Rheinisch-Westfälische Technische Hochschule Aachen
Knowledge-based Systems Group
Prof. Gerhard Lakemeyer, Ph. D.

Master's Thesis

Generating Macro Actions from a Plan Database for Planning on Mobile Robots

Till Hofmann

March 14, 2017

Supervisors: Prof. Gerhard Lakemeyer Ph. D.
Prof. Dr. Matthias Jarke
Advisor: Dipl.-Inform. Tim Niemueller

Acknowledgements

I would like to thank Tim Niemueller for providing critical and invaluable feedback on my work and for teaching me how to write effectively. I also wish to express my sincere thanks to Professor Gerhard Lakemeyer for providing the opportunity to work in such a great research group and for jointly supervising this thesis with Professor Matthias Jarke, which I greatly appreciate. I am very grateful for the valuable feedback from all members of the Knowledge-Based Systems Group, whose questions gave me the initial idea for this thesis.

I want to thank the whole Carologistics Team for a great RoboCup in Leipzig, which gave me some valuable insights for this thesis. Great thanks also goes to Gesche Gierse and Bahram Maleki-Fard for their work in the KBSG robotics lab.

Last but not least, I would like to thank my friends, my parents and my brothers and sisters for their support and encouragement.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Robots such as the vacuum cleaner Roomba [39] have started to be part of our lives and are already used for manufacturing tasks such as automotive assembly [50]. All these robots perform a single simple task or act in a well-known environment. However, robotics in more open environments with complex tasks is still an active research field [31] and has not yet been widely adopted outside of research. One challenge in such an environment is planning, which is a "process that chooses and organizes actions by anticipating their expected outcomes" [31]. There are various types of planning, e.g., path planning, manipulation planning, and task planning. In this thesis, we focus on domain-independent symbolic task planning, where the agent has to find a sequence of symbolic actions to reach a goal state from an initial state.

One of the major challenges of planning in robotics are long planning times, i.e., the time that a planner needs to find a solution for a given problem. While short planning times are desirable in general, they are even more so for robotics applications. For one, in many applications such as the museum tour-guide [10], robots interact with humans. If an interactive robot takes several minutes to reason about its actions, it becomes unreactive and the user will claim that the robot is broken. Even without human interaction, time is usually a limited resource in robotics, which can be seen in the scenario from the RoboCup Logistics League (RCLL) [64], where a team of robots needs to transport workpieces between production machines in order to fulfill orders within 15 minutes. Clearly, in such a scenario, planning times of several minutes are unacceptable. This becomes even more crucial with continual planning, where plan generation is interleaved with plan execution and sensing [8], and re-planning occurs frequently. Therefore, planning should not take longer than a few seconds.

On the other hand, planning is a difficult task. In fact, it is PSPACE-complete when constrained to propositional planning [11], and NEXPTIME-complete in the general case [24], which means it cannot be solved efficiently in general. Instead, current state-of-the-art planning systems such as Fast Downward [32] use heuristic approaches to solve planning problems in a reasonable time. Fast Downward uses heuristic forward search in the world state space, i.e., starting in the initial state, the planner generates successor states by applying all possible actions and evaluates the resulting states with a heuristic function. It then picks the successor state with the best heuristic value and continues the search from there.

When looking at the resulting plans from a robotics domain such as the Cleanup domain [35], we observe that those plans often look very similar and contain the same sub-sequences of actions. In such a domain, *assertions* [8] - placeholder actions that defer parts of of the planning task to later - can significantly reduce planning time [37], but they require a good understanding of the domain by the designer. A different approach

is taken by macro planners such as REFLECT and MACRO-FF, which automatically concatenate actions to *macros*, either by analyzing the domain (REFLECT), or by using previous planning results (MACRO-FF).

In this thesis, we present *database-driven macro planning* (DBMP), which automatically generates macro actions from previous planning results by identifying frequent action sequences in previous plans and re-using those action sequences for planning subsequent problems. We store previous planning results in a database, identify frequent action sequences in the database, and generate macros from those frequent action sequences. In contrast to other approaches, macros are represented as normal operators including conditional and quantified preconditions and effects. Therefore, no modifications of the underlying planner are necessary. To find such a representation, we provide a formal definition of a *macro representation of an operator sequence*, and we describe how to generate a macro's precondition and effects from an operator sequence by *regressing* all preconditions and *chaining* all effects of the operator sequence. From the generated macros, *evaluators* select the presumably most useful ones by assigning a score to each macro according to the macro's properties such as its frequency. The selected macros are added to the original domain, which can then be used to solve further planning problems. In the final step, macros in the resulting plans are expanded to the original action sequence so the plan can be executed by the agent.

This thesis is structured as follows: In Chapter 2, we introduce planning and the Planning Domain Definition Language (PDDL) with a particular focus on the Action Definition Language (ADL) fragment of PDDL. We summarize the main concepts of the heuristic planning systems FAST-FORWARD and FAST DOWNWARD. We give an overview of the Situation Calculus, the logic $\mathcal{ES}$, and ADL semantics based on $\mathcal{ES}$. We will use $\mathcal{ES}$ and the ADL semantics to define preconditions and effects of our macro operators. Then, we present the RoboCup Logistics League (RCLL) as one evaluation domain for this thesis, describe the robot framework Fawkes as the underlying software system, and summarize database concepts relevant for this thesis, before we give an introduction into cloud computing and containerization, which we use for benchmarking. In Chapter 3, we describe existing macro planners and the differences to DBMP and then present DBMP in detail in Chapter 4. In Chapter 5, we present a detailed evaluation of DBMP using domains from the previous International Planning Competition (IPC 2014) [75], the upcoming robotics planning competition based on the RCLL [57], and the *Cleanup* scenario from our lab [35]. We summarize our experience with using Kubernetes for running a large number of benchmark tasks, we investigate the performance of macro identification and generation, we analyze planning times and plan lengths for the macro-augmented domains, and we examine how well our evaluators selected macros. We conclude with a summary and future work in Chapter 6.

2 Background

In this section, we introduce classical planning, current state-of-the-art classical planners, the RoboCup Logistics League (RCLL) as benchmark domain, Fawkes as underlying software framework, relevant database concepts, and we give an overview of cloud computing, virtualization, and containerization.

2.1 Classical Planning

In the most general term, *planning* describes the problem of finding a sequence of *actions* to reach a certain world state, called the *goal state*, from an *initial state* [72]. In classical planning, a world state is represented by a set of propositions which are true. Actions are represented by a set of *preconditions* and *effects*. The preconditions must hold when the action is to be executed, the effects hold after the action has been executed. Depending on the formalism, there are restrictions on how the world state, the preconditions, and the effects can be represented. In the following, we will introduce different representation formalisms, starting with STRIPS and continuing with the de-facto standard formalism, the Planning Domain Definition Language (PDDL) and its different dialects.

2.1.1 Stanford Research Institute Problem Solver (STRIPS)

The Stanford Research Institute Problem Solver (STRIPS) [65] was introduced in 1971 and is both a representation formalism for planning problems and a solver for problems described in that formalism. Today, STRIPS is mainly known for the representation formalism. In STRIPS, the world model is represented as a set of propositions, which contains all propositions that are true. All propositions not in the set are assumed to be false. Thus, STRIPS makes the closed-world assumption. Actions are represented as operators. An operator is defined by its *name*, *parameters*, *precondition*, and *effects*. Precondition and effects are defined by sets of propositions. The effect of an operator is defined by two sets: The *add list* contains all propositions that are added to the world model after applying the operator. The *delete list* contains all propositions that are removed from the world model after applying the operator.

2.1.2 Planning Domain Definition Language (PDDL)

The Planning Domain Definition Language (PDDL) [48] was introduced by the Planning Competition Committee of the International Conference on Artificial Intelligence Planning Systems (AIPS), which today is part of the International Conference on Automated Planning and Scheduling (ICAPS). PDDL is a problem-specification language

with the goal to "improve empirical evaluation of planner performance" [48] by providing a standard format for problem definitions. PDDL is not a single formalism but a set of formalisms with different expressivity. Among others, it supports basic STRIPS actions, conditional effects, and existential and universal quantification. A PDDL description consists of two parts: The domain description and the problem description.

The *domain description* defines the available syntactic features (*requirements*), types, constants, predicates and actions. If no requirement is given, the requirement `:strips` is implicitly defined. Other formalisms are used by adding requirements to the domain specification. Each requirement adds features to the language, e.g., conditional effects can be used by adding the requirement `:conditional-effects`. The requirements constrain the applicable planning systems and can severely impact complexity. Among others, the language features shown in Table 2.1 are supported. *Strict* PDDL has the additional restriction that all domain keywords must appear in a particular order and each file can contain only one domain description.

Requirement	Description
`:strips`	Effects and preconditions are conjunctions of atomic propositions
`:typing`	Variables can have types
`:disjunctive-preconditions`	Allow '`or`' in preconditions and goal descriptions
`:conditional-effects`	Actions can have additional effects if a given condition is true, e.g., `(when (holding ?o) (at ?o ?l))`
`:equality`	Support '`=`' as built-in predicate
`:existential-preconditions`	Allow existential quantification in preconditions and goals, e.g., `(exists (?l - location) (aligned ?l))`
`:universal-preconditions`	Allow universal quantification in preconditions and goals, e.g., `(forall (?o - object) (not (holding ?o)))`
`:quantified-preconditions`	= `:existential-preconditions` + `:universal-preconditions`
`:adl`	= `:strips` + `:typing` + `:equality` + `:disjunctive-preconditions` + `:quantified-preconditions` + `:conditional-effects`

Table 2.1: A selection of possible PDDL requirements. Adapted from [48].

An action definition consists of a list of effects and an optional precondition. *Effects* are given as a conjunction of literals, in addition to conditional and universally quantified effects if the requirement `:conditional-effects` is given. Note that disjunctive effects are not allowed. In contrast, *preconditions* and *goal descriptions* can contain disjunctions, implications, existential quantifiers and universal quantifiers, depending on the given

requirements from Table 2.1. Therefore, action preconditions and goal definitions are considerably more expressive than action effects.

The second part of a PDDL description is the *problem definition*. A problem definition consists of an *initial situation* and a *goal* to be achieved. The initial situation is given as a set of true atomic formulas, anything not mentioned in the initial situation is assumed to be false. A goal is defined by a goal description in the same way as action preconditions.

PDDL2.1 and PDDL3 PDDL2.1 [27] is an extension to PDDL that adds numeric fluents and expressions, durative actions, and plan metrics, and thus allows for temporal modelling. PDDL3 [28] is another extension that supports constraints and soft goals. In this thesis, none of the additional features is used.

PDDL+ PDDL+ [26] extends PDDL with continuous aspects which allows the modelling of mixed discrete and continuous change. Additionally, it supports the modelling of predictable exogenous events such as an empty battery, which occurs after the robot has been driving around for a certain time. A PDDL+ task can be encoded as a hybrid automaton and solved with a Satisfiability Modulo Theories (SMT) solver [9, 12]. In this thesis, we will not support PDDL+ planning tasks.

ADL The Action Description Language (ADL) [66] was originally formulated as extension to STRIPS, but can also be described as a subset of PDDL. PDDL provides the `:adl` requirement (cf., Table 2.1), which allows typed objects, disjunctive preconditions, a built-in equality predicate, quantified preconditions, and conditional effects.

Listing 2.1: The action `goto(?to)` formulated as PDDL ADL action.

```
(:action goto
  :parameters (?to - location)
  :precondition
    (and
      (not (exists (?l - location) (aligned ?l)))
      (not (robot-at ?to)))
  :effect
    (and
      (robot-at ?to)
      (forall (?loc - location)
        (when (not (= ?loc ?to)) (not (robot-at ?loc))))
    )
)
```

An example for an action definition is shown in Listing 2.1. The precondition requires that the robot is not aligned to any location, because if it is aligned, it is very close to the location, and therefore needs to `back-off` first. Second, the robot cannot already

be at the goal location `?to`. The action `goto` has two effects: First, the robot is at the goal location `?to`. Second, for any location `?loc` that is not the goal location `?to`, the robot is not at `?loc` after doing action `goto(?to)`. Thus, after executing `goto(?l)`, the robot is at exactly one location, namely at `?l`.

2.1.3 Definite Clause Grammars

Definite clause grammars (DCG) [67] is a formalism that extends context-free grammars with context-dependency and extra conditions to be included in grammar rules, which are evaluated during parsing. A DCG can be implemented in Prolog [20]. Listing 2.2 shows a simple DCG rule for a PDDL requirement (e.g., `:adl`). The extra conditions starting in line 3 restrict the requirement to strings starting with ":" and are evaluated as normal Prolog predicates. Also, the input string `R` is transformed into a Prolog atom `RAtom`. In addition to parsing, DCG Prolog implementations can also be used to generate strings of the language. We will use DCGs to parse and generate PDDL domains.

Listing 2.2: A DCG rule for parsing a PDDL requirement.

```
1 requirement(RAtom) -->
2   [R],
3   {
4     atom_string(RAtom, R),
5     string_concat(":", _, R)
6   }.
```

2.2 Situation Calculus

The Situation Calculus was originally formulated as a first-order language by McCarthy [47] and was later extended by Reiter to a second-order language [70]. The central concept of the Situation Calculus are *situations* which describe the current world state. In the Situation Calculus, situations are represented by a world history, always starting in the initial situation S_0. A world history consists of a sequence of *actions*. Similar to PDDL actions, each action in the Situation Calculus has a precondition that must be satisfied in order to perform the action, and each action changes certain facts about the world. The Situation Calculus has a special function symbol $do(\alpha, s)$, which describes the situation after performing action α in situation s. As an example, the situation after performing the actions `<goto(table), align-to(table)>` starting in the initial situation S_0 is denoted with:

$$do(\textit{align-to}(\textit{table}), do(\textit{goto}(\textit{table}), S_0))$$

In the Situation Calculus, relations and functions that may change their value from situation to situation are called *fluent*, while relations and functions that do not change are called *rigid*.

In the following, we introduce the language $\mathcal{L}_{sitcalc}$ as defined by Pirri & Reiter [68]. For a more detailed introduction, we refer to [45].

2.2.1 The Language $\mathcal{L}_{sitcalc}$

The language $\mathcal{L}_{sitcalc}$ is second-order language with equality. It is a sorted logic with the three sorts *action*, *situation*, and *object*. It has the following alphabet:

1. countably infinitely many individual variable symbols of each sort:

$$a_1, a_2, \ldots, s_1, s_2, \ldots, o_1, o_2, \ldots$$

2. a constant symbol S_0 of sort *situation*, which denotes the initial situation;
3. a binary function symbol $do : \mathit{action} \times \mathit{situation} \rightarrow \mathit{situation}$, where $do(a, s)$ denotes the successor situation resulting from performing action a in situation s;
4. a binary predicate symbol $Poss : \mathit{action} \times \mathit{situation}$, where $Poss(a, s)$ describes that it is possible to perform the action a in situation s;
5. for each $n \geq 0$, rigid predicate symbols of arity n and sorts $(\mathit{action} \cup \mathit{object})^n$ to denote situation-independent relations;
6. for each $n \geq 0$, rigid function symbols of arity n and sort $(\mathit{action} \cup \mathit{object})^n \rightarrow \mathit{object}$ to denote situation-independent functions;
7. for each n, predicate symbols with arity $n+1$ and sorts $(\mathit{action} \cup \mathit{object})^n \times \mathit{situation}$ to denote relations that depend on the current situation. As the truth value of these predicates may change from situation to situation, they are also called relational fluents;
8. for each n, function symbols of sort $(\mathit{action} \cup \mathit{object})^n \times \mathit{situation} \rightarrow \mathit{action} \cup \mathit{object}$ to denote functions that depend on the situation. As the value of these functions may change from situation to situation, they are also called functional fluents.

Furthermore, $\mathcal{L}_{sitcalc}$ uses the standard logical symbols $\wedge, \neg, \exists$ with the usual definitions.

2.2.2 Basic Action Theories

A *basic action theory* is a set of axioms that describe a particular domain. It defines possible situations, action preconditions and effects, and the initial situation S_0. In the Situation Calculus, it also includes foundational axioms such as the unique-name axioms, and axioms that fix certain properties of situations, e.g., that situations are finite sequences of actions. Additionally, a basic action theory also incorporates a solution to the *frame problem* as proposed by Reiter [71]. The frame problem states that if we define the effects of an action, we cannot only specify which fluents are changed by the

action, but we also need to specify all fluents that do not change. Axioms that specify the action invariants are called *frame axioms*. Reiter's solution to the frame problem is to make a completeness assumption, i.e., the conditions characterized in the basic action theory are the only conditions under which an action may cause a fluent to change.

We first define action precondition axioms and successor state axioms, and then define a basic action theory using those axioms.

Definition 2.2.1 (Action Precondition Axiom [45])**.**
An action precondition axiom is a sentence of the form:

$$Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$$

where A is an action function symbol, and $\Pi_A(\vec{x}, s)$ is a formula that is uniform in s and whose free variables are among $\vec{x}, s$.

As an example, the action precondition axiom of the `goto` action shown in Listing 2.1 can be formulated as:

$$Poss(goto(to), s) \equiv \neg robot\text{-}at(to, s) \wedge \neg \exists l\ aligned(l, s)$$

Definition 2.2.2 (Successor State Axiom [45])**.**

1. A successor state axiom for a relational fluent F is a sentence of $\mathcal{L}_{sitcalc}$ of the form:

   $$F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$$

   where $\Phi_F(\vec{x}, a, s)$ is a formula uniform in s, all of whose free variables are among $a, s, \vec{x}$. Following Reiter's solution to the frame problem, $\Phi_F(\vec{x}, a, s)$ has the following form:

   $$\Phi_F(\vec{x}, a, s) \equiv \gamma_F^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s)$$

   The relational fluent $F(\vec{x})$ is true iff performing action a in situation s makes the fluent true ($\gamma_F^+(\vec{x}, a, s)$) or if the fluent was true in situation s ($F(\vec{x}, s)$) and performing action a in situation s does not change the fluent to be false ($\neg\gamma_F^-(\vec{x}, a, s)$).

2. A successor state axiom for a functional fluent f is a sentence of $\mathcal{L}_{sitcalc}$ of the form

   $$f(\vec{x}, do(a, s)) = y \equiv \phi_f(\vec{x}, y, a, s)$$

   where $\phi_f(\vec{x}, y, a, s)$ is a formula uniform in s, all of whose free variables are among $\vec{x}, y, a, s$. Again following Reiter's solution to the frame problem, $\phi_f(\vec{x}, y, a, s)$ has the following form:

   $$\phi_f(\vec{x}, y, a, s) \equiv \gamma_f(\vec{x}, y, a, s) \vee f(\vec{x}, s) = y \wedge \neg\exists y'.\, \gamma_f(\vec{x}, y', a, s)$$

   The functional fluent $f(\vec{x})$ has value y iff performing action a in situation s causes the fluent to be changed to y ($\gamma_f(\vec{x}, y, a, s)$) or if the fluent had the value y in situation s ($f(\vec{x}, s) = y$) and performing action a does not change the fluent value to any other value ($\neg\exists y'.\, \gamma_f(\vec{x}, y', a, s)$).

As an example, assuming `goto` is the only action in our domain, the successor state axiom for the fluent *robot-at* can be formulated as:

$$\textit{robot-at}(l, do(a, s)) \equiv a = \textit{goto}(l) \vee \textit{robot-at}(l, s) \wedge \neg\exists l_2.\, a = \textit{goto}(l_2)$$

Definition 2.2.3 (Basic Action Theory (Situation Calculus) [68])**.**
In the Situation Calculus, a *basic action theory* is a theory $\mathcal{D}$ of the form

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{SS} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}$$

where

- Σ are the foundational axioms for situations,
- $\mathcal{D}_{SS}$ is a set of successor state axioms, one for each fluent of the language $\mathcal{L}_{sitcalc}$,
- $\mathcal{D}_{ap}$ is a set of action precondition axioms, one for each action function symbol of the language $\mathcal{L}_{sitcalc}$,
- $\mathcal{D}_{una}$ is a set of unique names axioms for all actions,
- $\mathcal{D}_{S_0}$ is the initial database, i.e., S_0 is the only term of sort *situation* mentioned by the sentences of $\mathcal{D}_{S_0}$.

2.3 The Logic $\mathcal{ES}$

The logic $\mathcal{ES}$ [43] is a modal logic for "reasoning about the knowledge, action, and perception of an agent" [42]. It is a logical variant of the Situation Calculus. As the Situation Calculus, it is based on the notion of *situations*. However, in contrast to the Situation Calculus, situations do not appear as terms in the language. Instead, situations are part of the semantics. As an example, the fact that the robot is in the kitchen after doing action *goto(kitchen)* is expressed with $[\textit{goto}(\textit{kitchen})]\, \textit{at}(\textit{kitchen})$. Since there are no situation terms, there is also no sort *situation*. Additionally, the sorts *object* and *action* are not distinguished.

$\mathcal{ES}$ is an epistemic logic, i.e., it allows to express the agent's knowledge. However, in the following, we will only describe the objective, non-epistemic subset of $\mathcal{ES}$. We start by defining the alphabet of the language $\mathcal{L}_{\mathcal{ES}}$, its terms and formulas, and then give a short introduction into the semantics of $\mathcal{ES}$. For more details, see [43, 42].

2.3.1 The Language $\mathcal{L}_{\mathcal{ES}}$

The language $\mathcal{L}_{\mathcal{ES}}$ is a modal logic with the following alphabet [43]:

- countably infinitely many variable symbols

$$V = \{x_1, x_2, \ldots, y_1, y_2, \ldots, z_1, z_2, \ldots, a_1, a_2, \ldots\}$$

- for each $n \geq 0$, fluent predicate symbols of arity n: $F^n = \{f_1^n, f_2^n, \ldots\}$,
- a distinguished unary predicate symbol *Poss*, where $Poss(a)$ describes that it is possible to perform action a,
- for each $n \geq 0$, rigid function symbols of arity n: $G^n = \{g_1^n, g_2^n, \ldots\}$,
- connectives and other symbols: $=, \wedge, \neg, \forall, \Box$, round and square parentheses, period, comma.

Terms Using the alphabet above, we can describe the terms of the language. The terms of $\mathcal{L}_{\mathcal{ES}}$ are the least set such that:

1. Every first-order variable is a term.
2. If $t_1, \ldots, t_n$ are terms and $g \in G^n$, then $g(t_1, \ldots, t_n)$ is a term.

We denote the set of all ground terms with R. In contrast to the Situation Calculus, there is no distinction between the sorts *actions* and *objects*, all terms are of the same sort.

Formulas The well-formed formulas of $\mathcal{L}_{\mathcal{ES}}$ of the language are the least set such that

1. If $t_1, \ldots, t_n$ are terms and $F \in F^n$, then $F(t_1, \ldots, t_n)$ is an atomic formula.
2. If t_1 and t_2 are terms, then $(t_1 = t_2)$ is a formula.
3. If t is a term and α is a formula, then $[t]\,\alpha$ is a formula.
4. If α and β are formulas, then $(\alpha \wedge \beta)$, $\neg\alpha$, $\forall x.\, \alpha$, and $\Box\alpha$ are formulas.

2.3.2 Semantics of $\mathcal{ES}$

In $\mathcal{ES}$, models are called *worlds*. Intuitively, a world w determines which fluents are true after any sequence of actions. As an example, $w \models at(kitchen)$ expresses that the robot is in the kitchen initially, and $w \models [goto(hall)]\, at(hall)$ states that after doing action *goto(hall)*, the robot is in the hall. It follows a formal definition of a *world* and truth of a formula α in a given world w.

Definition 2.3.1 (Semantics of objective $\mathcal{ES}$ [17])**.**

1. Let P denote the set of all pairs $\sigma{:}\rho$, where $\sigma \in R^*$ is a sequence of actions, and $\rho = F(r_1, \ldots, r_n)$ is a ground fluent atom from F^n. A *world* is a mapping from P to truth values $\{0, 1\}$.

2. Given a world w, for any formula α with no free variables:

$$\begin{aligned}
w, \sigma &\models F(r_1, \ldots, r_n) && \text{iff } w\,[\sigma{:}F(r_1, \ldots, r_n)] = 1 \\
w, \sigma &\models (r_1 = r_2) && \text{iff } r_1 \text{ and } r_2 \text{ are identical} \\
w, \sigma &\models (\alpha \wedge \beta) && \text{iff } w, \sigma \models \alpha \text{ and } w, \sigma \models \beta \\
w, \sigma &\models \neg\alpha && \text{iff } w, \sigma \not\models \alpha \\
w, \sigma &\models \forall x.\, \alpha && \text{iff } w, \sigma \models \alpha^x_r \text{ for every } r \in R \\
w, \sigma &\models [r]\, \alpha && \text{iff } w, \sigma \cdot r \models \alpha \\
w, \sigma &\models \Box\alpha && \text{iff } w, \sigma \cdot \sigma' \text{ for every } \sigma' \in R^*
\end{aligned}$$

2.3.3 Basic Action Theories

As in the Situation Calculus, an $\mathcal{ES}$ basic action theory is a set of axioms describing a particular domain. As before,

- Σ_0 expresses what is initially true,
- Σ_{pre} is one large precondition axiom that defines for each action when it can be performed,
- and Σ_{post} is a set of successor state axioms, one per fluent, which incorporate Reiter's solution to the frame problem.

Since $\mathcal{ES}$ does not have an explicit notion of situations, no foundational axioms for situations are necessary. It follows a formal definition of a basic action theory in $\mathcal{ES}$.

Definition 2.3.2 (Basic Action Theory ($\mathcal{ES}$) [43])**.**
Given a set of fluent predicates $\mathcal{F}$, a set $\Sigma \subseteq \mathcal{ES}$ of sentences is called a *basic action theory* over $\mathcal{F}$ iff $\Sigma = \Sigma_0 \cup \Sigma_{pre} \cup \Sigma_{post}$, where Σ mentions only fluents in $\mathcal{F}$ and

1. Σ_0 is any set of fluent sentences,
2. Σ_{pre} is a singleton sentence of the form $\Box Poss(a) \equiv \pi$, where π is a fluent formula,
3. Σ_{post} is a set of sentences of the form $\Box\,[a]\, f(\vec{x}) \equiv \gamma_f$, one for each fluent $f \in \mathcal{F}$ and where γ_f is a fluent formula.

As an example, assuming `goto` is the only action in the domain, the precondition axiom Σ_{pre} can be formulated as:

$$\Box\, Poss(a) \equiv \exists l.\, a = goto(l) \wedge \neg robot\text{-}at(l) \wedge \neg\exists l.\, aligned(l)$$

Similarly, Σ_{post} contains the following successor state axiom for the fluent *robot-at*:

$$\Box\,[a] robot\text{-}at(l) \;\equiv\; a = goto(l) \vee robot\text{-}at(l) \wedge \neg\exists l'.\, a = goto(l')$$

2.4 Declarative ADL Semantics

Claßen and Lakemeyer proposed a declarative semantics of ADL as progression in $\mathcal{ES}$ [17]. They translate an ADL problem description into a basic action theory of $\mathcal{ES}$, define progression in $\mathcal{ES}$, and show that the state-transitional semantics for ADL corresponds to first-order progression in $\mathcal{ES}$. We will use these semantics to formalize macro operators in Chapter 4. In the following, we summarize Claßen and Lakemeyer's ADL semantics. We introduce ADL operators consisting of the operator's parameters, precondition formula, and effect formula and present how we can define an ADL problem description in $\mathcal{ES}$. We then summarize how an ADL problem description can be translated into a basic action theory. We refer to [17] for a more detailed description and to [16] for an extension to temporal PDDL.

2.4.1 ADL Operators

First, we describe how we can define ADL operators consisting of a precondition formula and an effect formula in $\mathcal{ES}$.

Definition 2.4.1 (ADL precondition formula)**.**
An ADL *precondition formula* is an $\mathcal{ES}$ formula of the following form:

- An atomic formula $F(\vec{t})$ is a precondition formula if each of the t_i is either a variable or a constant.
- An equality atom $(t_1 = t_2)$ is a precondition formula if each t_i is a variable or a constant.
- If ϕ_1 and ϕ_2 are precondition formulas, then so are $\phi_1 \wedge \phi_2$, $\neg\phi_1$, and $\forall x{:}\tau\ \phi_1$.

The quantifier $\forall x{:}\tau$ stands for "all x of type τ", where a type τ is a unary predicate from F^1 and $\tau(x)$ means that object x has type τ. The quantifier $\forall x{:}\tau$ is defined as:

$$\forall x{:}\tau\ \phi \stackrel{def}{=} \forall x\ \tau(x) \supset \phi$$

Furthermore, we denote tuples of terms and types with vectors, e.g., $\vec{t}$ and $\vec{\tau}$. If $\vec{\tau}$ denotes $\tau_1, \ldots, \tau_k$, $\vec{r}$ denotes $r_1, \ldots, r_k$ and $\vec{t}$ denotes $t_1, \ldots, t_k$, then we use the following short-hand notations:

$$(\vec{r} = \vec{t}) \stackrel{def}{=} (r_1 = t_1) \wedge \cdots \wedge (r_k = t_k)$$
$$\vec{\tau}(\vec{t}) \stackrel{def}{=} \tau_1(t_1) \wedge \cdots \wedge \tau_k(t_k)$$

We denote the free variables of type τ in a formula ϕ with $\mathit{Free}(\tau, \phi)$.

Definition 2.4.2 (ADL effect formula)**.**
An ADL *effect formula* is an $\mathcal{ES}$ formula of the following form:

- An atomic formula $F(\vec{t})$ is an effect formula if each of the t_i is either a variable or a constant.

- A negated atomic formula $\neg F(\vec{t})$ is an effect formula if each of the t_i is either a variable or a constant.
- If ψ_1 and ψ_2 are effect formulas, then $\psi_1 \wedge \psi_2$ and $\forall x{:}\tau.\ \psi_1$ are effect formulas.
- If γ is a precondition formula and ψ is an effect formula not containing "$\Rightarrow$" and "$\forall$", then $\gamma \Rightarrow \psi$ is an effect formula.

Definition 2.4.3 (ADL operator: general form)**.**
An ADL operator A is given by a triple $(\vec{y}{:}\vec{\tau}, \pi_A, \epsilon_A)$, where

- A is a symbol from G^p, where p is the number of parameters $\vec{y}$ of A,
- $\vec{y}{:}\vec{\tau}$ is a list of variable symbols with associated types,
- π_A is a precondition formula with free variables among $\vec{y}$,
- and ϵ_A is an effect formula with free variables among $\vec{y}$.

We also call $\vec{y}{:}\vec{\tau}$ the parameters of operator A. As an example, consider again the action `goto` from Listing 2.1. The ADL operator can be defined as the triple $(\vec{y}{:}\vec{\tau}, \pi_{goto}, \epsilon_{goto})$ with

$$\begin{aligned} \vec{y}{:}\vec{\tau} &= \mathit{to}{:}\mathit{location} \\ \pi_{goto} &= \neg \mathit{robot\text{-}at}(l) \wedge \neg \exists l'{:}\mathit{location}.\ \mathit{aligned}(l) \\ \epsilon_{goto} &= \mathit{robot\text{-}at}(l) \wedge \forall l'{:}\mathit{location}.\ l \neq l' \Rightarrow \neg \mathit{robot\text{-}at}(l') \end{aligned}$$

Definition 2.4.4 (ADL operator: normal form)**.**
An ADL operator A is in *normal form*, if its effect ϵ_A is of the following form:

$$\begin{aligned} &\bigwedge_{F_j} \forall \vec{x}_j{:}\vec{\tau}_{F_j}.\ \left(\gamma^{+}_{F_j,A}(\vec{x}_j) \Rightarrow F_j(\vec{x}_j)\right) \wedge \\ &\bigwedge_{F_j} \forall \vec{x}_j{:}\vec{\tau}_{F_j}.\ \left(\gamma^{-}_{F_j,A}(\vec{x}_j) \Rightarrow \neg F_j(\vec{x}_j)\right) \end{aligned}$$

If an ADL operator is in normal form, then for each F_j, there is at most one conjunct of the form $\cdots \Rightarrow F_j(\vec{x})$ and also at most one conjunct of the form $\cdots \Rightarrow \neg F_j(\vec{x})$.

2.4.2 ADL Problem Description

Using the definitions above, we can now describe how we can formulate an ADL problem description in $\mathcal{ES}$.

Definition 2.4.5 (ADL problem description)**.**
A problem description for ADL is given by

1. a finite list of types $\tau_1, \ldots, \tau_l, \mathit{Object}$, where *Object* is a special type that must always be included,

2. a finite list of statements of the form

$$\tau_i : \Big(\text{either } \tau_{i_1} \dots \tau_{i_{k_i}}\Big)$$

defining some of the types as compound types, where τ_i is the union of all τ_{i_j} and k_i is the number of sub-types of τ_i; a *primitive type* is a type other than *Object* that does not occur on the left-hand side of such a definition,

3. a finite list of fluent predicates $F_1, \dots, F_n$ with a list of types $\tau_{j_1}, \dots \tau_{j_{k_j}}$ for each F_j, which defines the types of the arguments of F_j,

4. a finite list of objects with associated primitive types $o_1{:}\tau_{o_1}, \dots, o_k{:}\tau_{o_k}$, where each o_i is a symbol from G^0,

5. a finite list of ADL operators $A_1, \dots, A_m$ in normal form, where each operator only contains symbols from the operator's parameters, and from (1), (3), and (4),

6. an initial state I that only contains symbols from (1), (3), and (4), and

7. a goal description G in form of a precondition formula, which only contains symbols from (1), (3), and (4).

2.4.3 Basic Action Theories

Given an ADL problem description, a corresponding $\mathcal{ES}$ basic action theory can be constructed as follows [17]:

Successor State Axioms Σ_{post} A set of operator descriptions $\{A_1, \dots, A_m\}$ can be transformed into a set of successor state axioms Σ_{post}. Let

$$\gamma^+_{F_j} \stackrel{def}{=} \bigvee_{\gamma^+_{F_j,A_i} \in NF(A_i)} \exists \vec{y}_i.\, a = A_i(\vec{y}_i) \wedge \gamma^+_{F_j,A_i}$$

$$\gamma^-_{F_j} \stackrel{def}{=} \bigvee_{\gamma^-_{F_j,A_i} \in NF(A_i)} \exists \vec{y}_i.\, a = A_i(\vec{y}_i) \wedge \gamma^-_{F_j,A_i}$$

where $\gamma^\pm_{F_j,A_i} \in NF(A_i)$ means that there only is a disjunct for A_i if there exists a $\gamma^\pm_{F_j,A_i}$ in the normal form of A_i.

Using the definitions for $\gamma^\pm_{F_j,A_i}$, we can define the successor state axiom for F_j:

$$\Box\,[a]\, F_j(\vec{x}_j) \equiv \gamma^+_{F_j} \wedge \vec{\tau}_{F_j}(\vec{x}_j) \vee F_j(\vec{x}_j) \wedge \neg\gamma^-_{F_j}$$

The Precondition Axiom Σ_{pre} The precondition axiom π is a disjunction over all operators of the problem domain:

$$\pi \stackrel{def}{=} \bigvee_{1 \le i \le m} \exists \vec{y}_i{:}\vec{\tau}_i.\, a = A_i(\vec{y}_i) \wedge \pi_{A_i}$$

Initial Description Σ_0 The initial description Σ_0 encodes the information about the initial state of the world and all information about the types of objects.

2.5 State-of-the-art PDDL Planners

In the following, we present the state-of-the-art PDDL planners FF and FAST DOWNWARD, which are used as basic planning systems in this thesis. However, the implementation will mostly be independent of the underlying planning system. Therefore, the planner should be easily substitutable.

2.5.1 Fast-Forward

FAST-FORWARD (FF) [34] is a heuristic planner which uses forward state space search. Generally, in forward state space search, the planner starts in the initial state and generates successor states by applying the effects of some applicable action. FF uses a heuristic that estimates goal distances by ignoring delete lists. FF was originally designed for the STRIPS subset of PDDL, but has been extended to ADL [34].

The Search Algorithm Enforced Hill-climbing is a variant of hill climbing which inspects not only neighbor states but also states which are more than one step away. Since plateaus are usually fairly small in planning problems [34], the search usually finds a successor quickly and outperforms normal hill-climbing. Enforced hill-climbing uses breadth first search to find a state with a strictly better evaluation than the current state. For the evaluation, it uses the heuristic described below.

The Heuristic FF uses an adapted version of GRAPHPLAN [4] on the relaxed problem as heuristic. In the relaxed problem, the delete lists of all actions are ignored. GRAPHPLAN generates a directed, layered graph called *planning graph* which contains two kinds of nodes: *fact nodes* and *action nodes*, and three kind of edges: *precondition edges*, *add edges*, and *delete edges*. Each layer consists of one kind of node and is therefore called *fact layer* or *action layer*, respectively. Fact layers and action layers alternate, one fact layer and one action layer together are called a *time step*. In each time step i, we have all facts which are possibly true after i time steps, and all actions which are possibly applicable after i time steps. Precondition edges are edges between fact nodes in time step $i-1$ and action nodes in time step i which have the corresponding fact as precondition. Similarly, add edges and delete edges connect actions in time step i and their add and delete effects in time step $i+1$, respectively.

GRAPHPLAN runs in stages. In stage i, GRAPHPLAN extends the planning graph from stage $i-1$ by adding an action level with all possible actions and a fact layer with the actions' effects. It then searches the planning graph for a valid plan recursively. Given a set of goals at time t, it searches a set of actions at time $t-1$ which has these goals as effects. It then recursively searches a plan for the preconditions of these actions. If it reaches the initial state, it succeeded and returns the partially ordered plan.

Note that in the original GraphPlan algorithm, *mutual exclusions* play an important role. Two actions are *mutually exclusive* if no valid plan can possibly contain both. Similarly, two facts are mutually exclusive if no valid plan can possibly make both facts hold. When searching for actions that achieve facts at time t, the planner cannot select any action that is mutually exclusive to an already selected action. If no such action exists, the planner must backtrack. However, FF uses GraphPlan on the relaxed problem. Therefore, no mutual exclusions exist, and thus, GraphPlan will never backtrack on a relaxed problem. For this reason, the procedure takes only polynomial time in the size of the relaxed task.

Pruning Techniques In addition to the described search method, FF uses multiple pruning techniques. For one, GraphPlan is extended to suggest *helpful actions* to the search algorithm. In short, helpful actions are applicable actions which provide a goal at the first time step, i.e., one of their effects is a goal in the first layer of the planning graph. Additionally, FF cuts out branches from the search tree where some goal g is achieved too early, i.e., other goals cannot be achieved without destroying g.

Completeness In general, the enforced hill-climbing search is *incomplete*. If it fails, FF starts over and searches for a valid plan with a complete heuristic search algorithm.

Extension to ADL FF was originally designed for STRIPS but has been extended to ADL. It preprocesses the ADL domain and task description by compiling the specified task down into a propositional normal form. In short, everything except the conditional effects are compiled away by the prepocessing step. The heuristic is adapted to deal with the conditional effects by extending the planning graph with an additional effects layer. Also, the adapted GraphPlan algorithm selects achieving effects instead of achieving actions. Finally, the pruning techniques are adapted to ADL.

2.5.2 Fast Downward

Fast Downward (FD) [32] is a classical PDDL planner, which supports all features of propositional PDDL2.1, and thus supports ADL. It uses a heuristic forward search and hierarchical problem decomposition. It is based on the PDDL planner FF (cf., Section 2.5.1), but uses the causal graph heuristic.

The Causal Graph Heuristic A fluent v of the domain is said to have a causal dependency on another fluent w, if there is an action with a precondition on w changing the value of v, or if there is an action which has an effect on both v and w. In the example in Figure 2.1, `holding product1` depends on `robot_position`, because the robot has to be at the same location as the workpiece in order to be able to pick it up. `product1_position` and `holding product1` depend on each other because they are both changed by the *pick_up* action. The problem is hierarchically split up in sub-problems, which contain a single variable and the variable's predecessors in the causal graph. If

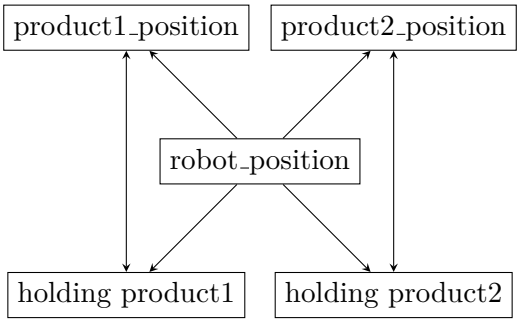


Figure 2.1: Part of the causal graph for the RCLL. In this graph, the products' positions and the robot position have been converted to multi-valued fluents.

the causal graph of a problem is cyclic, the heuristic ignores some causal dependencies, making the graph acyclic.

The causal graph of propositional domains, where fluents can only have the values *true* or *false*, is often quite complex. In order to simplify the causal graph, Fast Downward first converts the planning task to a multi-valued planning task.

The Search Fast Downward uses several search algorithms. It can use a greedy best-first search algorithm with the causal graph heuristic, splitting up the problem recursively into smaller sub-problems, starting from the goal variables. Second, it supports a *multi-heuristic best-first search*, where it combines multiple heuristics, namely the causal graph heuristic and the FF heuristics (cf., Section 2.5.1). Third, it can use a *focused iterative-broadening search* without heuristics, which uses the causal graph to reduce the number of operators it considers.

2.5.3 PDDL Planners in DBMP

We will use FF and Fast Downward as planners in DBMP. In particular, we will use Fast Downward for generating seed plans and FF for solving problems with macro-augmented domains. We use Fast Downward in a configuration that is optimized for plan length and not for planning time. Thus, Fast Downward takes longer to plan but finds shorter and better plans. Since the seed plans are better, the generated macros potentially also consist of more useful actions. On the other hand, FF typically finds a solution faster. Thus, when planning a problem with the augmented domain, we use FF to find a solution quickly. However, as the implementation is independent of the planners used, we can swap FF and Fast Downward with any PDDL planner.

We continue by introducing the RCLL as an example for a robotics domain.

2.6 RoboCup Logistics League

The RoboCup Logistics League (RCLL) [64] is an industry-oriented competition and part of RoboCup [41], an international initiative to advance intelligent robotics research

by means of robot competitions. The focus of the RCLL is a current industry trend from mass production towards customized products which requires the factory to adapt to dynamic demands and thus demands a more dynamic production and the use of cyber-physical systems [64]. In the RCLL, a team of robots has to "plan, execute, and optimize the material flow in a smart factory scenario and deliver products according to dynamic orders" [57]. Each team can use up to three robots. The goal is to produce complex products consisting of a base (red, black, or silver), zero to three colored rings (blue, green, orange, or yellow), and a cap (black or gray). In order to produce a product, each team has an exclusive set of six static Modular Production Systems (MPS). Each MPS is responsible for a specific manufacturing task: a *base station* produces bases, a *ring station* mounts colored rings on a workpiece, a *cap station* mounts the final cap on a product, and a *delivery station* is the collection point for finished products. In order to accomplish a manufacturing step, the robots have to transport the semi-finished product to the correct MPS. After all parts have been assembled, the product needs to be delivered to the delivery station. Product orders are created dynamically, the robots need to react to the dynamic orders fully autonomously, i.e., no interference by human team members is allowed.

In a game, two teams share the 12m × 6m field. The game is controlled by an *autonomous referee box* [63], a software component which communicates with the machines and each team's robots, manages the current game phase, and keeps track of the team scores. The game is split in two phases, the exploration phase and the production phase. In the *exploration phase*, each team has to explore the game field and detect and identify their machines and report them to the referee box. In the *production phase*, the referee box creates orders in a randomized fashion. An order consists of the configuration of the requested product, the number of requested products, and the time window in which the product is to be delivered. Points are assigned when a team delivers a product successfully, more complex products give more points.

The RCLL has been suggested as a benchmark domain for planning with the following characterization [57]:

Cooperative and Competitive: Robots of the same team must cooperate and each team is competing against another team on the same field.

Partially Observable: The robots can only observe parts of the relevant aspect of the world, e.g., a robot can only see parts of the field.

Non-Deterministic: A robot's action is non-deterministic as it may fail for various reasons, e.g., a failed grasping action. An action is typically not stochastic because there is no known probability distribution over action outcomes. Similarly, a robot's sensors are non-deterministic and not stochastic.

Sequential: A robot's decision influences the remainder of the game; actions cannot be seen as independent, atomic episodes.

Dynamic: While an agent is reasoning, the world may change as there are multiple, independently acting robots of different teams.

Continuous/Discrete: Time and a robot's motion and position are continuous, but a discrete representation is possible.

Known: An action's possible outcomes are known (but it may still be non-deterministic).

The RCLL is of medium complexity and allows both local-scope and global-scope planning. Current agent systems typically follow an incremental strategy [56, 57], meaning that they incrementally extend the plan during execution. Using a longer planning horizon or even a complete plan (i.e., a full plan towards a final goal) may lead to better results. Furthermore, current systems usually take a distributed approach because robot coordination is complex. A centralized approach can exploit additional efficiencies such as two robots cooperating to produce one product. Therefore, an agent system based on planning may improve the game performance and it allows the comparison of distributed and centralized approaches. Due to the dynamic properties of the domain, many different problem instances are possible. Thus, the RCLL is a suitable benchmark domain for planning. We will use it as a benchmark domain for our approach to macro planning.

2.7 Fawkes

Fawkes[1] is a component-based, open-source robot software framework [62]. It provides the following key features:

Component-Based A component based software framework consists of a number of executable units which provide services through well-defined interfaces [38]. In Fawkes, components are implemented through a plugin mechanism. In general, one plugin corresponds to one component. Every plugin is implemented as a dynamically loadable library. Therefore, plugins can be loaded and unloaded during run-time. Plugin runs in a separate POSIX threads and one plugin may consist of several threads. All plugin threads are managed by Fawkes. Fawkes threads borrow from the aspect-oriented design pattern, where certain predefined functionalities such as configuration, logging, blackboard access and camera access are provided by aspects, each aspect providing one functionality. Fawkes threads run in one of two different modes: In wait-for-wakeup mode, the thread blocks until it is woken up and then runs one iteration before blocking again. In continuous mode, the thread runs continuously in the background until it exits or is terminated. Usually, the continuous mode is used for threads which are blocked most of the time, such as a camera access thread, or threads with a high frequency loop, such as a motor controller thread. Fawkes provides a default implementation for the main loop which represents a *sense-think-act* cycle which is divided into stages. Threads can register for a stage and are woken up in every cycle when the main thread reaches that stage. Fawkes' multi-threading concept provides efficient synchronization between threads while exploiting the system's resources, especially on multi-core systems.

[1] `https://www.fawkesrobotics.org`

Blackboard Fawkes uses a hybrid blackboard/messaging infrastructure with well-defined interfaces. Fawkes' blackboard resembles the blackboard design pattern with the adaption that only a single writer is allowed for each data set. Thus, each component shares data with other components by writing the data to the blackboard. Other components can read from the blackboard and process this data. Additionally, reading components can send messages to the writing component, and can therefore send requests or commands to the writing component. Fawkes also provides a network infrastructure for communication, which is able to synchronize blackboard data, thereby allowing a distributed design.

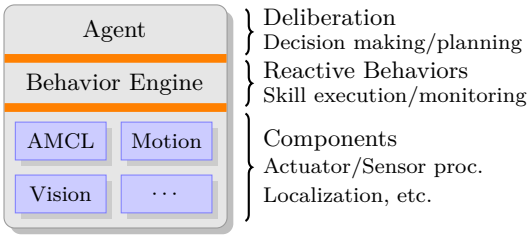


Figure 2.2: The behavior layer separation in Fawkes. Adapted from [56].

Behavior Engine Fawkes provides a three-layered hierarchy for high-level decision making as shown in Figure 2.2. In the lowest layer, multiple components provide access to the hardware and basic functionality such as driving, arm control and object detection. In the highest layer, Fawkes provides the possibility to integrate high-level agent formalisms such as CLIPS [56], Golog [44], IndiGolog [22], and PRS [60]. A Lua-based Behavior Engine [55] fills the gap between those layers as a middle layer. The Behavior Engine defines a set of skills, which are formalized as extended Hybrid State Machines. Each skill has a well-defined purpose, such as moving to a certain location, or picking up a specific object. Skills can call subskills, but a skill cannot decide to switch to a different skill. Thus, a skill only makes local decisions. Therefore, skills provide the necessary abstraction of the lowest layer and can be used by the high-level agent as primitive actions.

Fawkes in the RCLL Fawkes has also successfully been used in the RoboCup Logistics League by the team Carologistics [59]. In comparison to other approaches, Fawkes is more complex but also more easily adaptable to changing tasks, as the successes in the RCLL have shown in the recent years [61]. For the RCLL, Fawkes has been integrated with the rule-based production system CLIPS [78], which is used to implement high-level reasoning, task coordination, and execution monitoring [56]. This system can also be adapted to execute plans that were computed by a PDDL planner [46].

2.8 Databases

We use a database to store and analyze previous planning results. In the following, we introduce relevant database techniques in robotics and summarize the concept of the MapReduce programming technique.

2.8.1 Generic Robot Database

The Generic Robot Database [58] is a MongoDB database used to record data produced during run-time such as sensor outputs and the results of decision making procedures. This data can be used later, e.g., for fault analysis, performance analysis, and reinforcement learning. Collecting the data for later analysis avoids the additional cost of on-line analysis, as any analysis can be done later off-line. The data is collected from robot middleware, more specifically from the open-source robot operating system ROS [69] and the open-source robot software framework Fawkes [62]. The Generic Robot Database uses MongoDB because

1. it is document-oriented and therefore allows a direct mapping to typical message formats in robotics and supports complex queries on any document fields,
2. it is schema-less and thus allows changes to the data structure, which is common in fast changing robotics applications,
3. it supports indexing and MapReduce to allow complex queries and data processing,
4. it is a highly scalable database with replication and sharding support, and
5. it supports capped collections that automatically delete old data when a storage threshold is reached.

2.8.2 Robot Memory

Apart from storing data for off-line analysis, an agent system also needs to be able to memorize observations, gain knowledge, and share this knowledge with other agents during run-time, i.e., it needs a *robot memory* [81]. Such a robot memory can be used to represent and share the current world state, which can be used as input for a PDDL planner. Furthermore, such a robot memory can also be used as a plan database for previously computed plans. In our approach, we will use the robot memory to integrate our planner with the existing system to execute plans of the RCLL domain. However, using the robot memory as a plan database is not in the scope of this thesis. Instead, we will operate directly on MongoDB.

2.8.3 MapReduce

MapReduce [23] is a programming technique to process and generate large datasets. MapReduce takes a set of input key-value pairs and produces a set of output key-value pairs. The central idea of MapReduce is to split the operation into two steps:

1. *Map* takes one input pair and produces a set of intermediate key-value pairs.
2. *Reduce* takes an intermediate key and a set of values for that key. It merges the values to produce a set of values that is usually smaller than the intermediate set.

Listing 2.3: Counting word occurrences in a set of documents, adapted from [23].

```
1 def map(key, document):
2   for word w in document:
3     emit(w, 1)
4
5 def reduce(key, values):
6   result = 0
7   for v in values:
8     result += v
9   return result
```

Listing 2.3 shows how to count words in a collection of documents with MapReduce. In the `map` function, we iterate over one document and emit a document for each word in the document in line 3. The key of the emitted document is the word itself, the value is the count 1. In the `reduce` function, we obtain a list of documents with the same key, i.e., they share the same word. We sum over all counts of those documents to obtain the total number of occurrences for each word.

We use MapReduce to identify frequent action sequences in all plans in the database.

2.9 Cloud Computing, Virtualization and Containerization

In the following, we give an overview of cloud computing techniques, in particular over virtualization and containerization, which we can use for a scalable setup to generate seed plans and to run benchmarks. We introduce virtualization in general, present Xen as one example for virtualization, and then compare virtualization to containerization. Afterwards, we summarize the concepts of the cluster manager Kubernetes and the IT automation tool Ansible. We use both tools for managing a local cluster setup. Finally, we explain how containerization can improve the reproducibility of research.

2.9.1 Cloud Computing

The National Institute of Standards and Technology (NIST) defines cloud computing as "a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [80].

According to Zhang, Cheng, and Boutaba [80], the architecture of a cloud computing environment consists of four layers:

Hardware Layer This layer manages the physical resources, in particular the physical servers, and is usually implemented in a data center, which commonly contains thousands of servers.

Infrastructure Layer The infrastructure layer partitions the physical resources using virtualization technologies and is responsible for dynamic resource assignment.

Platform Layer The platform layer builds on top of the infrastructure layer and consists of operating systems and application frameworks.

Application Layer The application layer contains the actual application.

Zhang, Cheng, and Boutaba also distinguish three types of clouds.

Public clouds A public cloud is managed by a service provider. Resources in a public cloud can be rented on demand. Thus, a public cloud offers the most flexibility as no initial investment is necessary, and it shifts the risk of operating the cloud to the service provider.

Private clouds A private cloud is an internal cloud that is designed for exclusive use by a single organization. A private cloud offers more control over performance, reliability, and security.

Hybrid clouds A hybrid cloud combines a public cloud and a private cloud to address the limitations of each approach.

Our setup, where we use lab machines to set up a local cluster, can be considered to be a small-scale private cloud.

2.9.2 Virtualization

All modern cloud setups such as Google's App Engine and Amazon's Elastic Compute Cloud (EC2) use virtualization for their infrastructure [73]. In such a setup, the hardware is virtualized and only the virtualized hardware is exposed to the guest operating system. Virtualized hosts are managed by a hypervisor, which takes care of the host resources, runs virtual machines on the host, and provides isolation and portability. Scheepers distinguishes two types of virtualization [73]:

Full Virtualization The hardware is fully virtualized and the guest operating system cannot distinguish the virtual machine to a bare-metal machine. Thus, no modifications to the guest operating system are required.

Para Virtualization Instead of fully virtualizing the hardware of the machine, only parts of the machine are virtualized, and the guest operating system's software is adapted to deal with the virtualized hardware.

Xen One example for para-virtualization is Xen [1]. Xen is an x86 virtual machine monitor, which is capable of running Linux, Windows, and FreeBSD. Xen is para-virtualized, because it allows direct access to the host's hardware for certain operations such as system calls, but provides virtualized hardware for other operations such as interrupts. It modifies the kernel of the guest operating system to deal with para-virtualized instructions. Other than those kernel modifications, no modifications of the guest operating system are necessary. In particular, guest applications can be executed without modifications. In comparison to other virtualization techniques, Xen has a low virtualization overhead, which allows to run approximately 100 virtual machines on a single host.

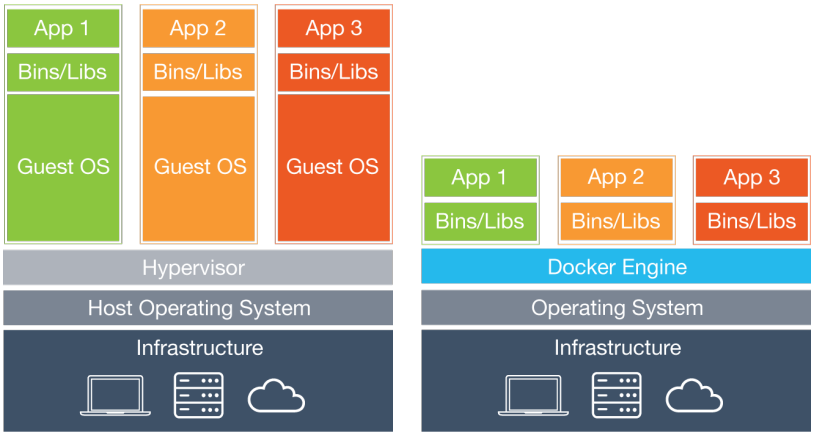
2.9.3 Containerization

A recent alternative to full virtualization and para-virtualization is container-based virtualization. The fundamental technology for container-based virtualization are Linux Containers (LXC), which provide "lightweight operating system virtualization" [73]. Instead of virtualizing hardware, LXC uses Linux kernel features such as cgroups and namespaces to isolate processes and manage resources. Each process runs with the same kernel in an isolated environment, which reduces the overhead significantly, but still separates the process from other processes [73]. In particular, there is no need for the host to run its own kernel [40]. On a traditional hypervisor, all the multiple kernels running in the guest operating systems use a large fraction of the machine's physical resources. Instead, a container only consists of the necessary binaries, libraries, and applications, it does not contain a complete operating system. Thus, a container is much leaner than a virtual machine and a single host can run hundreds to thousands of containers. A comparison of virtualization and containerization architectures can be seen in Figure 2.3.

Docker Docker [76] is the most common container software and provides tooling to conveniently create and manage LXC containers. In the following, we give a short introduction in the fundamental components of Docker. For a more complete reference, see [74].

A *Docker image* is a binary image which contains all the necessary software to run one specific application and is comparable to an image of a virtual machine for classical virtualization. Docker images are layered: While building the image, each step forms an additional layer on top of the previous layer. The lowest layer always provides basic system libraries which are used by the application. On top of this layer, the application's dependencies, its libraries, and finally its binaries are added. Since the image only contains the libraries and binaries necessary to run one particular application, a Docker image is much smaller than a typical virtual machine image. Furthermore, each layer only needs to be stored once. Thus, if multiple images share the same layers (e.g., a Python interpreter), then this layer is shared among the different images, which greatly reduces the storage requirements.

Each Docker image is created from a *Dockerfile*, which is a text script similar to a Makefile. A Dockerfile describes each step of creating a Docker image and contains



(a) Virtualization architecture (b) Docker architecture

Figure 2.3: A Comparison between the architectures of a virtual machine and a Docker container [76]. While both architectures contain a separation layer between host and guest (hypervisor and Docker Engine), a virtual machine additionally contains a complete operating system. In contrast, a Docker container only contains the relevant binaries, libraries, and applications.

commands such as `COPY`, which copies a local file from the host into the image, and `RUN`, which runs an arbitrary command within the image. Listing 2.4 is an example for a Dockerfile which installs the planners used in this thesis into a base Fedora image and then copies a local worker script into the container.

Note that the result of building a Docker image from a Dockerfile does not always result in the same image. This is because the result of running commands in the Dockerfile may be different. As an example, if Fast Downward was updated in the repositories, then the resulting image will contain the newer version. However, once an image has been built, it will always contain the same files and thus the same versions of the software.

Docker images are stored in *registries*. There are two types of registries: public registries, where everyone can access the stored images, and private registries, which contain images that are not intended for public use. Docker Hub[2] is the biggest public registry and provides images for many common software applications.

A *Docker container* is a running instance of a Docker image. When starting a container, an additional layer is created that contains any file modifications that happen during runtime. Multiple containers can be created from one image, each of those containers runs independently; any modifications are only applied on the additional layer and are not propagated to the other containers running the same image. It is common practice to keep a Docker container *stateless* so it can easily be destroyed and re-created. To keep a container stateless, its permanent data must be stored separately. For this purpose, Docker provides *data volumes*, which are virtual file system devices that are mounted into the container during initialization. Data volumes can be shared between

[2] `https://hub.docker.com`

Listing 2.4: A Dockerfile for a cluster worker. This installs the four planners FF, Fast Downward, Macro-FF and Marvin from a *Fedora Cool Other Package Repo* (COPR) and then copies the local `worker.py` file into the Docker image.

```
FROM fedora:24
MAINTAINER Till Hofmann <hofmann@kbsg.rwth-aachen.de>
RUN dnf -y --refresh install "dnf-command(copr)" && \
    dnf -y copr enable thofmann/planner && \
    dnf -y --refresh install \
      fast-downward \
      fast-forward \
      macroff \
      marvin \
    && \
    dnf clean all
COPY worker.py /usr/bin/
```

containers and persist even if the container is destroyed. Alternatively, data can be stored in a database.

2.9.4 Cluster Management with Kubernetes

Kubernetes [77] is an "open-source cluster manager for Docker containers" [3]. Kubernetes allows to define high-level tasks, services, and jobs which abstract from the application containers and decouple the application from the system on which they run. It does so by managing running containers, automatically restarting failed containers, distributing new containers in the cluster to balance the load, and by providing virtual networking, data management and management of secrets such as passwords and certificates for the cluster. As a result, the container becomes independent of the particular host it runs on. The developer does not start containers manually, but instead defines services that specify the containers to run, which data volumes and secrets these containers can access, and whether the service should be scaled. Containers are grouped into *pods*, which contain several application containers that are tightly coupled.

Additionally, Kubernetes allows the definition of *jobs*. In the most general term, a job creates a number of pods and guarantees that they successfully terminate. Jobs are typically used if there are a number of one-time tasks that need to be done without a particular order. Kubernetes automatically manages the job queue and allocates resources accordingly. The job definition includes a resource specification. Kubernetes only schedules the creation of a pod if the required resources are available. Thus, a pod is always guaranteed to have the same resources available. If those resources are exceeded, the pod is automatically terminated. Therefore, Kubernetes jobs offer an excellent platform for reproducible benchmarks of planning tasks: Each task has the same CPU share and

amount of memory available. If the task takes too long or uses too much memory, it is automatically killed. Listing 2.5 shows the Kubernetes job definition which we use to start planning tasks.

Listing 2.5: A Kubernetes job definition for a planning task. Starting in line 12, the container of this job is specified: The container runs the image `morxa/planner` and starts the script `worker.py` with the arguments `blocksworld` and `p1`.

```
1  apiVersion: batch/v1
2  kind: Job
3  metadata:
4    name: planner-ff-blocksworld-p1
5    labels:
6      jobgroup: planning-tasks
7  spec:
8    template:
9      metadata:
10       name: planner-worker
11     spec:
12       containers:
13       - name: planner-worker
14         image: morxa/planner
15         command: ["worker.py"]
16         args: ["blocksworld", "p1"]
17         resources:
18           requests:
19             cpu: "1"
20             memory: "4G"
21           limits:
22             cpu: "1"
23             memory: "5G"
24       restartPolicy: OnFailure
```

2.9.5 Using Ansible for IT Automation

Ansible[3] is an open-source IT automation tool aims to replace existing configuration management systems, deployment systems, orchestration projects, and provisioning tools [33] while keeping it as simple as possible. It is capable of doing the following tasks:

Configuration management Configuration management is the task to enforce some kind of state description for a number of servers, e.g., installing packages, making sure that configuration files contain expected values, and starting and stopping services.

[3] `https://www.ansible.com/`

Deployment Deployment is the process of "taking software that was written in-house, generating binaries or static assets (if necessary), copying the required files to the server(s), and then starting up the services" [33].

Orchestration In a typical environment with multiple servers, actions have to be taken in a specific order, e.g., the database has to be started before bringing up the web server.

Provisioning In the context of public clouds (cf., Section 2.9.1), provisioning is the task of creating new instances and managing existing instances of virtual machines.

Ansible uses a domain-specific language (DSL) based on YAML [2]. Ansible aims to be *idempotent*. In the context of configuration management tools, *idempotent* means that running the same task multiple times results in the same state as running a task a single time, i.e., if the system is already in the desired state, running the task will not cause any changes to the system.

In Ansible, a cluster configuration is typically split up in several *playbooks*. A *playbook* is a configuration management script that usually executes one task such as setting up a Kubernetes cluster. Each playbook may assign multiple *roles* to hosts. As an example, the Kubernetes playbook assigns the roles `kubernetes-master` and `kubernetes-node` to the hosts in the cluster. Each role definition consists of multiple *tasks* that need to be run in order to set up the role, e.g., one task is installing Kubernetes on the machine. Listing 2.6 shows an example for a simplified `kubernetes-master` role.

Listing 2.6: An Ansible script that installs, configures, and starts Kubernetes.

```
1  - name: install kubernetes
2    dnf: name=kubernetes state=latest
3  - name: add kubernetes config
4    template:
5      src: etc-kubernetes-config.j2
6      dest: /etc/kubernetes/config
7      backup: yes
8  - name: start kube-apiserver
9    service:
10     name: kube-apiserver
11     enabled: yes
12     state: started
```

We will use Ansible to set up Kubernetes, the plan database, and all other components required to run DBMP.

2.9.6 Using Containers for Reproducible Research

In addition to efficient virtualization and easy scalability, containers can also improve the reproducibility of research, which is often a difficult task [5]. Boettiger says that "crucial scientific processes such as replicating the results, extending the approach or testing the conclusions in other contexts, or even merely installing the software used by the original researchers can become immensely time-consuming if not impossible" [5]. He identifies three major technical challenges:

Dependency hell Installing the software dependencies of one particular application is often difficult or even impossible. Different versions of the same software may produce different results.

Imprecise documentation Documentation how to install and run software often lacks the necessary quality and hinder reproducibility by other researchers.

Code rot Dependencies receive software updates which may break the application or change the results.

Containerization offers a solution to all these challenges:

1. Dependency hell: In a Docker image, all libraries and binaries are already installed such that the software can be run out-of-the-box.
2. Imprecise documentation: Dockerfiles make the paradigm shift from explaining each step to providing a script that automatically creates the image. This shift is part of a "recently emphasized philosophy" [5] known as *Development* and Systems *Operations* (DevOps).
3. Code rot: A Docker image is a binary image which can be easily stored and archived for later re-use. Docker also offers image versioning and a central platform for sharing images (cf., Section 2.9.3).

By using containerized software and Ansible for IT automation, we aim to simplify the setup and usage of DBMP, so other researchers can easily use our planning software and verify the results.

3 Related Work

We describe related macro planning approaches and their differences to DBMP.

3.1 STRIPS with Generalized Plans

The planner STRIPS has been extended to use macros in the form of *generalized plans* [25]. A generalized plan is a (partial) solution to a previous planning problem, where constants are substituted by parameters and the resulting parameterized plan is used as a macro operator for further planning problems.

To compute a generalized plan, the proposed algorithm first computes for each of the plan's operator all the effects in the add list that survive the subsequent operators. In order to do this, it utilizes a *triangle table* as shown in Table 3.1. The first column contains in the i-th row the precondition of the i-th operator. All other columns contain the atoms from the operator's add list (A_i), and $A_{i/j}$ denotes the atoms from i-th add list that are not destroyed by operator j. Thus, the last row contains the effects of the macro operator and the first row contains the macro's preconditions.

Starting with the triangle table for a particular plan, the algorithm now generalizes by replacing constants with parameters in the triangle table. While doing so, the validity of the resulting macro has to be retained. For the details of this procedure, see [25].

As the algorithm is an extension to STRIPS, it only works on STRIPS domains. Furthermore, it does not store the resulting operators as regular STRIPS operator but as a special `MACROP`. Thus, macros are dependent on the planner and cannot be used by other planners. Finally, each macro is generated from a single plan, and therefore the general applicability of the macro is unknown. In contrast, our approach is independent of the planner and generates macro candidates by analyzing a large database of previous plans. Thus, the resulting macros are generally applicable.

Preconditions	OP 1	OP 2	OP 3	OP 4
PC_1				
PC_2	A_1			
PC_3	$A_{1/2}$	A_2		
PC_4	$A_{1/2,3}$	$A_{2/3}$	A_3	
PC_5	$A_{1/2,3,4}$	$A_{2/3,4}$	$A_{3/4}$	A_4

Table 3.1: A triangle table, adapted from [25].

3.2 Reflect

The planner REFLECT [21] is another macro planner that operates on STRIPS-like domains. In contrast to the macro extension of STRIPS, REFLECT does not generate macros from previous planning results (`MACROPs`), but instead generates macro operations (`BIGOPs`) by analyzing the domain during the preprocessing phase. Thus, a `BIGOP` uses intrinsic properties of the domain rather than previous planning results and is therefore independent of particular problem instances. In REFLECT, macros are generated as follows: First, REFLECT computes all pairs of operators that can be applied successively. Then, any pair which does not have a common variable is removed from the macro set. Finally, any macro which would be a no-op is pruned. The remaining operator pairs are used as macro operators.

In contrast to REFLECT, our approach generates macros from previous plans. While REFLECT generates macros during preprocessing, our approach generates macros off-line. REFLECT is limited to a STRIPS-like language and does not support ADL.

3.3 Macro-FF

The PDDL planner MACRO-FF [7, 6] uses two different approaches to macro planning: *Component Abstraction-Enhanced Domains* (CA-ED) and a *Solution-Enhanced Planner* (SOL-EP).

With CA-ED, the original domain is augmented with additional actions generated from the macros, such that the planner can treat the macro as if it were a primitive action. This has the advantage that the planner can be easily substituted as no adaptions to the planner are required. Macro actions are generated statically during the preprocessing phase by using *component abstraction*, a "technique that exploits permanent relationships between two low-level features of a problem" [7]. Macro candidates are filtered heuristically to generate only macro actions that are likely to be useful. CA-ED macro planning is restricted to STRIPS domains.

The second approach SOL-EP allows full ADL domains, but the planner needs to be adapted. A macro is represented as a sequence of actions and a mapping of the macro's variables. The preconditions and effects of a macro are not explicitly computed. Instead, the planner checks if the macro is applicable during search. Macros are extracted from solutions of training problems of the same domain and are limited to action sequences of length 2. There are two reasons why SOL-EP does not represent macros as normal actions: For one, there is "no straight-forward way to generate a macro's formulas" [7] for its preconditions and effects for full ADL macros. Second, the generated macros tend to have a high number of parameters. Since most planners generate all possible instantiations of an action during search, using macros with many parameters will decrease the performance of the planner.

The performance gain with macros can be described with two improvements: the *embedding improvement* and the *evaluation improvement* [7]. The embedding improvement is exploited if macros augment the search space by adding successor states which

otherwise wouldn't be reachable with one step from a particular state. The evaluation improvement is exploited if macros improve the heuristic evaluation of a state. As an example, this is used in FF if macros are also used for the relaxed problem, where the delete effects of all actions are ignored.

In the CA-ED approach, both improvements are used because macros are equivalent to normal actions in the augmented domain. In the SOL-EP approach, the planner needs to be adapted for each improvement separately. In Macro-FF's SOL-EP, only the embedding improvement is implemented.

Our approach differs from Macro-FF's SOL-EP in the way that it represents a macro as regular PDDL action, and thereby removes the need to adapt the underlying planner. Furthermore, we support macros of length greater than 2. Similar to SOL-EP but different to CA-ED, our macros are generated from previous solutions. In contrast to SOL-EP, we do not use special training problems for that purpose, but instead re-use results from previous planner calls. Our approach exploits both the embedding improvement and the evaluation improvement. We approach the identified problems of full ADL macros by (1) defining regression on ADL action sequences to compute a macro's precondition and effect chaining of sequences of effects to define a macro's effect formula, (2) coalescing parameters that are commonly assigned to the same value.

3.4 Marvin

The PDDL planner Marvin [19, 18] is another macro planner based on FF. Marvin supports ADL domains with some restrictions, e.g., it does not support disjunctive goals. It is designed based on the observation that escaping plateaus in the search space is often the most time consuming part during planning. A *plateau* occurs if the heuristic value of all successor states is the same as or worse than the heuristic value of the current state. Plateaus are difficult for planners because the search degenerates to blind search and it often takes a long time to find an action sequence that escapes the plateau. Marvin is based on the observation that for a particular domain, it is often the same action sequence that escapes the plateau. Thus, remembering and re-using such action sequences can lead to significant performance gains. Marvin only applies macros if a plateau is encountered. Macros are generated online and only from action sequences that escape plateaus. Furthermore, macros are generated on a per-problem basis and are not re-used for other problems.

In contrast to Marvin, in our approach macros are always used and are not limited to plateaus. Furthermore, macros are generated off-line and can be re-used for new problem instances. Finally, in our approach, no adaption of the planner is needed. In Marvin, macros are part of the search strategy and thus are specific to the planner.

3.5 Wizard

Wizard is a genetic algorithm to learn macros [53, 54] that supports STRIPS. It generates macros off-line from solutions of less complex problem instances, so-called seeding

problems. A macro's precondition and effects are computed by regression. From an initial macro, it uses a genetic algorithm to improve the macro by prepending and appending actions, removing actions from the beginning and end, and splitting the macro into two separate macros. The resulting macros are then evaluated by solving ranking problems with the domain augmented by the macro. The fitness function of the genetic algorithm takes into account three measurements:

Cover the percentage of ranking problems solved when the macro is used;

Score the weighted mean time gain/loss over all ranking problems;

Point the percentage of ranking problems solved faster with the augmented domain than with the original domain.

Wizard computes macros based on small seeding problems and improving them with a genetic algorithm. In contrast, our approach generates macros from previous solutions of problems with the same complexity. Thus, our approach will be able to generate macros that are only helpful in more complex problem instances. Furthermore, instead of selecting macros iteratively with a genetic algorithm, our macros are selected based on an evaluation function in a non-iterative process. Finally, in contrast to Wizard, DBMP supports full ADL domains.

3.6 The Duet Planner

The Duet Planner [30] combines domain-independent heuristics with domain-specific knowledge. It does so by combining the PDDL planner LPG [29] with the HTN planner SHOP2 [52]. LPG is a planner using domain-independent heuristics, while SHOP2 allows to exploit domain-specific knowledge by using HTNs. Duet splits the problem into sub-gals and passes each sub-goal to both planners subsequently until all sub-goals are reached.

Compared to our approach, the fundamental idea is different: While macro planning allows to infer domain-specific knowledge from previous planning results, Duet is designed such that an expert can add domain-specific knowledge to the domain. Thus, macros can be computed automatically, while the HTNs of Duet need to be specified by the domain designer. Additionally, Duet is based on LPG and cannot be used with other PDDL planners, while our approach allows to substitute the underlying planner.

3.7 MUM

MUM [15, 13] is a PDDL planner that learns macros from the solutions of previous problems. MUM only supports the STRIPS fragment of PDDL. Similar to Duet, it first solves less complex training problems to generate macro candidates. It uses the concept of *outer entanglements* [14] to rank and prune macros. Outer entanglements are dependencies between operators and initial or goal predicates. As an example from the Blocksworld domain, a block is only unstacked if it is stacked on another block initially

(assuming there are no space restrictions). This is called an outer entanglement by init. Similarly, a block is only stacked onto another block if this is the case in the goal, which is called an outer entanglement by goal. If such an outer entanglement exists, any instances on the operator that do not fulfill the condition can be immediately pruned. Thus, outer entanglements allow to reduce the search space. In MUM, outer entanglements are used to rank macros: Macros with outer entanglements by init and goal are in top rank, macros with one entanglement are in middle rank, macros without entanglements are in low rank. Additionally, outer entanglements are used during planning; any macro instances not fulfilling the entanglement condition are pruned.

In addition to entanglements, MUM introduces independent actions, which are actions that can be swapped in a plan without making the plan invalid. Independent actions are considered during macro generation; a macro may also use non-consecutive actions if the actions in-between are independent.

In contrast to MUM, we do not make use of independent actions or outer entanglements. In MUM, the planner is modified to consider outer entanglements of macros; our approach is independent of the underlying planner. MUM only uses simpler training plans and uses at most six training plans; in our approach, we consider training plans of the same complexity and we use a high number of training plans. MUM only supports macros of length 2 while we also consider longer macros. Finally, MUM only supports STRIPS while DBMP also supports the ADL fragment of PDDL.

4 Approach

The goal of this thesis is to improve planning performance by automatically generating macros which can be used during planning and which speed up planning. The general concept of DBMP is to use previous planning results to identify macro actions and to augment the original domain with those macro actions.

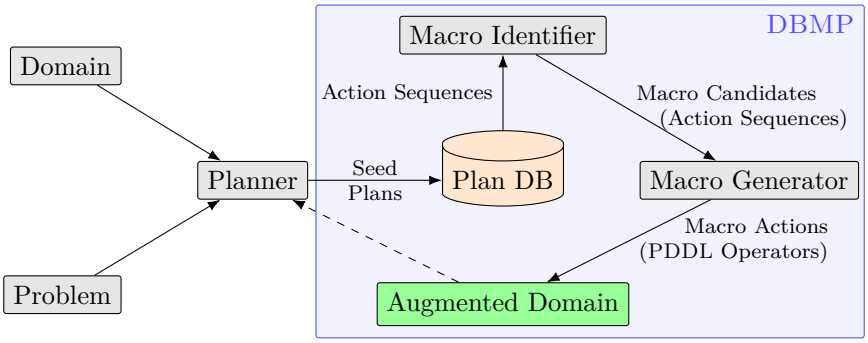


Figure 4.1: DBMP Architecture overview, adapted from [36].

Figure 4.1 shows the required steps to generate a domain augmented with a macro action from a set of problems:

1. *Collecting Seed Plans*: The seed plans are plans which are analyzed in order to identify frequent action sequences. The seed plans are normal plans of the same domain (in particular, we do not use simplified seed problems). They can be collected on-line during the execution of the robot, or they can be computed from a number of generated problems. All plans are collected in the *plan database*.

2. *Identification*: From the seed plans, frequent action sequences including their parameter assignment are computed. Identification uses the *MapReduce* programming paradigm and does not require any insights into the particular domain or planning in general.

3. *Generation*: From the previously identified action sequences, we generate new PDDL actions that can be added to the original domain. As a PDDL action is described by a single precondition and effect, we need to combine all preconditions of the actions in the sequence by *regressing the precondition* of each action to the beginning of the sequence. Additionally, we need to *chain the effects* and compute effect collisions of all effects in the action sequence. The generated macro actions are then added to the original domain to obtain the *augmented domain*.

4. *Macro Planning* and *Macro Expansion*: The augmented domain can be used for planning new problems. Since macros are represented as normal actions, the planner does not need to be modified, but can be used as-is to solve a planning task. However, in order to execute the resulting plan, the plan needs to be translated back to the original domain, i.e., all macro actions in the plan have to be replaced by the respective action sequence.

The first three steps can be executed off-line, i.e., we obtain an augmented domain that can be used by the planner without any identification or generation steps during the execution. The expansion step is done on-line because it is problem-specific and needs to be executed for each problem after a plan was computed.

4.1 DBMP Database

The database is a central component of DBMP. Every step of the process shown in Figure 4.1 gets the input from the database and saves the results in the database. We store planning results for the original domain in the database, save identified action sequences in the database, and store generated macros and augmented domains in the database. We decided to use MongoDB for the plan database for the following reasons:

- MongoDB natively supports MapReduce, which we use for macro identification.
- It is document-oriented and schema-less, which allows us to store PDDL domains, problems, actions, and formulas in a structured way.
- It supports complex queries on arbitrary document fields.
- It is highly scalable.
- It has been integrated with Fawkes and is used for robot memory (cf. Section 2.8), thus integrating our approach into the existing framework will be easier.

Database Structure A MongoDB database consists of multiple *collections*, where each collection stores documents of similar structure. In our database, we have the following collections:

domains: All original and augmented domains are stored in this collection. The domain itself is stored in raw text format accompanied with some meta information such as included macros and evaluation scores.

problems: All problems of all domains are stored in this collection. The domain is referenced by name so a problem may belong to multiple augmented domains that extend the same domain. The problem is stored in raw text format.

solutions: All plans generated by the planners are stored in this collection. Solutions are stored both in raw text format and structured format. To obtain the structured

format, a planner-specific solution parser translates the raw solution into a list of actions. Additionally, each solution has a boolean field `use_for_macros`, which specifies whether the solution can be used as a seed plan for macro identification.

action sequences: For each domain name in the `domains` collection, there is a separate collection with the name `action_sequences_<domain_name>`. In this collection, the result of the MapReduce operation to identify action sequences is stored.

macros: This collection stores the results of the macro generation step together with the evaluation scores of each macro.

Listing 4.1: A database entry for an augmented domain.

```
1  {
2    "_id" : ObjectId("58ade052ee1e5bbfce91825f"),
3    "macros" : [
4      ObjectId("58ade052ee1e5bbfce918245")
5    ],
6    "augmented" : true,
7    "evaluation" : {
8      "complementarity_weighted_fp_evaluator_50_50" : 45650,
9      "complementarity_weighted_fp_evaluator_0_100" : 200,
10     "complementarity_weighted_fp_evaluator_100_0" : 91100,
11   },
12   "name" : "cleanup",
13   "base_domain" : ObjectId("58989933caa4414d3ee08a49"),
14   "raw" : "(define (domain cleanup_with_kif) ...)"
15 }
```

Listing 4.1 shows an example for a database entry for an augmented domain. This domain is augmented with one macro. A reference to the macro is kept in the document in line 4. Additionally, the document contains a dictionary of evaluation scores. This domain was evaluated by the `complementarity_weighted_fp_evaluator_50_50` (CFP) evaluator with a score of 45650. See Section 4.4 for more information on evaluators.

Listing 4.2 shows a solution document for a problem that was successfully solved by the planner FF. Storing solutions in such a structured way allows to analyze the planning results with a database query. As an example,the following query finds all plans that contain the `goto` operator:

```
db.solutions.find({"actions.operator": "goto"})
```

We will use the structured planning results in Section 4.2 to find common action sequences in the database.

Listing 4.2: A solution document for a problem from the Cleanup domain.

```
{
  "_id" : ObjectId("5898af40779a150001c38276"),
  "resources" : [ 0.015849, 0.004772, 11544, ...],
  "planner" : "ff",
  "problem" : ObjectId("58989935caa4414d3ee08afe"),
  "start_time" : ISODate("2017-02-06T17:15:44.061Z"),
  "use_for_macros" : true,
  "domain" : ObjectId("58989933caa4414d3ee08a49"),
  "end_time" : ISODate("2017-02-06T17:15:44.095Z"),
  "raw" : "(GOTO LIVINGROOM_TABLE)\n(ALIGN_TO LIVINGROOM_TABLE) ...",
"actions" : [
    {
      "operator" : "goto",
      "parameters" : [
        "livingroom_table"
      ]
    },
    {
      "operator" : "align_to",
      "parameters" : [
        "livingroom_table"
      ]
    },
    ...
  ]
}
```

4.2 Macro Identification

The goal of *macro identification* is to identify common action sequences in the plan database, which are then used to generate macros. In the plan database, we store plans which are simply sequences of grounded actions. From these sequences, we need to identify frequent sub-sequences and compute the respective (ungrounded) action sequences. As an example, consider a domain with the following actions:

- `goto(?to - location)`: Go to location `?to`.
- `align-to(?align-loc - location)`: Align to location `?align-loc`.
- `pick-up(?c - cup)`: Pick up cup `?c`.
- `put(?c - cup ?put-loc - location)`: Put cup `?c` at location `?put-loc`.

The database may contain the following three plans:

Plan 1 `<goto(table),align-to(table),pick-up(cup)>`

Plan 2 `<goto(counter),align-to(counter),put(cup,counter)>`

Plan 3 `<goto(kitchen),align-to(dishwasher),put(cup,dishwasher)>`

From those plans, we can identify the sequences with respective parameter assignment and occurrence count shown in Table 4.1.

Action Sequence	Parameter		Count
	Assignment	Enumeration	
`<goto, align-to>`	`?to` → `?l1` `?align-loc` → `?l1`	[1], [1]	2
`<goto, align-to>`	`?to` → `?l1` `?align-loc` → `?l2`	[1], [2]	3

`<align-to,put>`	`?align-loc` → `?l1`
	`?c` → `?c1`
	`?put-loc` → `?l1`
		[1], [2, 1]	2

`<align-to,put>`	`?align-loc` → `?l1`
	`?c` → `?c1`
	`?put-loc` → `?l2`
		[1], [2, 3]	2

`<goto, align-to, put>`	`?to-loc` → `?l1`
	`?align-loc` → `?l1`
	`?c` → `?c1`
	`?put-loc` → `?l1`
		[1], [1], [2, 1]	1

`<goto, align-to, put>`	`?to-loc` → `?l1`
	`?align-loc` → `?l2`
	`?c` → `?c1`
	`?put-loc` → `?l2`
		[1], [2], [3, 2]	2

Table 4.1: An example for identifying action sequences in a plan database. The parameter assignment defines for each parameter in the action sequence to what parameter it should be assigned to. The parameter enumeration enumerates the assignment, i.e., if a parameter is assigned to the same value, it will have the same enumeration. Depending on the assignment, the occurrence count may differ. As an example, consider the first two sequences: If `?to` and `?align-loc` are assigned to the same parameter, then the sequence `<goto, align-to>` occurs twice. If they are assigned to different parameters, the sequence occurs three times, because in Plan 3, the parameters differ. Note that the plan database contains more sequences, this table is a sample selection.

The example in Table 4.1 already shows that a plan database may potentially contain a very large number of action sequences, especially if we take different parameter assignments into account. In order to deal with this, we use the MapReduce paradigm, as introduced in Section 2.8.3. For MapReduce, we need to define two operators, the *Map* operator and the *Reduce* operator:

Map As its input, the map operator gets a single plan consisting of a list of grounded actions. From that plan, it computes all occurring action sub-sequences including all possible parameter assignments. For each action sub-sequence, Map *emits* a document that contains all parameter assignments for that sub-sequence including the respective occurrence counts. The pseudo-code for *Map* is shown in Listing 4.3.

Reduce The input for the Reduce operator is a list of documents from the previous step, where the list contains all documents with a specific action sequence. The Reduce operator sums up all occurrence counts to obtain an occurrence count for each action sequence and parameter assignment for the whole plan database. The pseudo-code for *Reduce* is shown in Listing 4.4.

Listing 4.3: The *Map* operator to find all sub-sequences in a plan.

```
1  input:
2    plan: a list of grounded actions with length l
3    maxLength: a maximum sequence length
4  output: a list of action sequences with parameter enumerations
5  begin
6    enumerations := {}
7    for seqStart := 0 to l
8      for i := 1 to maxLength
9        currentEnumerations := {}
10       actions := plan[seqStart:seqStart+i]
11       params := getParameters(actions)
12       foreach enumeration in getPossibleEnumerations(length(params))
13         if isValidEnumeration(actions, enumeration) do
14           currentEnumerations.add(enumeration)
15         end
16       end
17       enumerations.add((actions, currentEnumerations))
18     end
19   end
20   return enumerations
21 end
```

Listing 4.4: The *Reduce* operator to sum up parameter enumerations.

```
1  input:
2    enumerations: a list of parameter enumerations
3  output:
4    a dictionary of pairs <enumeration, occurrence count>
5  begin
6    result := {}
7    foreach enumeration in enumerations:
8      result[enumeration] := result[enumeration] + 1
9    end
10 end
```

4.3 Macro Generation

The purpose of *macro generation* is to compute a PDDL action representation for a previously identified macro, i.e., take a sequence of PDDL actions and compute the macro action's precondition and effects such that the macro can be used as a normal PDDL action in the augmented domain. For each such action sequence s, macro generation consists of the following steps:

1. Parse the PDDL domain.
2. Compute the precondition of the action sequence s using regression.
3. Compute the effects of the action sequence s using effect chaining.
4. Compute the macro's parameters.
5. Generate a PDDL action description for the generated macro action.

We describe each step in detail in the following sections.

4.3.1 PDDL Parser

In order to parse a PDDL domain, we formulated (strict) PDDL as a definite clause grammar (cf., Section 2.1.3). Definite clause grammars can be directly implemented in Prolog. The parser gets a PDDL string as input and returns the PDDL domain in a structured form. As an example, the rule shown in Listing 4.5 parses a string that describes a PDDL action into the action's *name*, *parameters*, *precondition*, and *effects*.

As another example, the rules shown in Listing 4.6 describe an action effect. For each kind of effect (i.e., atomic effect, negated atomic effect, conjunction, conditional effect, and quantified effect), there is a separate rule. Body parts of the form {...} as in line 5 describe additional constraints formulated as Prolog predicate that must be satisfied so the rule can be applied.

Listing 4.5: A DCG rule for parsing a PDDL action.

```
action(Name, Parameters, Precondition, Effects) -->
  ["(",":action"], [Name],
  [":parameters"], ["("], typed_list(Parameters), [")"],
  [":precondition"], ["("], goal_description(Precondition), [")"],
  [":effect"], effect(Effect),
  [")"].
```

Listing 4.6: A DCG rule for parsing a PDDL action effect.

```
effect(Effect) --> atomic_formula(Effect).
effect(Effect) --> ["(", "not"], atomic_formula(Effect), [")"].
effect(Effect) -->
  ["(", "and"], effect_list(Effects), [")"],
  { Effect =.. [and|Effects] }.
effect(when(Cond,Effect)) -->
  ["(", "when"], goal_description(Cond), effect(Effect), [")"].
effect(all(VarList,Effect)) -->
  ["(", "forall"],
  ["("], typed_list(VarList), [")"],
  effect(Effect), [")"].
```

The result of the PDDL parser is used for computing the macro's parameters, precondition, and effects. Additionally, the definite clause grammar is also used to generate a PDDL representation of the resulting macro action.

4.3.2 Precondition

To compute the precondition of a macro, we need to combine all the preconditions of the actions in the given sequence. As one action in the sequence may affect the precondition of a subsequent action, we need to regress the precondition of each action over the effects of all preceding actions. As an example, consider the action sequence `<goto(?l),align-to(?l)>`. The precondition of `align-to(?l)` is `at(?l)`. However, the action `goto(?l)` has as its effect `at(?l)`, and thus the precondition of `align-to(?l)` can be regressed to `true`.

More generally, we require the following: If the precondition of the macro is satisfied, then the precondition of each action in the corresponding action sequence must be satisfied after applying the effects of all preceding actions. We first define the notion of an *executable action sequence*, which is an action sequence whose actions can be applied consecutively.

Definition 4.3.1 (Executable action sequence)**.**
Given a sequence of grounded actions $\sigma = \langle a_1, \ldots, a_n \rangle$ with preconditions $\pi_1, \ldots, \pi_n$. We say σ is *executable in world state* w if the following holds:

$$w \models \pi_1 \wedge [a_1] (\pi_2 \wedge [a_2] (\pi_3 \wedge \ldots \wedge [a_{n-1}] (\pi_n) \ldots))$$

We can then formalize the requirement above by defining *macro representations of preconditions*:

Definition 4.3.2 (Macro representation of preconditions)**.**
Let m be an ADL operator, π_m the precondition of m, and $\sigma = \langle a_1, \ldots, a_n \rangle$ a sequence of ADL operators with free variables $x_1 \ldots, x_k$. We say π_m is a *macro representation of the preconditions of* σ if π_m contains no free variables but $x_1, \ldots, x_k$ and for all world states w and ground terms $r_1, \ldots, r_k$, the following holds:

$$\text{If } w \models \pi_m(r_1 \ldots, r_n), \text{ then } \sigma(r_1, \ldots, r_n) \text{ is executable in } w.$$

Note that this does not require that the converse also holds, i.e., even if the corresponding action sequence may be executable, the macro's precondition does not need to be satisfied. There are two reasons for this: First, we always only add macros to the domain, but we never remove actions. Thus, even if the macro's precondition is never satisfied, completeness is not impaired. If a plan exists for a given problem in the original domain, then this plan is also valid in the augmented domain. Second, not requiring the converse allows us to simplify the precondition of the generated macro. In particular, we can reduce the number of disjunctions and implications in the precondition, as these typically impair planner performance. As an example, let action a_1 be an action with precondition $\pi_1 = \top$ and effect $e_1 = p(o_1)$ and let o_1 have type *object*. Let the precondition π_2 of the subsequent action a_2 be $\pi_2 = \forall o{:}obj\, [p(o)]$. In this case, the sequence $\langle a_1, a_2 \rangle$ is executable in world state w if the following holds:

$$w \models \forall o{:}object\, [o = o_1 \vee p(o)]$$

Thus, the precondition of the macro $\langle a_1, a_2 \rangle$ should be $\forall o{:}object\, [o = o_1 \vee p(o)]$. However, we could also simplify the precondition to $\forall o{:}object\, [p(o)]$ in order to avoid an additional disjunction in the precondition.

To compute the precondition of an ADL operator sequence, we define the operator $\mathcal{R}_1$, where $\mathcal{R}_1(\alpha, e)$ is the regression of precondition formula α over effect formula e.

Definition 4.3.3 (Regression $\mathcal{R}_1$ over a single effect)**.**
Let α, α_1, α_2 be precondition formulas, ϕ an atomic formula, e an effect formula, $\vec{r}$, $\vec{s}$, $\vec{t}$ ground terms and $F(\vec{s})$, $F(\vec{t})$, $G(\vec{r})$ atomic formulas with distinct fluent predicate names F and G. The operator $\mathcal{R}_1(\alpha, e)$ is defined inductively over the precondition formula α and the effect formula e.

$$\mathcal{R}_1(F(\vec{t}), F(\vec{t})) = \top \tag{4.1}$$
$$\mathcal{R}_1(F(\vec{t}), G(\vec{r})) = F(\vec{t}) \tag{4.2}$$
$$\mathcal{R}_1(F(\vec{s}), F(\vec{t})) = \vec{s} \neq \vec{t} \wedge F(\vec{s}) \vee \vec{s} = \vec{t} \tag{4.3}$$
$$\mathcal{R}_1(\neg\alpha, e) = \neg\mathcal{R}_1(\alpha, e) \tag{4.4}$$
$$\mathcal{R}_1(\alpha_1 \wedge \alpha_2, e) = \mathcal{R}_1(\alpha_1, e) \wedge \mathcal{R}_1(\alpha_2, e) \tag{4.5}$$
$$\mathcal{R}_1(\alpha_1 \vee \alpha_2, e) = \mathcal{R}_1(\alpha_1, e) \vee \mathcal{R}_1(\alpha_2, e) \tag{4.6}$$
$$\mathcal{R}_1(\forall\vec{x}{:}\vec{\tau}\,\alpha, e) = \forall\vec{x}{:}\vec{\tau}\,\mathcal{R}_1(\alpha|^{\vec{x}}_{\vec{v}}, e)|^{\vec{v}}_{\vec{x}} \tag{4.7}$$
where $\vec{v}$ are new variables not occurring in e or α
$$\mathcal{R}_1(\exists\vec{x}{:}\vec{\tau}\,\alpha, e) = \exists\vec{x}{:}\vec{\tau}\,\mathcal{R}_1(\alpha|^{\vec{x}}_{\vec{v}}, e)|^{\vec{v}}_{\vec{x}} \tag{4.8}$$
where $\vec{v}$ are new variables not occurring in e or α
$$\mathcal{R}_1(F(\vec{t}), \neg F(\vec{t})) = \bot \tag{4.9}$$
$$\mathcal{R}_1(F(\vec{t}), \neg G(\vec{r})) = F(\vec{t}) \tag{4.10}$$
$$\mathcal{R}_1(F(\vec{s}), \neg F(\vec{t})) = \vec{s} \neq \vec{t} \wedge F(\vec{s}) \tag{4.11}$$
$$\mathcal{R}_1(\phi, e_1 \wedge e_2) = \mathcal{R}_1(\mathcal{R}_1(\alpha, e_1), e_2) \tag{4.12}$$
$$\mathcal{R}_1(\phi, \forall x{:}\tau\, e) = \begin{cases} \bigvee_{v \in \mathit{Free}(\tau,\phi)} \mathcal{R}_1(\phi, e|^x_v) & \text{if } \mathit{Free}(\tau,\phi) \neq \emptyset \\ \phi & \text{else} \end{cases} \tag{4.13}$$
$$\mathcal{R}_1(\phi, \gamma \Rightarrow e) = \gamma \wedge \mathcal{R}_1(\phi, e) \vee \neg\gamma \wedge \phi \tag{4.14}$$

In the following, we present some examples on regressing a precondition formula on a single effect:

1. $\mathcal{R}_1(\mathit{aligned}(l), \mathit{at}(l)) \overset{(4.2)}{=} \mathit{aligned}(l)$
2. $\mathcal{R}_1(\mathit{at}(l), \mathit{at}(l)) \overset{(4.1)}{=} \top$
3. $\mathcal{R}_1(\mathit{at}(l), \neg\mathit{at}(l)) \overset{(4.9)}{=} \bot$
4. $\mathcal{R}_1(\mathit{at}(l_1), \forall x{:}\mathit{location}\, x \neq l_2 \Rightarrow \neg\mathit{at}(x))$
   $\overset{(4.13)}{=} \mathcal{R}_1(\mathit{at}(l_1), l_1 \neq l_2 \Rightarrow \neg\mathit{at}(l_1))$
   $\overset{(4.14)}{=} l_1 \neq l_2 \wedge \mathcal{R}_1(\mathit{at}(l_1), \neg\mathit{at}(l_1)) \vee l_1 = l_2 \wedge \mathit{at}(l_1)$
   $\overset{(4.3)}{=} l_1 \neq l_2 \wedge \bot \vee l_1 = l_2 \wedge \mathit{at}(l_1)$
   $= \; l_1 = l_2 \wedge \mathit{at}(l_1)$

5. $\mathcal{R}_1(\forall o{:}obj\, p(o), p(o_1))$
$\overset{(4.7)}{=} \forall o{:}obj\, \mathcal{R}_1(p(v), p(o_1))|_o^v$
$\overset{(4.3)}{=} \forall o{:}obj\ (v \neq o_1 \wedge p(v) \vee v = o_1)\,|_o^v$
$= \forall o{:}obj.\ o \neq o_1 \wedge p(o) \vee o = o_1$
$= \forall o{:}obj.\ p(o) \vee o = o_1$

6. $\mathcal{R}_1(\exists l{:}location.\ at(l) \wedge \neg aligned(l), \neg aligned(l_1))$
$\overset{(4.8)}{=} \exists l{:}location.\ \mathcal{R}_1(at(v) \wedge \neg aligned(v), \neg aligned(l_1))|_l^v$
$\overset{(4.5)}{=} \exists l{:}location\ \big[\mathcal{R}_1(at(v), \neg aligned(l_1)) \wedge \mathcal{R}_1(\neg aligned(v), aligned(l_1))\big]|_l^v$
$\overset{(4.2)}{=} \exists l{:}location\ \big[at(v) \wedge \mathcal{R}_1(\neg aligned(v), \neg aligned(l_1))\big]|_l^v$
$\overset{(4.4)}{=} \exists l{:}location\ \big[at(v) \wedge \neg\mathcal{R}_1(aligned(v), \neg aligned(l_1))\big]|_l^v$
$\overset{(4.11)}{=} \exists l{:}location\ \big[at(v) \wedge \neg(v \neq l_1 \wedge aligned(v)))\big]|_l^v$
$= \exists l{:}location\ \big[at(v) \wedge (v = l_1 \vee \neg aligned(v))\big]|_l^v$
$= \exists l{:}location.\ at(l) \wedge (l = l_1 \vee \neg aligned(l))$

Generating the precondition of a macro. Using the regression operator $\mathcal{R}_1$, we compute the precondition π_m of a macro m representing an action sequence $\sigma = \langle a_1, \ldots, a_n\rangle$ with preconditions $\pi_1, \ldots, \pi_n$ and effects $e_1 \ldots e_n$ as shown in Listing 4.7.

Listing 4.7: Generating a macro precondition.

```
1 input: int n, goal formulas π_1,...,π_n, effect formulas e_1,...,e_n
2 output: goal formula π_m
3 begin
4   π_m := π_n
5   for i := n-1 to 1
6     π_m := R_1(π_m, e_i) ∧ π_i
7   end
8   return π_m
9 end
```

Note that we apply the effects of σ in reverse order, i.e., we first apply e_n on α, then e_{n-1} on the resulting formula, and so on, until e_1 was applied. This is because e_n is the last effect of the sequence and thus may cancel previous effects. As an example, consider the effect sequence $\sigma = \langle p(a), \neg p(a)\rangle$. Clearly, when regressing $p(a)$ over σ, the result should be $\bot$, because $\neg p(a)$ directly conflicts with $p(a)$. The previous effect $p(a)$ is canceled by the subsequent effect $\neg p(a)$.

Implementation We implemented the computation of preconditions and in particular the $\mathcal{R}_1$ operator in Prolog by defining a Prolog predicate

```
regress(Effects, TypedVariables, Formula, ResFormula)
```

which is true iff `ResFormula` is the result of regressing `Formula` on `Effects`, given the names and types `TypedVariables`.

Listing 4.8: The Prolog implementation of $\mathcal{R}_1(F(\vec{s}), \neg F(\vec{t}))$.

```
regress_(
  [not(Effect)|RemainingEffects], Types, Formula, ResFormula
) :-
  % Effect is of the form Predicate(EffectArgs).
  Effect =.. [Predicate|EffectArgs],
  % Formula is of the form Predicate(FormulaArgs).
  Formula =.. [Predicate|FormulaArgs],
  % Predicate is actually a predicate and not a compound formula.
  \+ member(Predicate, [and,or,all,imply,when,not]),
  % Generate equations of the form not(EffectArg = FormulaArg)
  % for each pair of arguments in EffectArgs and FormulaArgs.
  maplist(\EffectArg^FormulaArg^(=(not(EffectArg = FormulaArg))),
    EffectArgs, FormulaArgs, Equations),
  % The resulting formula is a conjunction
  % of all equations from above.
  ResStepFormula =.. [and,Formula|Equations],
  % Continue regressing on the remaining effects.
  regress_(RemainingEffects, Types, ResStepFormula, ResFormula).
```

As an example, Listing 4.8 shows the implementation of $\mathcal{R}_1(F(\vec{s}), \neg F(\vec{t}))$. Note that the implementation slightly differs from the definition of $\mathcal{R}_1$. For one, the order of arguments is different, i.e., the first argument is a list of effects, the second argument is a list of typed variables, the third argument is the formula, and the fourth argument is the regressed formula. Second, the formula is not only regressed on a single effect, but on a list of effects, and `regress_` is called recursively. Third, the predicate shown here is called `regress_`. We also define a predicate `regress`, which calls `regress_` and simplifies the result, as shown in Listing 4.9.

Listing 4.9: The predicate `regress`, which calls `regress_` and simplifies the result.

```
regress(Effects, Types, Formula, SimplifiedRegressedFormula) :-
  once(regress_(Effects, Types, Formula, RegressedFormula)),
  simplify(RegressedFormula, SimplifiedRegressedFormula).
```

Listing 4.10: A simplified `goto` action.

```
(:action goto
  :parameters (?from ?to - location)
  :precondition
    (and
      (robot-at ?from)
      (not (robot-at ?to)))
  :effect
    (and
      (not (robot-at ?from))
      (robot-at ?to)
    )
)
```

4.3.3 Effects

To compute the effect of a macro action, we need to combine all effects in the respective action sequence. In particular, we need to compute conflicting effects and resolve these conflicts. As an example, consider the simplified `goto` action shown in Listing 4.10. When we chain two `goto` actions in the sequence `<goto(?l1,?l2)>,goto(?l2,?l3)>`, then the sub effect `(robot-at ?l2)` of the first action conflicts with the effect `(not (robot-at ?l2))` of the second action. Since the second action's effect is applied after the first action's effect, the resulting effect should be:

`(and (not (robot-at ?l1)) (not (robot-at ?l2)) (robot-at(?l3)))`

Note that in contrast to the precondition, the resulting macro effect must exactly match the effects of all actions in the sequence. Thus, we cannot omit sub effects or add more effects to the macro. Formally, we define a macro representation of an effect:

Definition 4.3.4 (Macro representation of effects)**.**
Let m be an ADL operator, e_m the effect of m, and $\sigma = \langle a_1, \ldots, a_n \rangle$ a sequence of ADL operators with free variables $\vec{v}$ of types $\vec{\tau}$. We say e_m is a *macro representation of the effects of* σ if the following holds for all atomic formulas ϕ:

$$\models \forall \vec{v}{:}\vec{\tau}.\, [m]\phi \;\equiv\; [a_n][a_{n-1}] \ldots [a_1]\phi$$

Note that we do not distinguish between a macro and an ADL operator. In fact, any ADL operator can be a macro representation of some operator sequence. Before we define effect chaining, we first define a restriction on effect formulas.

Definition 4.3.5 (Properly quantified effects)**.**
We call an effect formula e *properly quantified* if the following holds for each conditional effect sub-formula $\gamma \Rightarrow e'$ of e: If v is a free variable in γ, but not a free variable in e, then it must occur in each atomic sub-formula of e'.

In other words, we do not allow ∀-quantification only over variables in the condition of a conditional effect. As an example, consider the effect $e = \forall o{:}\tau.\, p(o) \Rightarrow q(a)$. If $p(o)$ holds for *any* o, then the effect of the action will always be $q(a)$. This is different from the effect $e' = (\forall o{:}\tau\, p(o)) \Rightarrow q(a)$. In this case, the effect $q(a)$ only follows if $p(o)$ is true *for all* objects of type τ. In the first case, the quantifier $\forall o{:}\tau$ quantifies over the conditional effect $p(o) \Rightarrow q(a)$. In the second case, the quantifier is part of the condition, and only quantifies over the formula $p(o)$. Note that, assuming there is an object of type τ, we can formulate an equivalent effect e' that is properly quantified: $e_{proper} = (\exists o{:}\tau\, p(o)) \Rightarrow q(a)$.

To compute the resulting effect of two consecutive actions, we define the chain operator $\mathcal{C}(e_1, e_2)$ that chains two effects e_1, e_2, where e_1 is before e_2.

Definition 4.3.6 (Effect chaining $\mathcal{C}$)**.**
Let e_1, e_2, e_3 be properly quantified effect formulas, ϵ an atomic effect, $\vec{r}$, $\vec{s}$, $\vec{t}$ ground terms and $F(\vec{s}), F(\vec{t}), G(\vec{r})$ atomic formulas with distinct fluent predicate names F and G. The operator $\mathcal{C}(e_1, e_2)$ is defined inductively over the effects e_1 and e_2:

$$\mathcal{C}(F(\vec{s}), F(\vec{t})) = F(\vec{s}) \tag{4.15}$$
$$\mathcal{C}(\neg F(\vec{s}), F(\vec{t})) = \vec{s} \neq \vec{t} \Rightarrow \neg F(\vec{s}) \tag{4.16}$$
$$\mathcal{C}(F(\vec{s}), G(\vec{t})) = F(\vec{s}) \tag{4.17}$$
$$\mathcal{C}(\neg F(\vec{s}), G(\vec{r})) = \neg F(\vec{s}) \tag{4.18}$$
$$\mathcal{C}(F(\vec{s}), \neg F(\vec{t})) = \vec{s} \neq \vec{t} \Rightarrow F(\vec{s}) \tag{4.19}$$
$$\mathcal{C}(\neg F(\vec{s}), \neg F(\vec{t})) = \neg F(\vec{s}) \tag{4.20}$$
$$\mathcal{C}(F(\vec{s}), \neg G(\vec{r})) = F(\vec{s}) \tag{4.21}$$
$$\mathcal{C}(\neg F(\vec{s}), \neg G(\vec{r})) = \neg F(\vec{s}) \tag{4.22}$$
$$\mathcal{C}(e_1 \wedge e_2, e) = \mathcal{C}(e_1, e) \wedge \mathcal{C}(e_2, e) \tag{4.23}$$
$$\mathcal{C}(\gamma \Rightarrow e_1, e_2) = \gamma \Rightarrow \mathcal{C}(e_1, e_2) \tag{4.24}$$
$$\mathcal{C}(\forall x{:}\tau\; e_1, e_2) = \forall x{:}\tau\, \mathcal{C}(e_1|^x_v, e_2)|^v_x \tag{4.25}$$
where v is a new variable not occurring in e_1 or e_2
$$\mathcal{C}(\epsilon, e_1 \wedge e_2) = \mathcal{C}(\mathcal{C}(\epsilon, e_1), e_2) \tag{4.26}$$
$$\mathcal{C}(\epsilon, \forall x{:}\tau\; e_1) = \begin{cases} \bigwedge_{v \in \mathit{Free}(\tau, \epsilon)} \mathcal{C}(\epsilon, e_1|^x_v) & \text{if } \mathit{Free}(\tau, \epsilon) \neq \emptyset \\ \epsilon & \text{else} \end{cases} \tag{4.27}$$
$$\mathcal{C}(\epsilon, \gamma \Rightarrow e_1) = \mathcal{R}_1(\gamma, \epsilon) \Rightarrow \mathcal{C}(\epsilon, e_1) \wedge \neg \mathcal{R}_1(\gamma, \epsilon) \Rightarrow \epsilon \tag{4.28}$$

In the following, we present some examples for effect chaining:

1. $\mathcal{C}(p(l), p(l)) \overset{(4.15)}{=} p(l)$
2. $\mathcal{C}(\neg p(l), p(l)) \overset{(4.16)}{=} (l \neq l \Rightarrow \neg p(l)) = \bot$
3. $\mathcal{C}(\neg p(l), p(r)) \overset{(4.16)}{=} (l \neq r \Rightarrow \neg p(l))$

4. $\mathcal{C}(p(a) \wedge \neg p(b), p(b)) \overset{(4.23)}{=} \mathcal{C}(p(a), p(b)) \wedge \mathcal{C}(\neg p(b), p(b))$
$\overset{(4.15)}{=} p(a) \wedge \mathcal{C}(\neg p(b), p(b))$
$\overset{(4.16)}{=} p(a) \wedge \bot$
$= p(a)$

5. $\mathcal{C}(p(b) \wedge \neg p(a), p(b)) \overset{(4.23)}{=} \mathcal{C}(p(b), p(b)) \wedge \mathcal{C}(\neg p(a), p(b))$
$\overset{(4.15)}{=} p(b) \wedge \mathcal{C}(\neg p(a), p(b))$
$\overset{(4.16)}{=} p(b) \wedge a \neq b \Rightarrow \neg p(a)$

6. $\mathcal{C}(p(a) \Rightarrow q(a), p(a)) \overset{(4.24)}{=} p(a) \Rightarrow \mathcal{C}(q(a), p(a))$
$\overset{(4.17)}{=} p(a) \Rightarrow q(a)$

7. $\mathcal{C}(\forall o{:}obj\, p(o), \neg p(a)) \overset{(4.25)}{=} \forall o{:}obj\, \mathcal{C}(p(v), \neg p(a))|_o^v$
$\overset{(4.16)}{=} \forall o{:}obj\, (v \neq a \Rightarrow p(v))|_o^v$
$= \forall o{:}obj.\, o \neq a \Rightarrow p(o)$

8. $\mathcal{C}(\forall o{:}obj\, \neg p(o) \wedge q(a, o), q(a, a) \Rightarrow p(a))$
$\overset{(4.25)}{=} \forall o{:}obj\, \mathcal{C}(\neg p(v) \wedge q(a, v), q(a, a) \Rightarrow p(a))|_o^v$
$\overset{(4.23)}{=} \forall o{:}obj\, \left[\mathcal{C}(\neg p(v), q(a, a) \Rightarrow p(a)) \wedge \mathcal{C}(q(a, v), q(a, a) \Rightarrow p(a))\right]\Big|_o^v$
$\overset{(4.28)}{=} \forall o{:}obj\, \Big[\mathcal{R}_1(q(a, a), \neg p(v)) \Rightarrow \mathcal{C}(\neg p(v), p(a))$
$\wedge \neg\mathcal{R}_1(q(a, a), \neg p(v)) \Rightarrow \neg p(v)$
$\wedge \mathcal{R}_1(q(a, a), q(a, v)) \Rightarrow \mathcal{C}(q(a, v), p(a))$
$\wedge \neg\mathcal{R}_1(q(a, a), q(a, v)) \Rightarrow q(a, v)\Big]\Big|_o^v$
$\overset{(4.2,4.3)}{=} \forall o{:}obj\, \Big[q(a, a) \Rightarrow (v \neq a \Rightarrow \neg p(v))$
$\wedge \neg q(a, a) \Rightarrow \neg p(v)$
$\wedge (a \neq v \wedge q(a, v) \vee a = v) \Rightarrow q(a, v)$
$\wedge \neg(a \neq v \wedge q(a, v) \vee a = v) \Rightarrow q(a, v)\Big]\Big|_o^v$
$= \forall o{:}obj\, \Big[\, (v \neq a \vee \neg q(a, a) \Rightarrow \neg p(v)) \wedge q(a, v)\Big]\Big|_o^v$
$= \forall o{:}obj\, \Big[\, (o \neq a \vee \neg q(a, a) \Rightarrow \neg p(o)) \wedge q(a, o)\Big]$

9. $\mathcal{C}(\forall x{:}obj.\ q(x) \Rightarrow p(x), \forall x{:}obj.\ \neg p(x))$
   $\overset{(4.25)}{=} \forall x{:}obj.\ \mathcal{C}(q(v) \Rightarrow p(v), \forall x{:}obj.\ \neg p(x))\big|_x^v$
   $\overset{(4.24)}{=} \forall x{:}obj.\ [q(v) \Rightarrow \mathcal{C}(p(v), \forall x{:}obj.\ \neg p(x))]\big|_x^v$
   $\overset{(4.27)}{=} \forall x{:}obj.\ [q(v) \Rightarrow \mathcal{C}(p(v), \neg p(v))]\big|_x^v$
   $\overset{(4.19)}{=} \forall x{:}obj.\ (q(v) \Rightarrow \bot)\big|_x^v$
   $= \bot$

Using the chain operator $\mathcal{C}$, we can now define an algorithm that computes the effect of a sequence of actions. As shown in Listing 4.11, the algorithm loops over all effects starting with the last effect and adds each effect by chaining it to the currently computed effect. The resulting effect is the effect of the whole action sequence.

Listing 4.11: Generating a macro effect.

```
1 input: int n, effect formulas e_1, ..., e_n
2 output: effect formula e_m
3 begin
4   e_m := e_n
5   for i := n-1 to 1
6     e_m := C(e_i, e_m) ∧ e_m
7   end
8   return e_m
9 end
```

Limitations

1. We have already restricted effect formulas to *properly quantified formulas*. With the definition of the chain operator $\mathcal{C}$ above, we can show an example why we need this restriction. Consider two actions a_1, a_2 with the effects $e_1 = p(a)$ and $e_2 = \forall o{:}obj.\ c(o) \Rightarrow \neg p(a)$.

$$\begin{aligned}\mathcal{C}(e_1, e_2) &= \mathcal{C}(p(a), \forall o{:}obj.\ c(o) \Rightarrow \neg p(a)) \\ &= \mathcal{C}(p(a), c(a) \Rightarrow \neg p(a)) \\ &= \mathcal{R}_1(c(a), p(a)) \Rightarrow \mathcal{C}(p(a), \neg p(a)) \wedge \neg\mathcal{R}_1(c(a), p(a)) \Rightarrow p(a) \\ &= \neg c(a) \Rightarrow p(a)\end{aligned}$$

   Thus, the effect e_m of macro a_m resulting from the effect generation would be

$$e_m = \neg c(a) \Rightarrow p(a) \wedge \forall o{:}obj.\ c(o) \Rightarrow \neg p(a)$$

However, e_m is not a macro representation of the effects of $\langle a_1, a_2 \rangle$. Consider a world w with $w \models c(b) \wedge \neg c(a)$. Since $w \models c(b)$, it follows that $w \models [a_2] \neg p(a)$, and therefore also $w \models [a_2][a_1] \neg p(a)$, because the effect of a_2 is applied last. However, $w \models \neg c(a)$, and thus $w \models [a_m] p(a)$. This example shows that the chaining $\mathcal{C}(e_1, e_2)$ is incorrect if e_2 is not properly quantified. The correct macro representation of the effects of $\langle a_1, a_2 \rangle$ would be:

$$e_m = \forall o{:}obj\, [\neg c(o)] \Rightarrow p(a) \wedge \forall o{:}obj.\, c(o) \Rightarrow \neg p(a)$$

Note that the quantifier in the first conjunct only quantifies over the condition $c(o)$, while the quantifier in the second conjunct quantifies over the whole effect formula $c(o) \Rightarrow \neg p(a)$. Thus, the first conjunct is a conditional effect with a $\forall$-quantifier in the condition, while the second conjunct is a $\forall$-quantified effect with a nested conditional effect. Despite the syntactic similarity, the effects are very different. While we can give a macro representation for the example above, it is not immediately clear how to chain non-properly quantified effects in the general case. For that reason, we require all effects to be properly quantified.

2. Additionally, the chaining of effects is incorrect if both effects are $\forall$-quantified, and one quantifier ranges over a subtype of the other quantifier. As an example, consider the case where we have a type *obj* with a sub-type *thing*. Consider the two actions a_1 and a_2 with effects $e_1 = \forall o{:}obj.\, p(o)$ and $e_2 = \forall o{:}thing.\, \neg p(o)$. The chaining $\mathcal{C}(e_1, e_2)$ of the two effects is the following:

$$\begin{aligned} \mathcal{C}(e_1, e_2) &= \mathcal{C}(\forall o{:}obj.\, p(o), \forall o{:}thing.\, \neg p(o)) \\ &= \forall o{:}obj. \mathcal{C}(p(v), \forall o{:}thing.\, \neg p(o))\big|_o^v \\ &= \forall o{:}obj\, p(o) \end{aligned}$$

In the last step, $\mathit{Free}(thing, p(v)) = \emptyset$ because v is of type *obj*. However, we need to take into account that *thing* is a sub-type of *obj*, and therefore we need to make a case distinction for v whether it is a *thing* or not. The correct result for the chaining would be the effect formula

$$\forall o{:}obj.\, \neg thing(o) \Rightarrow p(o)$$

While we can express this in $\mathcal{ES}$ and it is also consistent with the ADL semantics introduced in Section 2.4, this cannot be expressed in PDDL because we cannot state $\neg thing(o)$ in PDDL. We omit this case in the definition of the chaining $\mathcal{C}$.

Closure under chaining $\mathcal{C}$ Even if we take the above limitations into account, we can see that some PDDL dialects are not closed under chaining. In particular, STRIPS effects are not closed under chaining. Consider Example 3 above: The result of the chaining of the two STRIPS effects $\neg p(l)$ and $p(r)$ is $\mathcal{C}(\neg p(l), p(r)) = l \neq r \Rightarrow \neg p(l)$, which is a conditional effect and therefore not a STRIPS effect. However, with the limitations above, ADL effects are closed under chaining.

Implementation Similar to the implementation of the precondition computation, we implemented effect chaining in Prolog. The central component in the Prolog implementation is the predicate `resolve_conflicting_effects`, which implements the chaining operator $\mathcal{C}(e_1, e_2)$.

Listing 4.12: The implementation of the chaining $\mathcal{C}(\epsilon, \forall x{:}\tau\ e_2)$.

```
resolve_conflicting_effect(
  (all([(Type,[Var|TypedVars])|Vars],QuantifiedEffect),PrevParams),
  (Effect,Params),
  QuantifiedVars,
  (ResEffect,ResParams)
) :-
  findall(ResSubParameterizedEffect,
    ( is_in_typed_list(EffectVar, Params),
      has_type(EffectVar, Params, EffectVarType),
      ( EffectVarType = Type
      ; domain:subtype_of_type(EffectVarType, Type)
      ),
      substitute(Var, [QuantifiedEffect], EffectVar,
        [SubstitutedEffect]),
      resolve_conflicting_effect(
        (all([(Type,TypedVars)|Vars], SubstitutedEffect),
          [(Type,[EffectVar])|PrevParams]),
        (Effect,Params),
        [(Type,[EffectVar])|QuantifiedVars],
        ResSubParameterizedEffect
    )),
    ResSubParameterizedEffects
  ),
  ( ResSubParameterizedEffects = []
  -> ResEffect = Effect, ResParams = Params
  ; maplist(
      \ParameterizedEffect^SubEffect^SubParams^(
        =(ParameterizedEffect,(SubEffect,SubParams))),
      ResSubParameterizedEffects,
      ResSubEffects,
      SubParamsList
    ),
    ResEffect =.. [and|ResSubEffects],
    merge_typed_lists(SubParamsList, ResParams)
  ).
```

Listing 4.12 shows the implementation of the chaining $\mathcal{C}(\epsilon, \forall x{:}\tau\ e_2)$. Again, the order of parameters is different to the order in the definition of $\mathcal{C}$. Also note that each effect is actually a pair of an effect formula a typed list of the formula's variables. The additional parameter `QuantifiedVars` is kept so we can distinguish between the effects parameters

and quantified variables in recursive calls. Starting in line 7, we find each free variable v in ϵ and compute the chaining for each substitution $x \to v$ in line 15. All chained substituted effects are collected in the variable `ResSubParameterizedEffect`. If no such substitution exists, we are in the case $\mathit{Free}(\tau, \epsilon) = \emptyset$ and the resulting effect is ϵ (cf., Equation 4.27). Otherwise, we reconstruct the parameter list in line 26-32 and then compute the resulting effect as conjunction of all chained substituted effects in line 33.

4.3.4 Parameters

In order to compute the parameters of a macro, we need to merge the parameters of all actions in the respective action sequence while taking the parameter reassignment into account. As an example, consider the action sequence `<align-to, put>` with the parameter enumeration `[1],[2,1]`, as shown in Table 4.1. The resulting parameters of the macro should be $\{l_1{:}\mathit{location}, c_1{:}\mathit{cup}\}$.

Listing 4.13: The algorithm to compute the parameters of a macro. The input is a sequence of parameters that contains all parameters of all operators in the macro's operator sequence. The resulting parameters are the parameters of the macro operator.

```
1  input: a sequence of parameters P = ⟨p_1:τ_1, ..., p_k:τ_k⟩
2  output: a set of parameters P_m = {p_{m,1}:τ_{m,1}, ..., p_{m,l}:τ_{m,l}}
3  begin
4    P_m := ∅
5    for p:τ ∈ P:
6      if p:τ' ∉ P_m for any type τ':
7        P_m := P_m ∪ {p:τ}
8      else if p:τ' ∈ P_m and τ ⊊ τ':
9        P_m := (P_m \ {p:τ'}) ∪ {p:τ}
10     end
11   end
12   return P_m
13 end
```

Additionally, it may be the case that two actions have the same reassigned parameter, but one is a sub-type of the other. In that case, we need to restrict the parameter to the sub-type. As an example, consider the action sequence `<goto, align-to>` with the parameter assignment `[1],[1]`. Assume for now that the parameter `?to` of the action `goto` is of type `location`, while the parameter of the action `align-to` is of type `table`, a sub-type of `location`. Thus, the resulting parameter of the macro operator needs to be restricted to the type `table`. Listing 4.13 shows pseudo-code for the parameter computation. The algorithm takes as input the sequence of all parameters of the macro's operators and returns a set of parameters for the macro operator. Note that the input is

a sequence and thus may contain the same parameter twice while the output can contain each parameter only once.

4.3.5 PDDL Representation

In the final step, we take the generated ADL operator $(\vec{y}{:}\vec{\tau}, \pi_m, e_m)$ and translate it back into a PDDL string representation. In order to do this, we re-use the definite clause grammar presented in Section 4.3.1. The implementation of definite clause grammars in Prolog is bi-directional: It can be used both for parsing and generating a language. In addition to reducing the implementation effort, this has the advantage that it ensures compatibility of the input (the original domain) and the output (the generated macro).

4.4 Macro Pruning

Identifying frequent action sequences and generating macro actions from those sequences potentially results in a large number of macros. From those macros, we need to select macros that are most promising. In order to do so, we use *macro evaluators*, which assign a score to each macro and to each set of macros. The goal is to find evaluators that assign high scores to useful macros and low scores to not so useful macros. To evaluate a single macro, we use two properties of the macro:

Frequency $f(m)$**:** The number of occurrences of the corresponding action sequence σ_m of m in the seed plans.

Parameter Reduction $p(m)$ The difference between the sum over the number of parameters of the operator sequence and the number of parameters of the macro operator. If m is a macro operator with k_m parameters, $\langle a_1, \ldots, a_n \rangle$ is the corresponding operator sequence, and k_i is the number of parameters of operator a_i, then the parameter reduction $p(m)$ of macro m is defined as:

$$p(m) = \sum_{i=1}^{n}(k_i) - k_m$$

Now we can define an evaluator that evaluates a macro by a weighted sum of frequency and parameter reduction with some weight $w \in [0, 1]$:

$$FP_w(m) = wf(m) + (1 - w)p(m)$$

When evaluating a set of macros, we additionally need to compare the macros' similarity. We want to avoid sets of macros that are very similar, since a duplicate macro increases the size of the search space without adding any benefit. Therefore, we use the *complementarity of the macros* as measurement for macro sets.

Definition 4.4.1 (Complementarity of a set of macros)**.**
Let $M = \{m_1, \ldots, m_n\}$ be a set of macros and for each macro, let $\sigma_i = \langle a_{i_1}, \ldots a_{i_{l_i}} \rangle$ the corresponding action sequence. The complementarity $C(M)$ of the macro set M is

$$C(M) = \frac{\left|\bigcup_{i=1}^{n} \left\{a_{i_1}, \ldots, a_{i_{l_i}}\right\}\right|}{\sum_{i=1}^{n} \left|\left\{a_{i_1}, \ldots, a_{i_{l_i}}\right\}\right|}$$

Using the complementarity, we can now define the evaluation function CFP:

Definition 4.4.2 (Evaluation function CFP(M))**.**
The evaluation $CFP_w(M)$ of a set of macros M with some weight $w \in [0, 1]$ is:

$$CFP_w(M) = C(M) \cdot \frac{\sum_{m \in M} FP_w(m)}{\sqrt{|M|}}$$

Note that we divide by the square root of the number of macros in the set M: If we do not take the number macros into account, then adding another macro to M will always increase the evaluation score as long as the macro does not share any actions with a macro in the set. This is the case even if the additional macro only occurs once in the database: For any macro m that was generated from the plan database, $FP_w(m) > 0$ if $w > 0$. On the other hand, we should not choose a factor of $\frac{1}{|M|}$, because then a singleton set $\{m\}$ would always have the highest evaluation score. For this reason, we chose a factor of $\frac{1}{\sqrt{|M|}}$. In the following, we will denote

- $CFP_{0.5}$ as CFP,
- CFP_1 as CF,
- and CFP_0 as CP.

4.5 Planning with DBMP Macros

After identifying, generating, and pruning macros, we obtain a macro-augmented domain that can be used to plan further problems. The macro-augmented domain is the original domain with an additional action, as shown in Listing 4.14. The macro is represented as a normal ADL operator. Thus, there is no need to modify the underlying planner. Instead, we can call the planner as usual but give it the augmented domain instead of the original domain.

4.5.1 Macro Expansion

The resulting plans contain macro actions. These need to be substituted by the action sequence the macro was generated from, which we call *macro expansion*. Parameters have to be instantiated according to the macro definition and the macro's parameters. In order to do this, we save additional information in the domain file when augmenting the domain with a macro. This information contains:

Listing 4.14: The macro goto-align_to.

```
; MACRO goto-align_to ACTIONS [goto,align_to] PARAMETERS [[1],[1]]
( :action goto-align_to
  :parameters ( ?p1 - location )
  :precondition
    (and (not(exists(?l - location)
           (or (aligned ?l) (looking_at ?l))))
         (not (robot-at ?p1))
         (alignable ?p1 ?p1))
  :effect
    (and (aligned ?p1)
         (robot-at ?p1)
         (forall (?l - location)
           (when (not (= ?p1 ?l))
             (not (robot-at ?l)))))
)
```

- The name of the macro action.
- The action sequence that the macro represents.
- The parameter enumeration for each action in the sequence.

As an example, consider the macro that represents the action sequence <goto,align-to> where the parameters of each action are assigned to the same macro parameter, i.e., the resulting macro has only one parameter of type location. In the augmented domain, the macro is represented as shown in Listing 4.14.

In order to translate a macro action in a plan back to the original action sequence, we provide a wrapper for the planner that calls the planner, parses the planning result, translates the macro into the action sequence, and then returns the translated plan. In addition to translating macro actions, this provides the advantage that the resulting plan is always of the same format, no matter which planner is used.

5 Evaluation

In the following, we describe how we evaluated DBMP. First, we describe our lab cluster setup, which allowed us to compute a large number of plans. Then, we describe the domains that we used to evaluate DBMP. Next, we analyze our approach to macro identification and generation, before we compare the DBMP performance to PDDL planning without macros and to other macro planners. Finally, we have a look at our approach to macro pruning using evaluation functions.

5.1 Kubernetes Lab Cluster

In order to be able to run a large number of planning tasks, we set up a Kubernetes cluster as described in Section 2.9.4. The setup consists of seven machines with a Quad-core Intel Core i7-3770 CPU @ 3.40GHz and 16GB memory each. We dedicated one of the machines to be the cluster master and database server. The other six machines served as cluster nodes. All domains, problems, and solutions are stored in the database, as described in Section 4.1.

We used the cluster for two purposes: First, we computed seed plans in the cluster, which where then used to generate macros. Second, we ran all benchmarks in the cluster. For the seed plans, we limited all jobs to 60min runtime and 7GB memory usage. This allows to run two concurrent jobs on each node, resulting in twelve parallel jobs in the cluster. For benchmarks, we limited all tasks to 30min and 4GB memory, allowing three concurrent jobs on each node and therefore 18 concurrent jobs in the cluster.

Since the machines are standard lab machines which are also used by other people, we configured each machine in a way that whenever a user logs in, the machine would stop all running jobs and disable scheduling of further jobs on the machine, until the user logged out again. In this case, all running and pending jobs on that machine are automatically rescheduled to other cluster nodes. This setup allows to fully use the lab machines without disturbing other users. Additionally, we have a higher control over available resources, because we only run jobs when no other user uses the machine.

Resources are managed by Kubernetes, as described in Section 2.9.4. This ensures that each planning task has the same resources available. The cluster setup is automated with Ansible (cf., Section 2.9.5). Although our cluster is relatively small compared to typical setups with tens to hundreds of cluster nodes, it already proved to be very useful for our purposes. The highly automated setup resulted in well-structured results and made it possible to run 19666 planning tasks with a total runtime of 3856 hours.

5.2 Domains

In this section, we describe the domains we used to evaluate DBMP.

Blocksworld *Blocksworld* is a STRIPS domain from the International Planning Competition (IPC). The goal in Blocksworld is to rearrange a number of blocks such that they are stacked in a certain order. As an example, one instance can be to arrange the three stacked blocks `b1,b2,b3` to be in reverse order, i.e., `b3,b2,b1`. In this variant of Blocksworld, there is unlimited room on the table. For our evaluation, we used 100 generated blocksworld instances with 20 blocks in each problem.

Hiking *Hiking* is another STRIPS domain from the IPC. The setting is the following: A couple wants to do a circular hike over several days. The couple has two cars available, which must be used to carry luggage and their tent. Each day, the couple can do one leg of the hike, and the tent has to be set up when the couple reaches the endpoint. Compared to Blocksworld, Hiking has more actions, and each action has more parameters. This additional complexity can also be seen in the results. For this domain, we used the 20 IPC 2014 demo problems [75].

Barman *Barman* is the third and most complex STRIPS domain from the IPC that we used for our evaluation. In Barman, a robot has to mix a set of drinks using dispensers, glasses, and shakers. Each drink consists of several ingredients that need to be mixed. Additionally, the robot may only use clean and empty glasses, and it has a limited number of hands, usually two. As for Hiking, we used the 20 IPC 2014 demo problems.

Cave Diving *Cave Diving* is an ADL domain from the IPC that also uses action costs. In Cave Diving, there are a number of caves that are connected in the structure of an undirected acyclic graph. There are number of divers whose task is to take pictures of certain caves. There can be only one diver in a cave, and diving and taking pictures uses air. Divers can also carry oxygen tanks, which they can leave in any cave for other divers. Each diver has hiring costs and some divers refuse to work with other divers. We use an adapted version of the IPC domain and its 20 demo problems. As we do not support action costs, we removed any costs from the domain and use it without action costs. This makes the domain simpler as we do not need to optimize the cost, but the problem structure remains the same.

Cleanup In the *Cleanup* domain, a domestic service robot must clean up dishes by putting clean dishes back onto the counter and dirt dishes into the dishwasher. It is an adapted version of our robotics lab scenario [37]. Cleanup is an ADL domain that makes heavy use of quantified and conditional effects. The original version of the domain has also been executed on a real robot with a continual planner and execution monitor [35]. For this domain, we generated 230 problems with 1 to 20 cups and for each number n of cups, we set 0 to n cups to `dirty`.

RCLL The RCLL domain is a domain in the setting of the RoboCup Logistics League, as described in detail in Section 2.6. It is an adapted version of a draft for the RoboCup Logistics League Planning Competition at the ICAPS 2017 [57][1]. This domain is particularly challenging because even small problems take a long time to solve or cannot be solved by the planners that we used for our benchmark. Thus, generated macro actions are based on solutions of the simplest problem instances only. For the RCLL domain, we used a problem generator that produces random order of different complexity, from `C0` to `C3`. For each complexity, we generated 0 to 2 orders, resulting in 81 problems.

5.3 Seed Plan Generation

In order to generate seed plans, we ran either FF or Fast Downward on all problems of each domain, and used all successful plans for macro seeding. For seed planning, we limited the planner to 60min runtime and 7GB of memory. Table 5.1 shows the seeding configurations and results for all domains.

Domain	Seed Planner	# Problems	# Solutions	Total Time (min)
Blocksworld	FF	100	79	165
Hiking	FF	20	15	164
Barman	Fast Downward	20	20	1198
Cave Diving	Fast Downward	20	7	83
Cleanup (FF)	FF	230	85	408
Cleanup (FD)	Fast Downward	230	35	555
RCLL	Fast Downward	81	5	61

Table 5.1: The seed planning configuration and results for each domain. By default, we used FF as seed planner. If FF could not find any seed plans, we switched to Fast Downward. The total time is the run time for all successful plans.

5.4 Macro Identification

As described in Section 4.2, we use MapReduce to identify frequent action sequences. By using MapReduce on the MongoDB database, we are able to count all occurring action sequences in a reasonable time. As shown in Figure 5.1, counting all sub-sequences of length 10 or less took 283s in the Cleanup domain. Identifying shorter sequences with a length of 3 or less actions needed less than 10s in all domains. This is sufficiently fast for an off-line step that is done only once per domain.

However, for the Hiking and the Barman domains, identification failed for longer action sequences due to a combinatorial blowup in the parameter assignments and some database limitations. As described in Section 4.2, the documents emitted in the Map step contain a list of all possible parameter assignments for the given actions as shown

[1]The domain is available at `https://github.com/timn/ros-rcll_ros`.

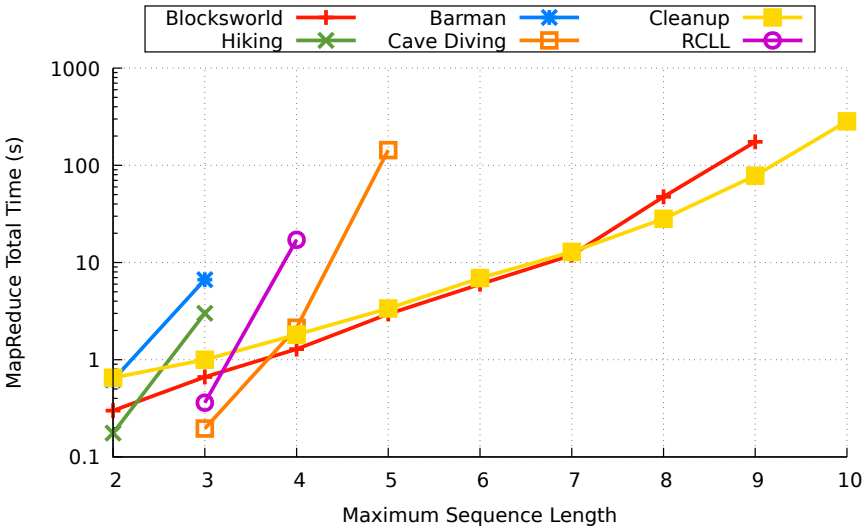


Figure 5.1: The total time needed by the MapReduce operation to identify frequent action sequences (logarithmic time scale). For all domains, the time to identify sequences of length 3 was less than 10s. In the Blocksworld and Cleanup domains, frequent action sequences up to a length of 10 actions could be identified. For the other domains, the algorithm did not scale as well.

in Table 4.1. For actions with many parameters, there are a lot of possible parameter assignments, and thus the emitted document can be quite large. In this case, the limitations by the database can be quickly reached. As an example, when counting the action sequences with a maximum length of 4 in the Barman domain, MongoDB's MapReduce fails with the following error:

```
exception: Error: an emit can't be more than half max bson size
```

This is caused by the action sequence

```
<shake,pour-shaker-to-shot,pour-shaker-to-shot,empty-shaker>
```

In this case, the action sequence has 23 parameters and 33752 possible parameter assignments, which results in a document size larger than 8MB. The size limit for emitted documents is half of the maximum document size, which is 16MB [51]. Therefore, documents of a size larger than 8MB cannot be used in MapReduce.

As a workaround, one can emit a document for each possible parameter assignment instead of emitting a single document that contains all assignments. This results in much smaller documents and the error does not occur anymore. However, this only defers the error from the Map step to the Reduce step, which then fails with the error:

```
exception: value too large to reduce
```

Here, the Map step emitted too many documents with the same key. Since all documents with the same key are merged into one document for reducing, the Reduce step fails again due to the 8MB size limit for documents.

A different solution is to stop emitting all possible parameter assignments. Instead, we could emit only the most specific possible parameter assignment, and then compute generalized parameter assignments in the next step. As an example, when encountering the action sequence `goto(m1), align-to(m1)` in a plan, instead of emitting the enumeration `[1],[1]` and its generalization `[1],[2]`, we would only emit the most specific enumeration `[1],[1]`. However, since we also want to have the generalized action sequence as a candidate for macro generation, we would need to add another step after the MapReduce operation to compute all possible generalizations of the identified action sequences.

5.5 Macro Generation

As described in Section 4.3, we generate a PDDL representation of a macro by regressing each precondition of the action sequence to the beginning of the sequence, and by chaining all the effects of the actions. Macro generation is implemented in Prolog and wrapped in a Python script that fetches the input data from the database, starts the generator, and then saves the generated macros and augmented domains including their evaluation scores in the database. As shown in Figure 5.2, macro generation is sufficiently fast. In all domains, generating a single macro needs about 0.2s and the generation time only slightly increases with a higher number of actions in the macro.

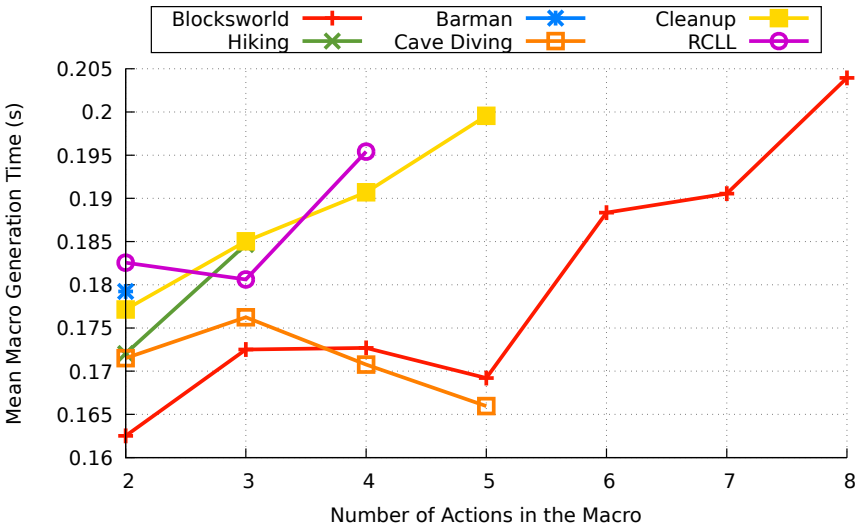


Figure 5.2: Mean times to generate a macro of a given length.

Spot checks of the generated macros showed that the generated actions were indeed macro representations of their respective action sequence, i.e., if the macro's precondition was satisfied, then the respective action sequence was executable, and the macro's effect were equal to the resulting effect of the action sequence.

5.6 Planner Performance

In this section, we compare DBMP to other macro planners and to planning without macros. In order to do so, we computed seed plans for all problems of each domain with either FF or Fast Downward. From those seed plans, we identified the 10 most frequent action sequences of lengths 2 to 5. For each parameter assignment of each of these sequences, we generated a macro. For each macro, we generated an augmented domain. Additionally, we generated an augmented domain for each distinct pair of macro actions. From all those domains, we selected the 10 domains with the highest evaluation score according to the evaluators introduced in Section 4.4. We used the following three evaluators:

CFP Evaluates sets of macros according to their comparability, and each macro with the sum of the macro's frequency and its parameter reduction.

CF Evaluates sets of macros according to their comparability, and each macro with the macro's frequency. Parameter reduction is not taken into account.

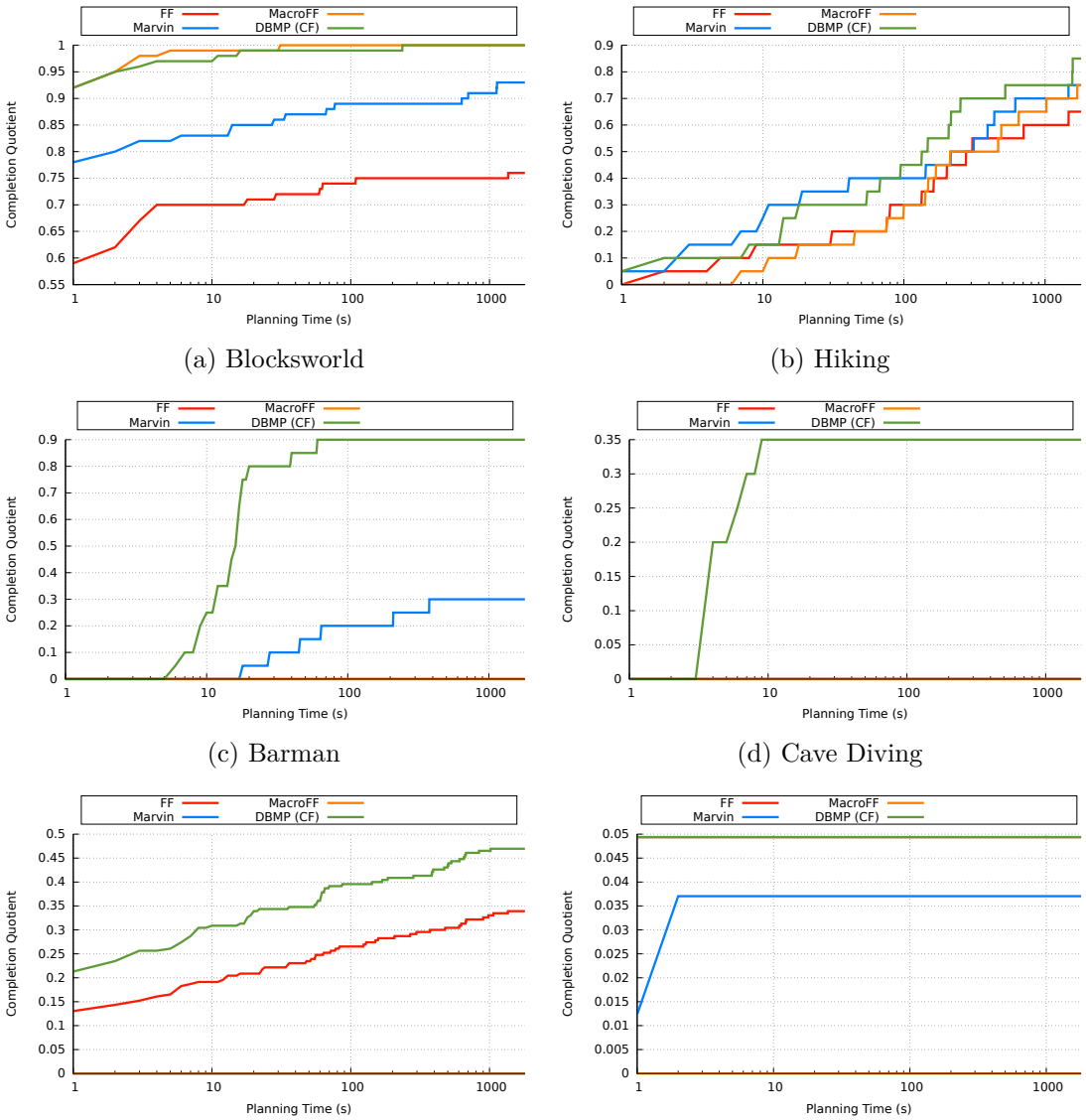
CP Evaluates sets of macros according to their comparability, and each macro with the macro's parameter reduction. The macro's frequency is not taken into account.

Note that, even though CP does not take the frequency into account, macros are still selected according to the frequency, because only the most frequent action sequences are selected for macro generation in the previous step.

After generating the macros, we ran all problems of each domain with FF on the augmented domains. We compare FF + DBMP to FF without macros, and to the macro planners Marvin and Macro-FF. All planners were limited to 30min and 4GB memory. A performance overview can be seen in Figure 5.3. In the following, we have a detailed look at the planning times and the resulting plan lengths.

5.6.1 Planning Times

The planning times for all domains and planners are shown in Table 5.2. For DBMP, we use FF and the three evaluator functions described above. Additionally, the last column gives the results for the fastest augmented domain, i.e., the domain that has the smallest mean time over all problems (including failed problems). For each domain and planner, we counted the number of solved problems and computed the mean planning time for all solved problems (i.e., failed problems were not included in the mean). Additionally, we computed the quartiles of the planning time over all problems (including failed problems). The quartiles are "the three values that divide the items of a frequency distribution into four classes with each containing one fourth of the total population" [49]. In this context, the first quartile $Q1$ is the smallest time in which a quarter of the problems could be solved, the second quartile $Q2$ is the median planning time, and the third quartile is the smallest time in which three quarters of the problems could be solved. Note that we use the quartiles over all problems. Thus, if only half the problems could be solved, then the third quartile $Q3$ is undefined, which is denoted with $\times$. The symbol – marks



(a) Blocksworld (b) Hiking

(c) Barman (d) Cave Diving

(e) Cleanup (FD) (f) RCLL

Figure 5.3: A comparison between FF, Marvin, Macro-FF, and DBMP in each domain. The completion quotient is the share of solved problems within the given time.

domains that are not supported by the particular planner, and the symbol † marks domains where the planner failed due to some error. For the sake of completeness, we also added the planning times for Fast Downward to Table 5.2. However, as we use a FD-configuration that is optimized for plan length and not for planning time, a comparison of the planning times is not useful.

					FF + DBMP			
	FF	FD	Marvin	Macro-FF	CFP	CF	CP	best
Blocksworld - 100 problems - seed: FF								
# solved	76	**100**	93	**100**	**100**	**100**	**100**	**100**
mean (s)	22.0	1011	41.4	**0.72**	1.32	2.82	1.76	1.32
Q1 (s)	0.006	910	0.12	0.16	**0.004**	0.04	0.19	**0.004**
Q2 (s)	0.08	963	0.22	0.23	**0.006**	0.05	0.32	**0.006**
Q3 (s)	80.0	1054	0.62	0.40	**0.011**	0.088	0.79	**0.011**
Hiking - 20 problems - seed: FF								
# solved	13	18	15	15	17	17	18	**20**
mean (s)	265	1339	244	349	321	287	170	**37.7**
Q1 (s)	77.5	1000	9.79	86.6	16.6	15.4	5.97	**4.33**
Q2 (s)	290	1798	263	339	100	141	83.4	**10.6**
Q3 (s)	×	1800	1649	1751	1303	1094	232	**61.4**
Barman - 20 problems - seed: FD								
# solved	0	**20**	6	0	18	18	0	19
mean (s)	×	1798	124	×	**17.2**	**17.2**	×	23.2
Q1 (s)	×	1798	285	×	10.6	10.6	×	**6.07**
Q2 (s)	×	1799	×	×	16.1	16.1	×	**10.2**
Q3 (s)	×	1800	×	×	**18.4**	**18.4**	×	29.0
Cave Diving - 20 problems - seed: FD								
# solved	0	**7**	0	0	0	**7**	4	**7**
mean (s)	×	644	×	×	×	4.77	1107	**3.85**
Q1 (s)	×	804	×	×	×	6.11	×	**6.07**
Cleanup - 230 problems - seed: FF								
# solved	78	27	–	†	62	88	71	**152**
mean (s)	127	212	–	†	154	138	**125**	143
Q1 (s)	71	×	–	†	71	13.7	190	**0.42**
Q2 (s)	×	×	–	†	×	×	×	**54.3**
Cleanup - 230 problems - seed: FD								
# solved	78	27	–	†	108	108	68	**152**
mean (s)	127	212	–	†	87	87	140	**82**
Q1 (s)	71	×	–	†	2.39	2.39	249	**0.36**
Q2 (s)	×	×	–	†	×	×	×	**35.9**
RCLL - 81 problems - seed: FD								
# solved	**4**	5	3	†	**4**	**4**	**4**	**4**
mean (s)	0.156	396	0.74	†	0.255	0.255	0.232	**0.094**

Table 5.2: A comparison of planning times between different planners with and without macros. CF, CFP, and CP denote the three best macro configurations according to the three evaluators, the best DBMP configuration in the last column are the results for the domain with the lowest mean planning time. The symbol '×' denotes undefined values, '–' denotes that the planner does not support the domain, and '†' denotes an error by the planner. Rows with no values have been omitted. The best value per row is indicated in bold face.

Comparison to Planning without Macros As shown in Table 5.2, DBMP clearly outperforms FF in most domains. In the Blocksworld domain, DBMP could solve all 100

problems, while FF could only solve 76 problems. Additionally, the mean planning time differs by a factor of 16 with 22s for FF and 1.32s for DBMP with the best macro configuration. The biggest performance difference can be observed in the third quartile: While FF needed 80s to solve three quarters of the problems, DBMP with the best macro configuration took only 0.011s to solve the same number of problems. Even with the CP evaluator, which performed worst in Blocksworld, DBMP still outperforms FF by a factor of 12.

The Hiking domain shows similar results, albeit in a smaller magnitude. With all evaluators DBMP was able to solve more problems than FF. Additionally, all quartiles of the DBMP solutions are smaller than the quartiles of FF. Interestingly, the mean planning times of CFP and the CF are higher than the mean planning time of FF without macros. Presumably, this is because the two additional problems that were solved by DBMP have a high planning time, and therefore lead to a higher mean time, as we compute the mean only over the solved problems.

The results for the Barman domain show that FF can benefit from DBMP with Fast Downward seed plans even if FF is not able to solve any of the original problems. Without macros, FF could not solve any of the Barman problems. With DBMP macros, it could solve 18 of the 20 problems with the CFP and CF evaluators and even 19 with the best macro configuration. Interestingly, with macros, FF was able to solve the problems quite fast, with a mean planning time of 23.2s and a third quartile of 29s. Thus, macro actions simplify the domain to a great extent.

The results for Cave Diving are very similar. FF cannot solve any problems without macros, but it benefits from DBMP with FD-seeding. With DBMP, FF could solve 7 of the 20 problems with a mean time of 3.85s in the best configuration.

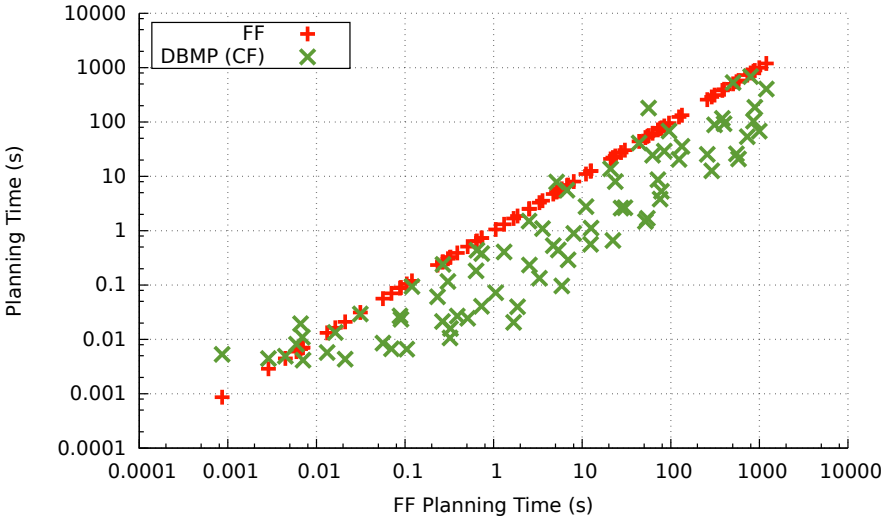


Figure 5.4: DBMP with FF seeding compared to FF without macros in the Cleanup domain (logarithmic scale). Although DBMP performs better than FF, the planning time increases proportionally to the FF planning time.

The *Cleanup* domain shows that planning in a robotics application can also benefit from DBMP. First, consider the Cleanup domain with FF-seeding as shown in Figure 5.4. In the CF configuration, DBMP was faster and could solve more problems than FF without macros. The difference is even bigger in the best configuration, where DBMP could solve 152 of the 200 problems with a planning time of 0.42s in the first quartile. However, in the other configurations, DBMP performed worse.

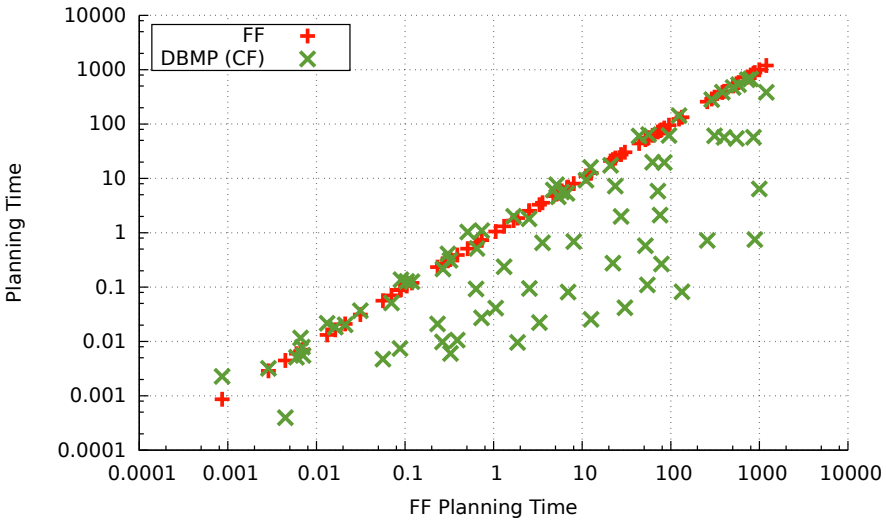


Figure 5.5: DBMP with FD seeding in comparison to planning without macros in the Cleanup domain (logarithmic scale). In this domain, FD brings a clear advantage over FF seeding (cf. Figure 5.4).

When looking at the performance of DBMP in the Cleanup domain with Fast Downward as seed planner as shown in Figure 5.5, we see that combining two planners can bring a significant performance benefit. In the configuration we used, Fast Downward primarily tries to find short plans with the cost of longer planning times. Since we generate seed plans off-line, longer seed planning times do not impair the performance. But the planner can gain from the shorter seed plans, as the seed plans also contain better sub-sequences. As an example, with FF as seed planner, a seed plan may contain the action sequence

```
<pick-up(cup1),put-down(cup1),pick-up(cup1)>
```

which is a useless sub-sequence. When using Fast Downward as seed planner, such sub-sequences are much rarer and the resulting plans are shorter. Thus, the generated macros are also of higher quality, which results in better planning performance: In the CFP configuration, DBMP can solve 108 of 230 problems with FD seed plans instead of 62 problems with FF seed plans, and those plans are computed in a shorter time.

Finally, in the RCLL, the number of solved problems does not increase and the mean planning times are worse for all configurations but the best one. We assume this is because there are four RCLL problems which consist of a single order, while all other

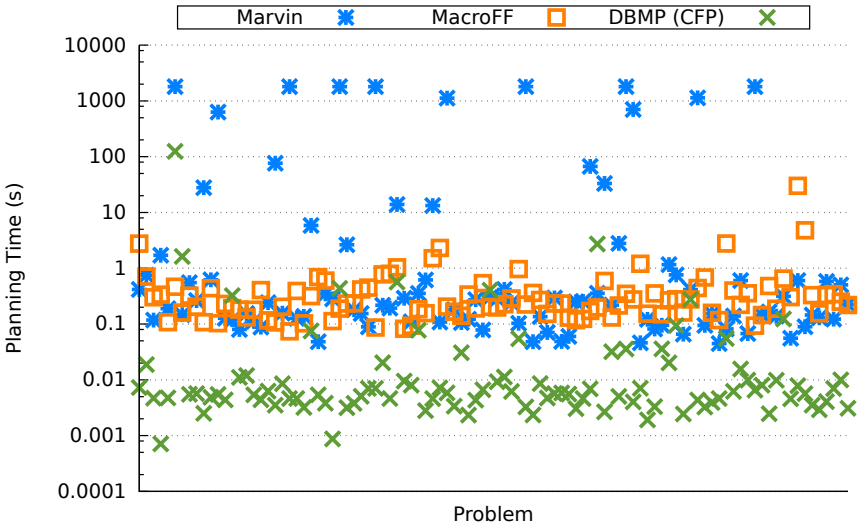


Figure 5.6: A comparison of the three macro planners MARVIN, MACRO-FF, and DBMP with the best CFP configuration in the Blocksworld domain. While DBMP performed best on most problems, it has more outliers with a long planning time, leading to a higher mean planning time than MACRO-FF.

problems contain multiple orders. Thus, the differencce in difficulty between these four problems and the other problems is severe. Apparently, FF can solve these four problems even without macros, while macros do not add enough benefit to solve a more complex problem. Only in the best configuration, DBMP plans faster than FF.

Comparison to other Macro Planners We now compare the planning times of FF with DBMP to other macro planners, in particular to MARVIN and MACRO-FF. For this comparison, we used MARVIN without a macro repository, i.e., we only used on-line macro generation, and we used MACRO-FF in the Solution-Enhanced Planner (SOL-EP) variant, which generally supports ADL domains. The results are shown in Table 5.2.

In the *Blocksworld* domain, we can see that DBMP and MACRO-FF can solve all 100 problem instances, while MARVIN can only solve 93 problems. Figure 5.6 shows the three macro planners in comparison. The planning times of DBMP are better than the planning times of MARVIN, and for most problems, DBMP also performed better than MACRO-FF. However, for one problem DBMP needed more than 100s. For this reason, the mean time for DBMP is higher than for MACRO-FF. In the worst configuration, DBMP has a mean planning time of 2.82s, while MACRO-FF could solve a problem in 0.72s in average. Thus, while DBMP performed better on most problems, MACRO-FF performed best in average.

In the *Hiking* domain, all configurations of DBMP could solve more problems than MARVIN and MACRO-FF. MARVIN performed better in the first quartile with 9.79s compared to 86.6s of MACRO-FF and 16.6s of DBMP (CFP). However, in the second

and third quartile, DBMP has a smaller planning time than the other two planners. All macro planners performed better than FF.

The *Barman* domain shows interesting results which can be explained with the different approaches of the macro planners. Since FF could not solve any of the problems, Macro-FF could not solve any problem either. This is because Macro-FF is an offline macro planner that generates macros after solving a problem. If the planner cannot solve any problem instance, it cannot generate any macros to solve the more difficult problems. In contrast, Marvin is an on-line macro planner that uses macros to escape plateaus (cf., Section 3.4). Thus, even though FF could not solve any problems and Marvin is based on FF, it was still able to solve 6 of the 20 problems. Since DBMP can use seed plans from any planner, it does not rely on FF. Instead, it uses Fast Downward for seed plans, which allowed DBMP (CFP) to solve 18 of the 20 problems of the domain. We conclude that cross-planner macro generation is indeed useful.

A similar effect occurs in the *Cave Diving* domain, where both Macro-FF and Marvin could not solve any problems, while DBMP could solve 7 of the 20 problems.

In the *Cleanup* domain, DBMP was the only macro planner that could solve any problem. Marvin does not support disjunctive goal formulas and therefore does not support the domain. While Macro-FF generally supports ADL domains, running it on a problem from the Cleanup domain always resulted in a segmentation fault. Thus, it could not be used for this domain despite the general support for ADL.

In the RCLL domain, Marvin could only solve 3 problems and solved the problems with a higher mean planning time than DBMP. Macro-FF failed with the same error as in the Cleanup domain. While DBMP did not perform better than FF without macros, it was the best-performing macro planner in this domain.

In summary, DBMP solves more problems in a shorter time than the other macro planners in most domains. Furthermore, DBMP supports all evaluated domains, while the other planners only implement ADL partially (Marvin) or fail with an error (Macro-FF). Third, in contrast to the other planners, DBMP computes a PDDL representation of macros, which allows to swap the underlying planner.

Differences to Previous Results When comparing the results in Table 5.2 to previous results of DBMP restricted to STRIPS (DBMP/S) [36], some differences can be noticed. In particular, the results of DBMP differ in both the number of solved problems and in planning time. There are several changes that may cause those differences:

- The version of DBMP presented in this thesis is a generalization of DBMP/S, which only supports STRIPS actions. Thus, the resulting preconditions and effects of macro actions may differ, even for STRIPS actions.
- In the previous results, all domains were augmented only with a single macro. In this thesis, we augment domains with multiple macro actions.
- In this thesis, macro actions are selected by evaluation functions, while the macros used previously were hand-picked. More specifically, in DBMP/S, the macro with the highest occurrence count and the most specific parameter assignment was used.

- For the Blocksworld and Hiking domains, this thesis uses FF for seeding, while we always used Fast Downward in DBMP/S.

5.6.2 Plan Lengths

The plan lengths for all domains and planners are shown in Table 5.3. Assuming unit costs for actions, shorter plans are better. As for planning times, we used DBMP with FF and the three evaluators CFP, CF, and CP. Additionally, the last column gives DBMP configuration with the lowest mean plan length over all successfully solved problems. For each planner, we counted the number of solved problems and computed the mean plan length for all solved problems (i.e., failed problems were again not included in the mean). Additionally, similar to the planning times, we computed the quartiles of the plan lengths over all problems (including failed problems).

Comparison to Planning without Macros We first compare the plan lengths DBMP of solutions to planning without macros. As we use a FD-configuration optimized for plan length, Fast Downward found the most and also the shortest solutions, but in a longer time (cf., Section 5.6.1). In the following, we compare DBMP and FF.

In the *Blocksworld* domain, we can see that DBMP had longer plan lengths in all configurations but the best one. As an example, the mean plan length for CFP is 72, while the mean plan length of FF without macros is 63. The best CP configuration performed even worse with a mean plan length of 144 actions. Only in the best configuration, the plan lengths of DBMP were shorter than the plans lengths of FF without macros, especially in the third quartile with 66 actions compared to 86 actions. Generally, the Blocksworld domain suggests that DBMP planning should not be used for simple problems such as most of our Blocksworld problems. This is because for simple problems, FF can find near-optimal solutions in a reasonable time even without macros. As macros may contain unnecessary actions, plan lengths increase unnecessarily. For more difficult problems, this effect is compensated by the fact that FF without macros does not find good solutions anymore, while using macros simplifies the problem and thus leads to better solutions. This is why in the best configuration and with CFP and CF, DBMP has a shorter plan length in the third quartile.

In the *Hiking* domain, DBMP performed slightly better. While the plan length in the first quartile is higher for all DBMP configurations, it is lower in the second quartile. Generally, in the Hiking domain, the plan lengths of FF without macros and FF with DBMP do not differ much.

In the *Barman* and *Cave Diving* domains, no comparison regarding plan length is possible, as FF without DBMP could not find any solutions.

In both *Cleanup* domains, the difference between FF without macros and FF with DBMP is small. Note that the mean plan length mainly differs because FF with DBMP could solve more problems than without DBMP. Looking at the first quartile, both planners performed similarly.

In the RCLL domain, using DBMP macros reduced the plan length slightly. In the best configuration, DBMP performed better with 33 actions compared to 38 actions.

					FF + DBMP			
	FF	FD	Marvin	Macro-FF	CFP	CF	CP	best
Blocksworld - 100 problems - seed: FF								
# solved	76	**100**	93	**100**	**100**	**100**	**100**	**100**
mean	63	**59**	129	190	72	72	144	63
Q1	58	**50**	104	150	64	64	113	61
Q2	65	**58**	122	175	70	70	139	64
Q3	86	**65**	166	211	80	80	172	66
Hiking - 20 problems - seed: FF								
# solved	13	**18**	15	15	17	17	**18**	15
mean	55	**47**	119	117	60	60	64	56
Q1	51	**34**	104	105	53	52	52	52
Q2	64	**42**	132	132	63	63	60	60
Q3	×	**65**	×	×	78	79	74	×
Barman - 20 problems - seed: FD								
# solved	0	**20**	6	0	18	18	0	13
mean	×	214	345	×	178	178	×	**175**
Q1	×	199	381	×	**170**	**170**	×	169
Q2	×	220	×	×	**178**	**178**	×	182
Q3	×	234	×	×	195	195	×	×
Cave Diving - 20 problems - seed: FD								
# solved	0	7	0	0	0	7	4	**8**
mean	×	**23**	×	×	×	23	25	**23**
Q1	×	**23**	×	×	×	23	×	**23**
Cleanup - 230 problems - seed: FF								
# solved	78	27	–	†	62	**81**	71	67
mean	95	**49**	–	†	105	96	87	85
Q1	126	×	–	†	175	123	**118**	121
Cleanup - 230 problems - seed: FD								
# solved	78	27	–	†	**108**	**108**	68	65
mean	95	**49**	–	†	112	112	86	84
Q1	126	×	–	†	123	123	**121**	123
RCLL - 81 problems - seed: FD								
# solved	4	**5**	3	†	4	4	4	4
mean	38	**26**	44	†	35	35	35	33

Table 5.3: A comparison of the resulting plan lengths of all planners in all domains. As before, CF, CFP, and CP denote the DBMP domains with the highest respective score, and the last column gives the values for the macro configuration with the shortest mean plan length. The symbol '×' denotes undefined values, '–' denotes that the planner does not support the domain, and '†' denotes an error by the planner. Rows with no values have been omitted. The best value per row is indicated with bold face.

Concluding, we can say that DBMP does not improve resulting plan lengths, but especially for simpler problems, may lead to longer plans. For more difficult problems, the difference between FF without macros and DBMP disappeared.

Comparison to other Macro Planners We now compare the resulting plan lengths of FF with DBMP to Marvin and Macro-FF. As before, we use Marvin without the macro repository and Macro-FF in the SOL-EP variant.

In *Blocksworld*, we can see that DBMP generates the shortest plans of all macro planners. The mean plan length of DBMP with CFP is 72 actions, compared to 129 actions of Marvin and 190 actions of Macro-FF. The difference is even larger in the third quartile, where DBMP with CFP needed 80 actions, while Marvin needed 166 actions and Macro-FF even 211 actions to solve three quarters of the problems.

We obtain a similar result in the *Hiking* domain. While the mean plan length of DBMP in all configurations is similar to the mean plan length without macros, both Marvin and Macro-FF need twice as many actions in average. This difference seems to be unrelated to the difficulty of the problem: Even in the first quartile, Marvin needs 104 actions and Macro-FF 105 actions compared to 51 actions of FF without macros and 53 actions of DBMP in the CFP configuration.

In the *Barman* domain, we can see that the plans generated by Marvin have a much higher length than the plans generated by DBMP. Additionally, we can see that the plan lengths for DBMP do not get much higher with increasing difficulty. While the plans of Marvin have a length of 381 actions in the first quartile, DBMP only needed 170 actions in the first quartile and only 195 actions in the third quartile. Similarly, in the RCLL domain, Marvin produced the longest plans in average.

Summarizing, planning with DBMP does not increase plan lengths significantly, while other macro planners produce much longer plans than FF with and without DBMP macros.

5.7 Macro Pruning with Evaluators

In this section, we analyze whether the evaluators indeed select the best macro configuration. In particular, we compare the three evaluators CFP, CF, and CP. In order to do so, we computed the ten best macro configurations for each domain according to each evaluator and then ran all problems with the augmented domains. For an optimal evaluator, a higher-scoring domain would always perform better than a lower-scoring domain, and the ranks of the score and the planner performance would be identical. The correlation between the ranks of two datasets can be computed with the Spearman correlation coefficient (SCC) [82]. To analyze evaluators, we compute the SCC between evaluation score and planning time, and between evaluation score and successfully solved problems. For two datasets X and Y, the Spearman correlation coefficient is computed by first ranking X and Y from smallest to largest with resulting rankings R_X and R_Y. Given the two rankings, the Spearman correlation coefficient r_S is defined as

$$r_S = \frac{\mathrm{cov}(R_X, R_Y)}{\sigma_{R_X}\sigma_{R_Y}}$$

where $\mathrm{cov}(R_X, R_Y)$ is the covariance of the ranks and σ_{R_X} (σ_{R_Y}) is the standard deviation of R_X (R_Y).

	planning time			# solved		
	CF	CFP	CP	CF	CFP	CP
Blocksworld	-0.185	-0.154	0.196	0.307	0.347	-0.007
Hiking	-0.008	-0.015	-0.116	0.051	0.097	0.284
Barman	-0.412	-0.312	0.558	0.426	0.321	-0.611
Cave Diving	0.079	0.167	0.068	-0.100	-0.257	-0.098
Cleanup (FF)	0.040	0.041	-0.025	-0.123	-0.130	-0.240
Cleanup (FD)	0.051	-0.136	0.052	0.682	0.691	-0.425
RCLL	0.004	0.003	-0.004	0.000	0.000	0.000

Table 5.4: The Spearman Correlation Coefficients between the three DBMP evaluators, resulting planning times and number of solved problems for all domains. For the planning time, the lower the SCC the better (with -1.0 being optimal) as this corresponds to higher scores for faster planning times. Conversely, for the number of solved problems, the higher the SCC, the better the evaluator (with an optimal SCC of 1.0).

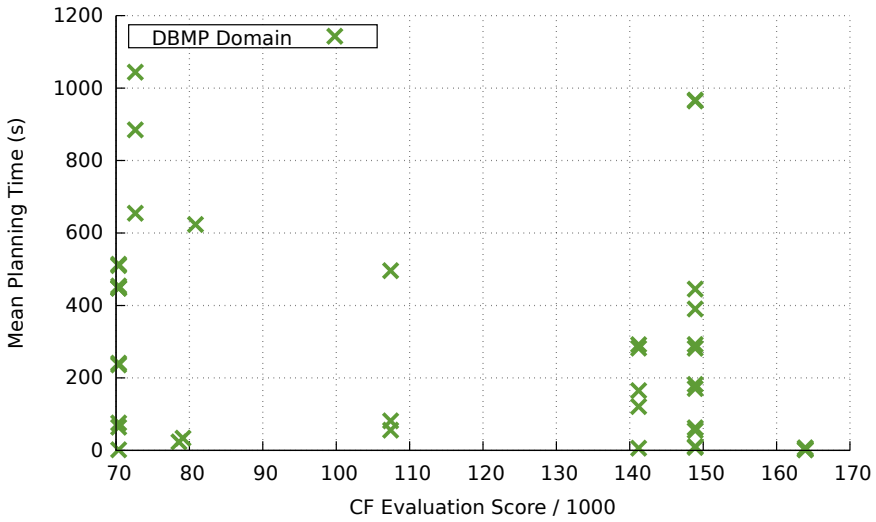


Figure 5.7: The CF evaluator compared to the mean planning time for Blocksworld with a weak negative Spearman correlation coefficient of $r_S = -0.185$.

The results shown in Table 5.4 are mixed. First, we look at the correlation of the evaluation score and planning time. For the CF evaluator, we can see moderate correlation between evaluation score and planning time in the Barman domain ($r_S = -0.412$) and weak correlation in the Blocksworld domain ($r_S = -0.185$), which is also shown in Figure 5.7. In the other domains, there is no or only very weak correlation. Looking at the CFP evaluator, the results look similar but not as strong as CF. With the CP evaluator, higher scores correlate with higher planning times weakly in Blocksworld and moderately in Barman. Thus, CP is an unsuitable evaluator, at least in these domains.

Next, looking at the number of solved problems, we can see a weak correlation between

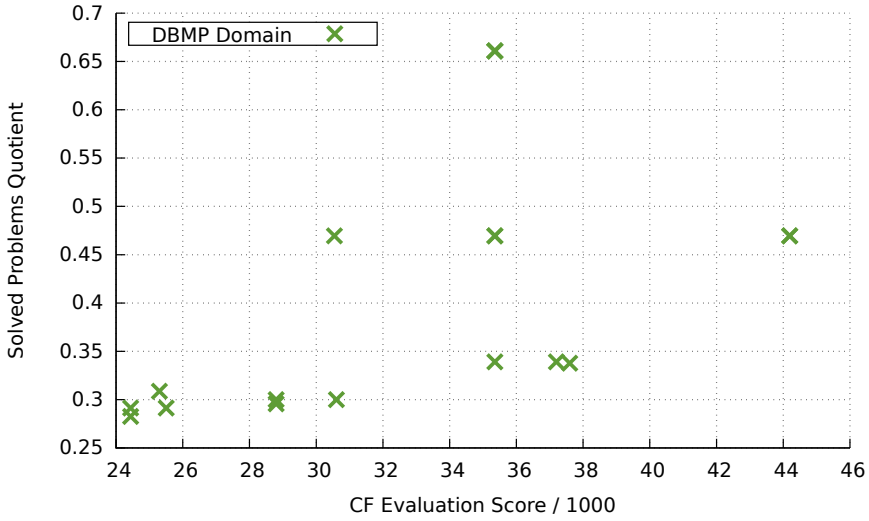


Figure 5.8: The CF evaluator compared to the quotient of solved problems for Cleanup with FD-seeding, which strongly correlate ($r_S = 0.682$).

higher CF scores and a higher number of solved problems for Blocksworld ($r_S = 0.307$), a moderate correlation for Barman ($r_S = 0.426$), and a strong correlation for Cleanup with FD-seeding ($r_S = 0.682$), which is also shown in Figure 5.8. Thus, CF is a suitable evaluator in those domains. The results for CFP scores look again similar, with a higher correlation in Blocksworld ($r_S = 0.347$) and Cleanup with FD-seeding ($r_S = 0.691$), but a lower correlation in Barman ($r_S = 0.321$). Finally, higher scores of CP moderately correlate with a lower number of solved problems in Barman ($r_S = -0.611$) and Cleanup with FD-seeding ($r_S = -0.425$) and weakly correlate in Cleanup with FF-seeding ($r_S = -0.240$). Thus, CP is an unsuitable evaluator in those domains.

In summary, CF performs rather well in Blocksworld and Barman concerning both planning time and number of solutions, and it performs well concerning the number of solutions in the Cleanup domain with FD-seeding. In the other domains, CF could not predict good planning performance. The performance of CFP is similar but not as strong. Finally, CP is a rather unsuitable evaluator overall. We can conclude that macro frequency is a relevant factor for selecting macros, more than parameter reduction. However, other performance measures may also be relevant. As one example, we could add the number of conditional effects and disjunctions in macro actions as a negatively weighted factor, because for most planners, conditional effects and disjunctions impair the planner's performance. As another potential improvement, we can weight occurrence counts according to the time needed to compute the plan in which the action sequence occurs. This would mean that action sequences that occur in solutions for difficult problems would get a higher weight than action sequences of simple solutions, resulting in macros that are mainly useful for solving harder problem instances.

6 Conclusion

We summarize our approach to database-driven macro planning (DBMP) and describe possible improvements both considering its performance and formalization.

6.1 Summary

Database-driven macro planning (DBMP) extracts and generates macros – concatenated action sequences – from a database of previous planning results and is motivated by the observation that plans typically contain the same patterns of actions, especially in robotics domains such as the RCLL or *Cleanup*. We exploit those recurring patterns by storing all previous planning results in a plan database and identifying frequent action sequences with the MapReduce paradigm. Common parameters in the actions of such a sequence are coalesced to reduce the search space expansion caused by the additional macros. In contrast to other macro planners, DBMP represents macros as normal PDDL operators with proper preconditions and effects while supporting the full ADL fragment of PDDL. Thus, we can use DBMP on domains that require features such as disjunctive and quantified preconditions and conditional effects without modifying the underlying planner, which makes the planner exchangeable. Based on a declarative ADL semantics in $\mathcal{ES}$, we provide definitions for a *regression operator* $\mathcal{R}_1$, which regresses a precondition formula on a sequence of effects, and a *chaining operator* $\mathcal{C}$, which combines effects of an operator sequence. We use those operators to compute the PDDL representation for a macro. After generating macros, we select the best macros with *macro evaluators*, which assign a score to each macro configuration based on the macros' frequency, parameter reduction, and complementarity. The selected macros are added to the original domain, which is then used to solve further planning tasks. In a final step, macros in plans resulting from the macro-augmented domains are expanded into the original action sequence so the plan can be executed by the agent.

We have provided a detailed evaluation of DBMP, both in comparison to planning without macros and to the macro planners Marvin and Macro-FF. We have demonstrated that containerization with Docker and Kubernetes simplifies large-scale benchmarks of planning tasks. With this setup, we were able to run 19666 planning tasks with a total runtime of 3856 hours in a well-controlled environment. The results show that DBMP can successfully use previous planning results to enhance planner performance. DBMP clearly outperforms planning without macros, it is able to solve more problems and it solves them in a shorter time. Furthermore, the resulting plans are of similar quality, as using macros does not result in longer plan lengths. In comparison to other macro planners, DBMP is in clear advantage in most domains and it is the only planner that could solve problems in all domains. In a robotics context, a robot can improve its

performance by learning macros over time and by computing seed plans off-line, which improves the more crucial run-time performance.

Summarizing, DBMP significantly improves planner performance by adding macros based on frequent action sequences in a plan database and it represents those macros as PDDL operators, which allows to swap the underlying planner without modifications.

6.2 Future Work

So far, DBMP is a stand-alone planner that can be used like other PDDL planners. As a next step, it may be useful to integrate DBMP into a robot software framework such as Fawkes (Section 2.7) and combine it with an agent knowledge database such as the *robot memory* (Section 2.8.2). Furthermore, extending DBMP beyond ADL with temporal aspects or conditional planning may provide benefits, especially in robotics applications such as the RCLL. Third, so far we have only used FF to plan with macro-augmented domain. The approach should be tested with other PDDL planners in order to verify the substitutability of the underlying planner.

In Section 4.3, we defined a *macro representation m of an operator sequence σ*, where m is a macro representation of σ if (1) the precondition of m entails that σ is executable, and (2) the results of executing m are the same as executing σ. We described how to generate macros with *effect regression* and *effect chaining*. It remains to be shown that the result of effect regression and chaining is indeed a macro representation of the respective operator sequence. To do this, we should first investigate the relation of effect regression to regression in $\mathcal{ES}$ and compare effect chaining to the regression operator defined by Zarrieß and Claßen [79], before we analyze whether effect regression and effect chaining indeed results in a macro representation.

For effect chaining, we put two restrictions on the effect formulas of the chained operators. First, we require all effects to be properly quantified, i.e., we do not allow effect formulas such as $e = \forall o{:}\tau.\, p(o) \Rightarrow q(a)$, where the quantifier ranges only over a condition of a conditional effect. Second, we do not allow the chaining of two $\forall$-quantified effects if one quantifier ranges over a sub-type of the other, e.g., $\mathcal{C}(\forall o{:}\mathit{obj}.\, p(o), \forall o{:}\mathit{thing}.\, \neg p(o))$, where *thing* is a sub-type of *obj*. To lift the first restriction, it may be possible to translate all effects into properly quantified effects. We have shown in Section 4.3 that we can translate the effect $e = \forall o{:}\tau.\, p(o) \Rightarrow q(a)$ into the properly quantified effect $e_m = (\exists o{:}\tau\, p(o)) \Rightarrow q(a)$, assuming there is always an object of type τ. A generalization of this approach may remove the first restriction. Alternatively, we can adapt the definition of the chaining $\mathcal{C}$ such that it also computes the correct resulting effect for effects which are not properly quantified. The second restriction cannot be lifted without further modifying the domain. However, adding complementary sub-types to the domain may allow statements such as "object o is of type *obj* but not of its sub-type *thing*".

Concerning the evaluators, we have seen that the evaluator CF provides a good macro evaluation for some domains, but fails in others (cf., Section 5.7). To improve macro evaluation, it may be useful to consider other macro properties, such as the number of disjunctions and conditional effects in the macro's precondition and effect.

Bibliography

[1] Paul Barham et al. “Xen and the art of virtualization”. In: *ACM SIGOPS Operating Systems Review* 37.5 (2003), p. 164. DOI: 10.1145/1165389.945462.

[2] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. *YAML Ain't Markup Language (YAML™) Version 1.2*. Tech. rep. 2009. URL: http://www.yaml.org/spec/1.2/spec.html.

[3] David Bernstein. “Containers and cloud: From LXC to Docker to Kubernetes”. In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84. DOI: 10.1109/MCC.2014.51.

[4] Avrim L. Blum and Merrick L. Furst. “Fast planning through planning graph analysis”. In: *Artificial Intelligence* 90.1-2 (1997). DOI: 10.1016/S0004-3702(96)00047-1.

[5] Carl Boettiger. “An introduction to Docker for reproducible research”. In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 71–79. DOI: 10.1145/2723872.2723882.

[6] Adi Botea, M. Müller, and Jonathan Schaeffer. “Learning partial-order macros from solutions”. In: *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS)*. 2005.

[7] Adi Botea et al. “Macro-FF: Improving AI planning with automatically learned macro-operators”. In: *Journal of Artificial Intelligence Research (JAIR)* 24 (2005). DOI: 10.1613/jair.1696.

[8] Michael Brenner and Bernhard Nebel. “Continual planning and acting in dynamic multiagent environments”. In: *Autonomous Agents and Multi-Agent Systems* 19.3 (2009). DOI: 10.1007/s10458-009-9081-1.

[9] Daniel Bryce et al. “SMT-based Nonlinear PDDL+ Planning”. In: *Proceedings of the 29th Conference on Artificial Intelligence (AAAI)*. 2015.

[10] Wolfram Burgard et al. “Experiences with an interactive museum tour-guide robot”. In: *Artificial Intelligence* 114.1-2 (1999), pp. 3–55. DOI: 10.1016/S0004-3702(99)00070-3.

[11] Tom Bylander. “Complexity Results for Planning”. In: *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI)* (1992).

[12] Michael Cashmore et al. “A Compilation of the Full PDDL + Language into SMT”. In: *Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS)* (2016), pp. 79–87.

[13] Lukáš Chrpa. "Generation of macro-operators via investigation of action dependencies in plans". In: *The Knowledge Engineering Review* 25.3 (2010). DOI: 10.1017/S0269888910000159.

[14] Lukáš Chrpa and Thomas Leo McCluskey. "On exploiting structures of classical planning problems: Generalizing entanglements". In: *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI)*. 2012. DOI: 10.3233/978-1-61499-098-7-240.

[15] Lukáš Chrpa, Mauro Vallati, and Thomas Leo McCluskey. "MUM: A Technique for Maximising the Utility of Macro-operators by Constrained Generation and Use". In: *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS)*. 2014.

[16] Jens Claßen, Yuxiao Hu, and Gerhard Lakemeyer. "A situation-calculus semantics for an expressive fragment of PDDL". In: *Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI)* (2007).

[17] Jens Claßen and Gerhard Lakemeyer. "A Semantics for ADL as Progression in the Situation Calculus". In: *Proceedings of the Eleventh Workshop on Nonmonotonic Reasoning (NMR)*. 2006.

[18] Andrew Coles, Maria Fox, and Amanda Smith. "Online Identification of Useful Macro-Actions for Planning". In: *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*. 2007.

[19] Andrew Coles and Amanda Smith. "Marvin: A heuristic search planner with online macro-action learning". In: *Journal of Artificial Intelligence Research (JAIR)* 28 (2007). DOI: 10.1613/jair.2077.

[20] Alain Colmerauer. "Metamorphosis Grammars". In: *Natural Language Communication with Computers*. Ed. by Leonard Bolc. Vol. 63. Springer, 1978, pp. 133–188.

[21] C Dawson and L Siklóssy. "The Role of Preprocessing in Problem Solving Systems". In: *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI)* (1977).

[22] Giuseppe De Giacomo et al. "IndiGolog: A high-level programming language for embedded reasoning agents". In: *Multi-Agent Programming*. 2009. DOI: 10.1007/978-0-387-89299-3.

[23] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Communications of the ACM* 51.1 (2008). DOI: 10.1145/1327452.1327492.

[24] Kutluhan Erol, Dana S Nau, and V S Subrahmanian. "Complexity, Decidability and Undecidability Results for Domain-Independent Planning". In: *Artificial Intelligence* 76 (1995), pp. 75–88.

[25] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. "Learning and executing generalized robot plans". In: *Artificial Intelligence* 3 (1972). DOI: 10.1016/0004-3702(72)90051-3.

[26] Maria Fox and Derek Long. "PDDL+: Modelling Continuous Time-dependent Effects". In: *3rd International NASA Workshop on Planning and Scheduling for Space* (2002).

[27] Maria Fox and Derek Long. "PDDL2.1: An extension to PDDL for expressing temporal planning domains". In: *Journal of Artificial Intelligence Research (JAIR)* 20 (2003). DOI: 10.1613/jair.1129.

[28] Alfonso Gerevini and Derek Long. *Plan Constraints and Preferences in PDDL3*. Tech. rep. 2005.

[29] Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. "Planning through stochastic local search and temporal action graphs in LPG". In: *Journal of Artificial Intelligence Research (JAIR)* 20 (2003). DOI: 10.1613/jair.1183.

[30] Alfonso Gerevini et al. "Combining Domain-Independent Planning and HTN Planning: The Duet Planner". In: *Proceedings of the 23rd Conference on Artificial Intelligence (AAAI)*. 2008. DOI: 10.3233/978-1-58603-891-5-573.

[31] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning - theory and practice*. Morgan Kaufmann Publishers, 2004.

[32] Malte Helmert. "The Fast Downward Planning System". In: *Journal of Artificial Intelligence Research (JAIR)* 26 (2006). DOI: 10.1613/jair.1705.

[33] Lorin Hochstein. *Ansible: Up and Running*. 2014, p. 334.

[34] Jörg Hoffmann and Bernhard Nebel. "The FF planning system: Fast plan generation through heuristic search". In: *Journal of Artificial Intelligence Research (JAIR)* 14 (2001). DOI: 10.1613/jair.855.

[35] Till Hofmann. "Continual Planning and Execution Monitoring in the Agent Language Golog on a Mobile Robot". Master's Thesis. RWTH Aachen University, 2015.

[36] Till Hofmann, Tim Niemueller, and Gerhard Lakemeyer. "Initial Results on Generating Macro Actions from a Plan Database for Planning on Mobile Robots". In: *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS)*. to appear, 2017.

[37] Till Hofmann et al. "Continual Planning in Golog". In: *Proceedings of the 30th Conference on Artificial Intelligence (AAAI)*. 2016.

[38] He Jifeng, Xiaoshan Li, and Zhiming Liu. "Component-Based Software Engineering". In: *Proceedings of the International Colloquium on Theoretical Aspects of Computing (ICTAC)*. 2005. DOI: 10.1007/11560647_5.

[39] J.L. Jones. "Robots at the tipping point: the road to iRobot Roomba". In: *IEEE Robotics & Automation Magazine* 13.1 (2006). DOI: 10.1109/MRA.2006.1598056.

[40] Ann Mary Joy. "Performance comparison between Linux containers and virtual machines". In: *Proceedings of the International Conference on Advances in Computer Engineering and Applications (ICACEA)* (2015), pp. 342–346. DOI: 10.1109/ICACEA.2015.7164727.

[41] Hiroaki Kitano et al. "RoboCup: The Robot World Cup Initiative". In: *Proceedings of the 1st International Conference on Autonomous Agents*. 1997.

[42] Gerhard Lakemeyer and Hector Levesque. "Semantics for a Useful Fragment of the Situation Calculus". In: *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*. 2005, pp. 490–496.

[43] Gerhard Lakemeyer and Hector J Levesque. "Situations, Si! Situation Terms, No!" In: *Proceedings of the 9th Conference on Principles of Knowledge Representation and Reasoning (KR)*. 2004, pp. 516–526.

[44] Hector J Levesque et al. "Golog: a Logic Programming Language for Dynamic Domains". In: *Journal of Logic Programming* 1066.96 (1997).

[45] Hector Levesque, Fiora Pirri, and Ray Reiter. "Foundations for the Situation Calculus". In: *Linköping Electronic Articles in Computer and Information Science* 3.18 (1998), p. 18.

[46] Matthias Löbach. "Centralized Global Task Planning with Temporal Aspects on a Group of Mobile Robots in the RoboCup Logistics League". Master's thesis (to appear). RWTH Aachen University, 2017.

[47] John McCarthy. *Situations, actions, and causal laws*. Tech. rep. Stanford University, 1963.

[48] Drew McDermott et al. *PDDL - The Planning Domain Definition Language*. 1998.

[49] Merriam-Webster. *Quartile*. 2017. URL: https://www.merriam-webster.com/dictionary/quartile (visited on 03/02/2017).

[50] G. Michalos et al. "Automotive assembly technologies review: challenges and outlook for a flexible and adaptive approach". In: *CIRP Journal of Manufacturing Science and Technology* 2.2 (2010), pp. 81–91. DOI: 10.1016/j.cirpj.2009.12.001.

[51] *MongoDB MapReduce*. URL: https://docs.mongodb.com/manual/reference/method/db.collection.mapReduce/ (visited on 03/08/2017).

[52] Dana Nau et al. "SHOP2 : An HTN Planning System". In: *Journal of Artificial Intelligence Research (JAIR)* 20 (2003).

[53] M.A. Hakim Newton et al. "Learning macro-actions for arbitrary planners and domains". In: *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*. 2007.

[54] M.A. Hakim Newton et al. "Wizard: Compiled Macro-Actions for Planner-Domain Pairs". In: *Booklet for the 6th International Planning Competition Learning Track*. 2008.

[55] Tim Niemueller, Alexander Ferrein, and Gerhard Lakemeyer. "A Lua-based Behavior Engine for controlling the humanoid robot Nao". In: *RoboCup Symposium 2010*. 2010. DOI: 10.1007/978-3-642-11876-0_21.

[56] Tim Niemueller, Gerhard Lakemeyer, and Alexander Ferrein. "Incremental task-level reasoning in a competitive factory automation scenario". In: *AAAI Spring Symposium: Designing Intelligent Robots*. 2013.

[57] Tim Niemueller, Gerhard Lakemeyer, and Alexander Ferrein. "The RoboCup Logistics League as a Benchmark for Planning in Robotics". In: *Proceedings of the ICAPS Workshop on Planning in Robotics*. 2015.

[58] Tim Niemueller, Gerhard Lakemeyer, and Siddhartha Srinivasa. "A Generic Robot Database and its Application in Fault Analysis and Performance Evaluation". In: *IEEE International Conference on Intelligent Robots and Systems (IROS)*. 2012. DOI: 10.1109/IROS.2012.6385940.

[59] Tim Niemueller, Sebastian Reuter, and Alexander Ferrein. "Fawkes for the RoboCup Logistics League". In: *RoboCup Symposium 2015*. 2015.

[60] Tim Niemueller et al. "Cyber-Physical System Intelligence". In: *Industrial Internet of Things*. Ed. by Sabina Jeschke et al. Springer, 2017, pp. 447–472.

[61] Tim Niemueller et al. "Decisive Factors for the Success of the Carologistics RoboCup Team in the RoboCup Logistics League 2014". In: *RoboCup Symposium 2014*. 2014.

[62] Tim Niemueller et al. "Design principles of the component-based robot software framework Fawkes". In: *Proceedings of the International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*. 2010. DOI: 10.1007/978-3-642-17319-6-29.

[63] Tim Niemueller et al. "Knowledge-Based Instrumentation and Control for Competitive Industry-Inspired Robotic Domains". In: *KI - Künstliche Intelligenz* 30.3-4 (2016), pp. 289–299. DOI: 10.1007/s13218-016-0438-8.

[64] Tim Niemueller et al. "RoboCup Logistics League Sponsored by Festo: A Competitive Factory Automation Testbed". In: *RoboCup Symposium 2013*. 2013. DOI: 10.1007/978-3-662-44468-9_30.

[65] Nils J. Nilsson and Richard E. Fikes. "STRIPS: A new approach to the application of theorem proving to problem solving". In: *Artificial intelligence* 2 (1972). DOI: 10.1016/0004-3702(71)90010-5.

[66] Edwin Pednault. "ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus." In: *KR* (1989).

[67] Fernando C N Pereira and David H D Warren. "Definite clause grammars for language analysis - A survey of the formalism and a comparison with augmented transition networks". In: *Artificial Intelligence* 13.3 (1980), pp. 231–278. DOI: 10.1016/0004-3702(80)90003-X.

[68] Fiora Pirri and Raymond Reiter. "Some contributions to the metatheory of the Situation Calculus". In: *Journal of the ACM* (1999).

[69] Morgan Quigley et al. "ROS: An open-source robot operating system". In: *ICRA Workshop on Open Source Software*. 2009.

[70] Raymond Reiter. *Knowledge in action: logical foundations for specifying and implementing dynamical systems*. MIT Press, 2001.

[71] Raymond Reiter. "The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression". In: *Artificial intelligence and mathematical theory of computation* (1991), pp. 359–380. DOI: 10.1016/B978-0-12-450010-5.50026-8.

[72] Stuart Jonathan Russell et al. *Artificial intelligence: a modern approach*. 3rd. Prentice Hall, 2010.

[73] Mathijs Jeroen Scheepers. "Virtualization and Containerization of Application Infrastructure : A Comparison". In: *21st Twente Student Conference on IT* (2014), pp. 1–7.

[74] James Turnbull. *The Docker Book: Containerization is the new virtualization*. 2017.

[75] Mauro Vallati et al. "The 2014 International Planning Competition: Progress and Trends". In: *AI Magazine* (2015).

[76] *What is Docker?* URL: https://www.docker.com/what-docker (visited on 02/12/2017).

[77] *What is Kubernetes?* URL: https://kubernetes.io/docs/whatisk8s/ (visited on 02/12/2017).

[78] Robert M. Wygant. "CLIPS — A powerful development and delivery expert system tool". In: *Computers & Industrial Engineering* 17.1-4 (1989). DOI: 10.1016/0360-8352(89)90121-6.

[79] Benjamin Zarrieß and Jens Claßen. "Verifying CTL* properties of GOLOG programs over local-effect actions". In: *Proceedings of the Twenty-First European Conference on Artificial Intelligence (ECAI 2014)*. 2014, pp. 939–944. DOI: 10.3233/978-1-61499-419-0-939.

[80] Qi Zhang, Lu Cheng, and Raouf Boutaba. "Cloud computing: State-of-the-art and research challenges". In: *Journal of Internet Services and Applications* 1.1 (2010), pp. 7–18. DOI: 10.1007/s13174-010-0007-6.

[81] Frederik Zwilling. "A Document-Oriented Robot Memory for Knowledge Sharing and Hybrid Reasoning on Mobile Robots". Master's thesis (to appear). RWTH Aachen University, 2017.

[82] Daniel Zwillinger and Stephen Kokoska. *Standard Probability and Statistics Tables and Formulae*. Chapman & Hall/CRC, 2000.