitchen_bread ?b) (at_kitchen_content ?c) (no_gluten_bread ?b)
        (no_gluten_content ?c) (notexist ?s)
        ; new preconditions:
        (make_gluten_free ?t) (mGlutenFreeS ?c2) (not (min ?c2)) (next ?c1 ?c2)
    )
    :effect (and
        ; original effects:
        (not (at_kitchen_bread ?b)) (not (at_kitchen_content ?c))
        (at_kitchen_sandwich ?s) (no_gluten_sandwich ?s) (not (notexist ?s))
        ; new effects:
        (not (make_gluten_free ?t)) (done_gluten_free ?s ?t)
        (not (mGlutenFreeS ?c2)) (mGlutenFreeS ?c1)
    )
)
```

Figure 6.3: ATB-DCK-enhanced planning operator defined for the Childsnack domain.

preconditions specified by the predicate $mGlutenFreeS(?c2)$ and the negated predicate $min(?c2)$. If the *count* object substituted for the $?c2$ argument is held by the min atom, then the number of yet unprepared sandwiches is equal to zero, which means that in the current planning state, the ATB-DCK transition is not applicable. The precondition represented by the predicate $next(?c1, ?c2)$ specifies the *count* object substituted for the $?c1$ argument, which in the planning operator effects decreases the value held by the $mGlutenFreeS$ atom by one. The value of the $mGlutenFreeS$ atom is decreased by replacing the current $mGlutenFreeS$ atom holding the value $?c2$ with the $mGlutenFreeS$ atom holding the value $?c1$.

Figure 6.4 shows the ASP representation of inference rules defined for the Childsnack problem. Rules for DCK memory predicates $mGlutenFreeS$, $mNormalS$, and $mToServeS$ are represented as cardinality queries, also called *predicate counters*. Cardinality queries are already defined in the section 2.0.3 about the derivation of the initial ATB-DCK configuration. In this case, the $mGlutenFreeS$ and the $mNormalS$ inference rules compute the initial number of all children waiting for gluten-free and normal sandwiches (i.e., the number of *waiting* atoms defined in the initial planning state and attributed with gluten-allergic and healthy children), respectively. The $mToServeS$ predicate counter computes the initial number of waiting children at each specific table. Inference rules min and max compute the lowest and the highest number, which will be used in the corresponding planning task. The rule *count* defines a new *count* object for each number between numbers held by min and max satisfied rules, while the $next$ rule defines all pairs of newly defined *count* objects holding adjacent integer values. Finally, the last rule

specifies that only a single instance of the $make_gluten_free$ rule, attributed with a chosen tray, can be satisfied and turned into the corresponding atom.

```
min(0).
max(C) :- mGlutenFreeS(C1), mNormalS(C2), C3 = #max { N : mToServeS(_, N) }, C = #max { C1 ; C2 ; C3 }.
count(C1..C2) :- min(C1), max(C2).

next(C1, C2) :- C1 = C2 - 1, count(C1), count(C2).

mGlutenFreeS(N) :- N = #count { Ch : i_allergic_gluten(Ch), i_waiting(Ch,_) }.
mNormalS(N) :- N = #count { Ch : i_not_allergic_gluten(Ch), i_waiting(Ch,_) }.
mToServeS(T, N) :- place(T), N = #count { Ch : i_waiting(Ch, T) }, T != kitchen.

{make_gluten_free(T) : tray(T)} = 1.
```

Figure 6.4: Inference rules defined for the Childsnack domain.

Figure 6.5 shows an example of the ATB-DCK-enhanced planning problem defined for the Childsnack domain. Identifiers of *count* objects are suffixed with an integer value they represent.

```
(define (problem childsnack_instance)

    (:domain childsnack)

    (:objects
        bread1 bread2 bread3 bread4 - bread-portion
        child1 child2 child3 child4 - child
        content1 content2 content3 content4 - content-portion
        count0 count1 count2 - count
        table1 table2 - place
        sandw1 sandw2 sandw3 sandw4 - sandwich
        tray1 tray2 - tray
    )

    (:init
        ; original initial state atoms:
        (at tray2 kitchen) (at tray1 kitchen)
        (at_kitchen_bread bread2) (at_kitchen_bread bread4) (at_kitchen_bread bread1) (at_kitchen_bread bread3)
        (at_kitchen_content content1) (at_kitchen_content content2) (at_kitchen_content content4) (at_kitchen_content content3)
        (no_gluten_bread bread1) (no_gluten_bread bread4)
        (no_gluten_content content2) (no_gluten_content content3)
        (allergic_gluten child3) (allergic_gluten child1)
        (not_allergic_gluten child4) (not_allergic_gluten child2)
        (waiting child2 table1) (waiting child4 table2) (waiting child3 table2) (waiting child1 table1)
        (notexist sandw4) (notexist sandw3) (notexist sandw1) (notexist sandw2)

        ; new initial state atoms formed from satisfied inference rules:
        (mNormalS count2)
        (mGlutenFreeS count2)
        (mToServeS table2 count2)
        (mToServeS table1 count2)
        (min count0)
        (next count0 count1)
        (next count1 count2)
        (make_gluten_free tray2)
    )

    (:goal
        (and (served child1) (served child2) (served child3) (served child4))
    )

)
```

Figure 6.5: ATB-DCK-enhanced PDDL planning problem defined for the Childsnack domain.

6.0.2 Reconfigurable Machines

The Reconfigurable Machines domain describes a simplified environment of the Reconfigurable Manufacturing System (RMS) utilizing reconfigurable machines known as Reconfigurable Machine Tools [18]. Reconfigurable Machine Tool (RMT) consists of machine-compatible operational modules which

can be removed, added, or adjusted to change the RMT's functionality or its effectiveness in performing a particular task. Each configuration of the operational modules on a single RMT might be suitable for different tasks. Therefore, a single RMT might be able to perform different operations, which makes RMS highly responsive and flexible in satisfying customers' diverse and volatile demands.

Planning tasks of the Reconfigurable Machines domain are defined as task scheduling problems. Each planning task defines a list of reconfigurable machines (RMTs), a list of machine configurations, and a list of tasks that must be processed. All machines can be configured to one or more defined configurations. RMTs might be of various types, and thus, two different machines can be configured to two different sets of configurations. Each machine configuration is capable of performing a set of defined tasks. Different machine configurations might be suitable for different sets of tasks, or they can be designed to perform the same tasks with varying efficiency. Each task specifies its release time (i.e., a time from which the processing of the task can start) and deadline (i.e., a time in which the task must be already finished). All operations (tasks processing and machines reconfiguration) take some time (the duration of the operation) and some amount of resources (the cost of the operation). The goal of the planning process is the completion of all defined tasks in their specified time frames while attempting to minimize the total cost of all performed actions present in the solution plan.

Similarly to the Childsnack domain, the Reconfigurable Machines domain also works with numbers (e.g., costs of operations or tasks' release times and deadlines). However, the complexity of utilizing numbers in the Reconfigurable Machines domain is much higher. Used integers are no longer adjacent to each other, and the simple arithmetic operation of subtraction by one is replaced by the addition of arbitrarily sized numbers. Thus, the use of special objects for each used number is no longer feasible. To represent non-trivial numerical tasks, classical automated planning can be enhanced by so-called numeric fluents. Numeric fluents, introduced by the newer PDDL version 2.1 [4], are special functions similar to predicates, which can hold any arbitrarily chosen number (e.g., release times and deadlines of defined tasks or financial costs of particular operations). The time aspect of the task scheduling problem could be typically captured by defining the problem as a temporal automated planning task. Temporal planning allows the use of so-called *durative actions*, also introduced by PDDL 2.1 [4], with specified durations and the ability to capture their possible concurrency. The solution plan then includes time frames of the planning process in which were durative actions performed. However, there is no easy (without the need for external preprocessing) and widely supported way of satisfying tasks' deadlines using temporal automated planning. Therefore, for demonstration purposes, a purely numerical approach of representing time in the planning process is used instead. Figure 6.6 shows

the PDDL representation of the *process_task* planning operator defined for the Reconfigurable Machines domain, demonstrating the use of numeric fluents in the automated planning task.

```
(:action process_task
    :parameters (
        ?m - machine
        ?c - configuration
        ?t - task
    )
    :precondition (and
        (configured ?m ?c)
        (processable ?c ?t)
        (not (processed ?t))

        ; currently elapsed time for the machine ?m must be greater or
        ; equal to the release time of the task ?t
        (>= (machine-time ?m) (release-time ?t))

        ; currently elapsed time for the machine ?m plus the time in
        ; which the task ?t is processed by the machine configuration
        ; ?c must be lesser or equal to the deadline of the task ?t
        (<= (+ (machine-time ?m) (processing-time ?c ?t)) (deadline ?t))
    )
    :effect (and
        (processed ?t)
        (increase (machine-time ?m) (processing-time ?c ?t))
        (increase (total-cost) (processing-cost ?c ?t))
    )

)
```

Figure 6.6: Planning operator defined for the Reconfigurable Machines domain.

The numeric fluent *machine-time* tracks the currently elapsed time since the beginning of the planning process for all machines substituted for the argument $?m$. At the beginning of the planning process, it is assigned a value of zero. After that, it is increased in the effects of each performed action (performed by the corresponding machine $?m$) by the action's duration. The main purpose of the *machine-time* numeric fluent is to ensure that all time constraints (i.e., release times and deadlines) imposed on defined tasks are satisfied. A machine $?m$ configured to a machine configuration $?c$ can perform a task $?t$ only if its current machine time is higher or equal to the task's release time and lower or equal to the task's deadline subtracted by the task's duration when processed by the machine configuration $?c$. The release time and the deadline of the task are represented by the *release-time* and the *deadline* numeric fluents, respectively. The duration of the task being performed by the configuration $?c$ is specified by the *processing-time* numeric fluent. The planning domain model also defines the *total-cost* numeric fluent, which captures the sum of costs of all performed operations in the current step of the planning process. The value of *total-cost* is used as a metric of the objective function (defined in the planning problem) that informs automated planning engines that the metric value should be minimized. The lowest possible value of the *total-cost* numeric fluent computed from any

feasible solution plan makes the given solution plan an optimal solution for the corresponding planning task. Metrics were introduced, the same as numeric fluents and durative actions, by the PDDL version 2.1 [4]. Figure 6.7 shows the truncated version of the original planning problem defined for the Reconfigurable Machines domain. It demonstrates the initialization of numeric fluents and the specification of the metric function.

```
(define (problem reconfigurable-machines-instance)

    (:domain reconfigurable-machines)

    (:objects
        m0 m1 - machine
        c0 c1 c2 c3 - configuration
        t0 t1 t2 t3 t4 - task
    )

    (:init
        (= (release-time t0) 36) (= (release-time t1) 4) (= (release-time t2) 52)
        (= (release-time t3) 0) (= (release-time t4) 24)

        (= (deadline t0) 51) (= (deadline t1) 19) (= (deadline t2) 64)
        (= (deadline t3) 13) (= (deadline t4) 33)

        (= (machine-time m0) 0) (= (machine-time m1) 0)

        (= (total-cost) 0)

        ; rest omitted
    )

    (:goal
        (forall (?t - task)
            (processed ?t)
        )
    )

    (:metric minimize (total-cost))     ; optimal cost = 116
)
```

Figure 6.7: Planning problem defined for the Reconfigurable Machines domain.

ATB-DCK defined for the Reconfigurable Machines domain is shown in Figure 6.8. This ATB-DCK is very simple as it doesn't impose any additional constraints or modifiers (apart from its source and destination states) on any of its transitions. It just restricts the number of original actions each machine can perform according to the ATB-DCK state it currently lies in. At the beginning of the planning process, all machines are placed into the *unused* ATB-DCK state. In this state, the machine $?m$ can perform any defined action. It can process some task $?t$ while staying at the *unused* state, it can reconfigure itself and be moved into the *reconfigured* ATB-DCK state, or it can wait for the release time of some task $?t$ and be moved into the *idling* state. When the machine $?m$ is reconfigured, it can't be reconfigured again until it processes the task $?t$ for which it was reconfigured. Multiple machine reconfigurations in a row can only worsen the metric value of the current solution because each reconfiguration has some financial cost, which is added to the total cost of all performed operations. Therefore, in the *reconfigured* ATB-DCK state, the machine $?m$ can either process the task $?t$ and move

back to the *unused* state or wait for the release time of the task t and be moved into the *idling* state. After waiting for the release time of some task $?t$, it wouldn't make sense for the machine $?m$ to do anything else than process the given task. While waiting for releases of multiple tasks in a row doesn't financially cost anything, its prohibition simplifies the planning process by reducing the search space. Thus, in the *idling* ATB-DCK state, the machine $?m$ can only process the task $?t$ and move back into the *unused* state.

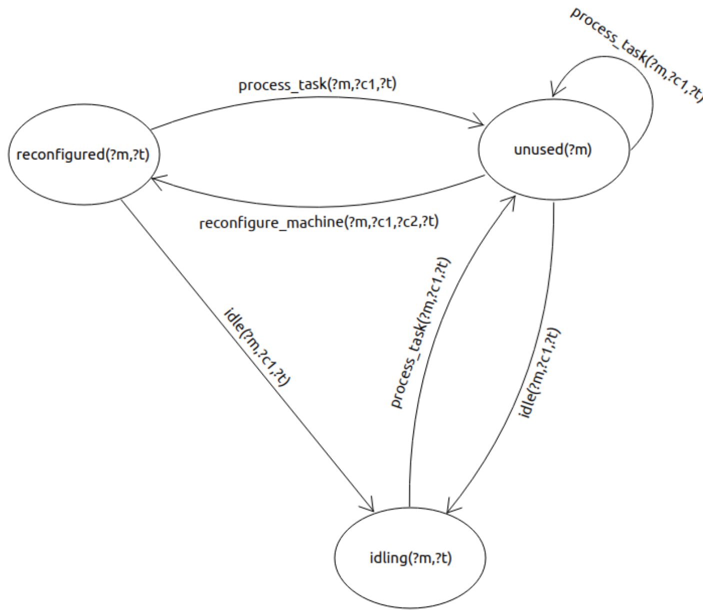

Figure 6.8: ATB-DCK defined for the Reconfigurable Machines domain.

Because ATB-DCK defined for the Reconfigurable Machines modifies preconditions of original operators only with its states, no additional DCK memory predicates need to be specified. For this domain, DCK memory predicates could be represented by an empty JSON object. As already mentioned, all machines are initially placed into the *unused* ATB-DCK state. Therefore, inference rules can be represented by a single rule as follows:

$$unused(M) \text{ :- } machine(M).$$

For each machine defined in the planning problem and substituted for M, a new *unused* atom is created and added to the initial planning state of the corresponding planning task.

6.0.3 Performance Evaluation

This section compares the performance of solving the original and the ATB-DCK-enhanced planning tasks of the Reconfigurable Machines domain. Both the Blockworld and the Childsnack problems were already, among other classical planning problems, fully tested in the paper that introduced ATB-DCK [2]. The aim of this section is to demonstrate how ATB-DCK performs for the numeric (non-classical) NP-hard planning problem of task scheduling in Reconfigurable Manufacturing Systems.

The support for numeric fluents is among domain-independent planners quite sparse. Because of the exponential growth of possible solutions, numeric planning engines typically utilize some heuristic search algorithms, such as local search [19], to preserve reasonable efficiency. Consequently, the solutions they generate can sometimes be significantly worse than the optimal ones. For testing instances of the Reconfigurable Machines problem, the numeric planner LPG-td, utilizing the heuristic method local search, was chosen.

Table 6.1 shows the test results for four instances. Each instance defines exactly two machines, four machine configurations, and from five to twenty tasks. Testing took place on a machine with 8GB of RAM and an Intel(R) Core(TM) i7-8750H processor clocked at 2.20 GHz. CPU time for finding the solution plan was limited to 3600 seconds.

		Without ATB-DCK		**With ATB-DCK**	
Instance	**Optimal cost**	**time (s)**	**cost**	**time (s)**	**cost**
2x4x5	116	0.26	116	0.26	116
2x4x10	273	1477.38	273	312.43	273
2x4x15	396	TIMEOUT	-	3047.62	578
2x4x20	583	TIMEOUT	-	TIMEOUT	-

Table 6.1: Test results from solving original and ATB-DCK-enhanced instances of the Reconfigurable Machines problem.

The first instance was solved successfully for both the original and the ATB-DCK-enhanced planning task in the same duration of time. The second instance was also solved successfully, but the ATB-DCK-enhanced planning task notably outperformed (the difference was about 20 minutes) the original one in terms of speed, despite being slow, too, for the relatively small size of the instance. The third instance was solved in time only for the ATB-DCK-enhanced planning task, but it took about 50 minutes, and the deviation of its solution's metric value from the optimum was about 45% of the optimum's value. The last instance with twenty defined tasks wasn't solved at all. Judging by the test result, it would be pointless to further test other instances

of the Reconfigurable Machines problem. It is evident that while ATB-DCK can significantly improve the performance of the original planning task, automated planning itself doesn't seem to be a suitable approach for solving NP-hard task scheduling problems, at least not yet.

It is important to realize that not every planning problem may benefit from ATB-DCK, as was proved by test results from the paper introducing ATB-DCK [2]. Each ATB-DCK transition generates a new planning operator. In some domains, the number of ATB-DCK-enhanced planning operators can be much higher than the number of original ones, which may introduce overheads that outweigh the more efficient guidance of domain-independent planning engines provided during the planning process by ATB-DCK. Predicate counters, representing integer values from possibly large intervals, could produce such overheads as well. The performance of ATB-DCK is also dependent on the chosen planning engine as different engines use different heuristics, and some heuristics might utilize ATB-DCK better (or worse) than others.

Conclusion

This thesis defined ATB-DCK and used this definition to design a language for ATB-DCK data representation. Furthermore, a graphical editor was implemented to enable the intuitive creation and visualization of ATB-DCK as a finite-state automaton. A text editor for ATB-DCK data representation, with its particular syntactical elements being distinctively colored, was developed and integrated into the graphical editor. Finally, the thesis specified ATB-DCK in three different domains with unique characteristics and compared the performance of ATB-DCK-enhanced planning tasks with the original ones defined for the NP-hard problem of task scheduling in Reconfigurable Manufacturing Systems. An integration of two planners into the graphical editor wouldn't make sense because the editor's only purpose is to create the ATB-DCK data representation. The whole project, including source codes of its individual components and test data of the experimental evaluation, can be found in the following public GitLab repository: `https://gitlab.com/hrustik97/atb-dck-graphical-editor`.

In the future development, both the graphical editor and the program for encoding ATB-DCK into the PDDL planning task specification could be merged into a single GUI (Graphical User Interface) application integrated with several domain-independent planning engines and with editors for representation of PDDL planning task specifications, DCK memory predicates, and inference rules.

Appendix A

Bibliography

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

[2] Lukáš Chrpa, Roman Barták, Jindřich Vodrážka, and Marta Vomlelová. Attributed transition-based domain control knowledge for domain-independent planning. *IEEE Transactions on Knowledge and Data Engineering*, 34(9):4089–4101, 2022.

[3] 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.

[4] Maria Fox and Derek Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

[5] Lindsay Bassett. *Introduction to JavaScript object notation: a to-the-point guide to JSON*. " O'Reilly Media, Inc.", 2015.

[6] Joshua M Willman. *Beginning PyQt*. Springer, 2020.

[7] Mark Summerfield. *Programming in Python 3: a complete introduction to the Python language*. Addison-Wesley Professional, 2010.

[8] Qt documentation. Signals & slots. `https://doc.qt.io/qt-6/signalsandslots.html`, 2023.

[9] Qt documentation. Coordinate system. `https://doc.qt.io/qt-6/coordsys.html`, 2023.

[10] Qt documentation. Layout management. https://doc.qt.io/qt-6/layout.html, 2023.

[11] Tarski documentation. https://tarski.readthedocs.io/en/latest/, 2020.

[12] Piero Bonatti, Francesco Calimeri, Nicola Leone, and Francesco Ricca. Answer set programming. *A 25-Year Perspective on Logic Programming: Achievements of the Italian Association for Logic Programming, GULP*, pages 159–182, 2010.

[13] Krzysztof R Apt. Logic programming. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, 1990:493–574, 1990.

[14] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In *Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[15] Clingo: Python api documentation. https://potassco.org/clingo/python-api/5.6/clingo/, 2018.

[16] Clorm documentation. https://clorm.readthedocs.io/en/latest/, 2018.

[17] Python documentation. Json encoder and decoder. https://docs.python.org/3/library/json.html, 2023.

[18] Yoram Koren, Uwe Heisel, Francesco Jovane, Toshimichi Moriwaki, Gumter Pritschow, Galip Ulsoy, and Hendrik Van Brussel. Reconfigurable manufacturing systems. *CIRP annals*, 48(2):527–540, 1999.

[19] Emile HL Aarts and Jan Karel Lenstra. *Local search in combinatorial optimization*. Princeton University Press, 2003.