DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Verified Grounding of PDDL Tasks Using Reachability Analysis

Maximilian Vollath

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Verified Grounding of PDDL Tasks Using Reachability Analysis

Verifiziertes Grounding von PDDL-Planungsproblemen durch Erreichbarkeitsanalyse

Author:	Maximilian Vollath
Supervisor:	Prof. Tobias Nipkow
Advisor:	Mohammad Abdulaziz
Submission Date:	June 2nd, 2025

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, June 2nd, 2025 Maximilian Vollath

Acknowledgments

I would like to express my utmost gratitude to my supervisor, Mohammad Abdulaziz, for his remarkable patience and guidance throughout this project. Our discussions were always extremely pleasant, because he often understood my ideas before I had even formed coherent sentences to explain them.

My heartfelt thanks go to my partner, Nina, for always having my back and supporting when I was stressed.

I'm also grateful for my good friend Ricardo, who pushed me forward when I wasn't stressed enough.

Thank you very much to my father for proofreading this thesis.

Finally, I thank everyone else who kindly endured my endless rambling about formal verification.

Abstract

Grounding classical planning tasks is a fundamental yet complex problem in automated planning. The *Fast Downward* planning system is known for its effective approach to grounding first-order PDDL tasks into the finite-domain representation SAS$^+$, employing reachability analysis to reduce the necessary number of variables and operators for problem solving. Recent advancements by Corrêa et al. have improved reachability analysis by increasing the computational efficiency.

This thesis presents a formally verified implementation of this grounding procedure within the Isabelle proof assistant. The implementation closely follows the methodologies applied in *Fast Downward* and includes a an algorithm for reachability analysis which is related to solving Datalog problems.

The step of invariant synthesis is explicitly excluded, resulting in an output in the propositional planning language STRIPS rather than the finite-domain language SAS$^+$.

Combined with a parser and a verified SAT solver, this project forms a complete pipeline for solving PDDL tasks.

Contents

1 Introduction

Automated planning is a critical domain in artificial intelligence that deals with the generation of action sequences to achieve specified goals from a given state. Planning problems can be expressed in various languages, among which the Planning Domain Definition Language (PDDL), SAS$^+$, and STRIPS are particularly noteworthy.

PDDL is a higher-level, first-order language that includes concepts such as types, and parameterized actions and predicates. It is concise, human-readable, and widely used to define planning problems. Due to its complexity, however, PDDL is not well-suited suited as direct input for many solvers.

STRIPS, on the other hand, is a propositional planning language, in which actions are not parameterized and the state is represented by binary variables. This simplicity makes STRIPS more feasible as input for many solving algorithms, but tasks are rarely written directly in STRIPS.

In contrast to STRIPS, SAS$^+$ uses finite-domain variables which can take on more than two discrete values. It is likewise suitable for solving algorithms but can often express a task with fewer variables.

The process of transforming a task from a first-order representation—also referred to as **lifted**—into a propositional one is called **grounding**. To solve a PDDL task, it is typically first grounded into an equivalent STRIPS or SAS$^+$ instance and then passed into a solver. A naive grounder, which simply instantiates every action and predicate with all possible parameter combinations is easy to implement, but leads to an exponential growth in ground variables and actions. This renders it infeasible for even modestly sized problems.

A well-designed grounder must therefore limit the number of ground variables and actions it produces. This is commonly achieved by employing **reachability analysis**, in order to omit impossible predicate and action instantiations from the grounded task. Since computing the exact set of reachable states or satisfiable ground actions is often infeasible, an overapproximation is commonly used.

A common grounding procedure was introduced by Helmert (2009) and implemented in the *Fast Downward* planning system. For a relaxed version of the input PDDL task, the exact set of satisfiable action and predicate instances is computed using Datalog. In addition, Helmert employs a procedure called **invariant synthesis** to produce finite-domain variables instead of binary ones, reducing the number of variables in the

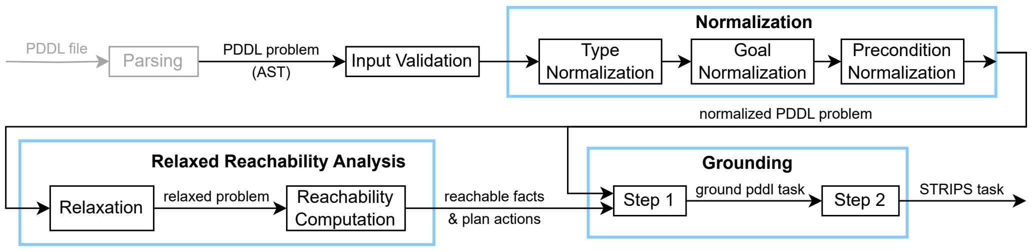

Figure 1.1: A diagram of the grounding process pipeline. A PDDL file is first parsed into an abstract syntax tree representation of PDDL. The well-formedness of this task is validated and it is normalized. Relaxed reachability analysis is employed to compute a superset of all reachable actions and facts, which are then passed to the grounding steps to produce a STRIPS instance. Parsing is not included in this project.

grounded task. It is important to note that invariant synthesis is orthogonal to the grounding process and can thus be applied to any STRIPS instance [Hel09].

Corrêa et al. (2023) proposed a refinement to reachability analysis by dividing it into two phases. Tree decomposition is applied to transform the Datalog program that is used to determine the set of reachable states into a more efficient form. Then, the set of satisfiable action instances is computed separately.

While implementations of these methods are publicly available, they are not formally verified. Such a verification is the aim of this project. The automated theorem prover Isabelle facilitates this process. In Isabelle, formulas and even complete programs can be expressed in a formal language. Within this framework, properties such as appropriately defined correctness specifications can be formally proven. However, this requires re-implementing the entire program in a functional programming language similar to Haskell. Note that the correctness of an algorithm does not necessarily indicate an efficient implementation.

To my knowledge, no formally verified grounder currently exists. However, there are verified plan checkers for our languages of interest, and even a verified solver for STRIPS. This raises the question: Why verify a grounder or solver? Indeed, given a PDDL planning task, one could input it into unverified grounding and solving tools, and then verify the result with a verified plan checker. The key difference is that if the solver finds no solution, there is no guarantee that it does not exist. Only if the entire procedure is verified, it can be confidently asserted that there is no solution. Additionally, other properties of the planning task could potentially be of interest, such as the optimality of solutions.

The contributions of this work include the formal verification of Helmert's grounding algorithm, implemented in Isabelle. Additionally, it includes an algorithm for reachability analysis that is equivalent to solving Datalog problems, a verified algorithm to compute the transitive closure of a relation, and a space-efficient enumeration of the Cartesian product of input lists, filtering out combinations via a predicate. The full source code of the implementation is available at: `https://github.com/MVollath/Isabelle-PDDL-Grounding`

This thesis is structured analogously to the grounding pipeline as depicted in figure 1.1. Following the introduction, a background section defines the relevant planning languages and Datalog, and formally describes the grounding procedures by Helmert and Corrêa et al. This is followed by an overview of related works. The following chapter on the implementation is divided into sections that correspond to the individual steps of the grounding pipeline: normalization, relaxation, reachability computation, and grounding. Each step includes a detailed description of the implementation and a summary of the major correctness theorems. The conclusion lays out the next steps for the full formal verification of the entire pipeline.

2 Background

The following section introduces the formalisms and algorithms that this work builds upon. It covers the syntax and semantics of PDDL, STRIPS, and Datalog, defines tree decompositions, and details the grounding techniques developed by Helmert and refined by Corrêa et al.

2.1 PDDL

The Planning Domain Definition Language (PDDL) is the de facto standard artificial intelligence planning language and is commonly used in planning competitions. A PDDL task defines variables with which parameterized predicates and actions can be instantiated. PDDL is commonly referred to as a first-order planning language because of the use of predicate logic and first-order formulas. There are multiple levels of PDDL and many optional features [PWiki]. We are concerned with the abstract syntax specified in *AI Planning Languages Semantics*by Abdulaziz and Lammich [AL20], which our implementation builds upon. It is very similar to the syntax used by Helmert but differs in a few ways which will be highlighted.

PDDL tasks are commonly divided into domain and problem: the domain describes the general planning task, and the problem defines the specific instance.

2.1.1 Abstract Syntax

A PDDL domain is a tuple $\langle \mathcal{T}, \mathcal{P}, \mathcal{C}, \mathcal{A} \rangle$.

The set of *primitive types* and their hierarchy are defined by the directed graph $\mathcal{T}$. Every node corresponds to a type and the edge (a, b) signifies that a directly inherits b. [AL20] supports Either types: A *type* is a list of primitive type alternatives $[a \mid b \mid \ldots]$. Intuitively, this means that certain variables can be instantiated by differently typed objects. In Helmert, types aren't explicitly part of the formalism but it is mentioned that they are removed in the normalization step.

$\mathcal{C}$ is the set of constants; a set of symbols that are assigned a type through the function $type(\cdot)$. Although [AL20] allow constants to have multiple types, that complicates normalization. We thus restrict them to singular types instead, which is consistent with

Helmert and Corrêa et al. For simplicity, we write $type(o) = t$ instead of $type(o) = [t]$ for constants.

$\mathcal{P}$ is the set of predicate symbols. Each predicate symbol p additionally has a corresponding signature $sig(p)$ expressed as a list of types. A predicate can be instantiated with an appropriate list of constants (or variables, see below) to form a binary variable called atom. A ground atom is an atom instantiated purely with constants. Equality and inequality can be modeled by predicates. However, this formalism considers $=$ a to be a built-in binary predicate.

A state is a set of ground atoms and thus defines a valuation of these binary variables: true if the atom is contained in the state, false if it isn't. The equality predicate is not explicitly part of a state.

$\mathcal{A}$ is the set of actions. An action A consists of a parameter list $params(A)$, a precondition $pre(A)$ and an effect, which itself consists of two sets of atoms $adds(A)$ and $dels(A)$. Like states, effects cannot reference the predicate $=$. The parameter list defines variables and assigns a type to each one of them. The precondition is a formula over atoms instantiated with domain constants and/or parameter variables. The effect's atoms are likewise instantiated with constants and/or variables.

Helmert allows preconditions to contain first-order quantifiers, but [AL20] restricts them to propositional formulas. In addition, the former formalism supports nested effect preconditions, i.e. formulas that restrict under which circumstances certain parts of an effect are actually applied. In other formalisms, action parameters are often implicitly defined via the open variables in the precondition and the effect. However, we consider them as explicit attributes of an action due to ambiguities that may arise from Either types or the type hierarchy.

A PDDL problem is a tuple $\langle \mathcal{D}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$. $\mathcal{D}$ simply refers to a domain and $\mathcal{O}$ (O for objects) is an additional set of constants, like $\mathcal{C}$. Whether a constant is defined on the domain- or the problem-level does not have any effect on the semantics. Unless stated otherwise, the term *constants* refers to the union $\mathcal{C} \cup \mathcal{O}$.

The initial state $\mathcal{I}$ is a state, and the goal $\mathcal{G}$ is a propositional formula over ground atoms.

Helmert does not distinguish between domain and problem, but that has no bearing on the semantics. However, Helmert allows for the definition of axioms, even though Corrêa et al. later omit them. Axioms are sets of rules that define derived predicate atoms based on which atoms are true in a given state. From a basic state, these rules are sequentially applied to determine the values of derived predicates, which can in turn be referenced by action preconditions or the goal. Axioms must adhere to specific restrictions in order to ensure that their evaluation cannot cause ambiguities. For a more detailed explanation, we refer the interested reader to [THN05].

2.1.2 Well-Formedness

The concept of well-formedness in PDDL tasks plays a pivotal role in ensuring that the tasks are logically coherent and importantly, their execution is unambiguous and well-defined. Here, these criteria are outlined, with special attention to the characteristics of the type system.

The type graph defines the inheritance relationships: a is a subtype of b if and only if $(a, b) \in \mathcal{T}^*$.

The type graph itself must be well-formed. Typically, it is restricted to a directed tree. However, rather unconventionally, Abdulaziz and Lammich's formalism allows for multiple inheritance, disconnected subgraphs and even cycles. The only restrictions are that the default type *object* is represented and that every type aside from *object* has at least one parent type.

A well-formed PDDL task must strictly adhere to type consistency. All atoms within the task are instantiated with constants or variables that match the expected predicate signatures w.r.t. the type graph. The following example should illustrate this rather complex system:

Consider the type graph given in figure 2.2. The types `car`, `train` and `parcel` directly inherit from `movable`, and `road-rail` inherits both `car` and `train`. Then consider a variable x with $\mathit{type}(x) = [\texttt{car} \mid \texttt{parcel}]$. The constants a, b, and c have the respective types $[\texttt{car}]$, $[\texttt{road} - \texttt{rail}]$, and $[\texttt{car} \mid \texttt{train}]$. We can instantiate x with a, since `car` is obviously `car` or `parcel`, with b, because `road-rail` is *both* `car` and `train`, but not with c, since c could potentially be `train` which is *neither* `car` nor `parcel`.

.

Finally, the domain must define every action, predicate, and type utilized within the task. There is sometimes the additional restriction that add and delete effects may not contradict each other. However, this is not necessary here as the add effect atoms are simply defined to take precedence.

When transforming PDDL tasks, it is crucial that, given a well-formed input, the output be well-formed as well. This, of course, is covered by the proof.

2.1.3 Semantics

The semantics of a PDDL task dictate the procedures for interpreting and executing actions. This section details the mechanisms behind action instantiation, grounding, and the conditions for plan execution.

A single step is specified by a *plan action*: an action and a corresponding argument list of constants from $\mathcal{C} \cup \mathcal{O}$. A plan action is well-formed if it references a defined action and the constants match its signature. A ground action is produced when a plan action is instantiated by substituting the variables in the precondition and the effect by the corresponding constants from the list. We use the term plan action to also refer to the corresponding ground action.

A plan action π is enabled in a state s if s satisfies the instantiated precondition, notation $s \Vdash pre(\pi)$. Only then is it legal to execute it. Upon execution, the ground atoms of the delete effect are removed from s and then the ground add effect is added. This way, the add effect takes precedence in case of a contradiction.

A plan is simply a sequence of plan actions that transforms the initial state $\mathcal{I}$ into a final state s. A plan is well-formed if each plan action is well-formed and enabled at its corresponding point in the execution sequence and the plan is valid if, additionally, s satisfies $\mathcal{G}$.

If some well-formed plan exists for which the resulting state contains a ground atom a, then a is *reachable*. In particular, every ground atom in I is achievable. A plan action π is reachable if some well-formed plan results in a state s in which $pre(\pi)$ is satisfied. Equivalently, π is reachable if it is part of some well-formed plan.

The certification of a PDDL task being well-formed and the execution semantics including plan validation are already implemented and proven in Isabelle by [AL20].

2.1.4 Example

This subsection introduces an example PDDL task used as a running example throughout this document. The scenario involves cities connected by roads and train tracks, various types of vehicles including cars, a train, a road-rail vehicle, and parcels that need to be transported. The syntax of PDDL is rather intuitive and isn't defined explicitly here. For the sake of consistency, the requirements are left in, but can be ignored for the purpose of this paper.

The domain in figure 2.1 defines the types, predicates, and actions relevant to the transportation task. The setting is a classic delivery task. There are cities connected by roads and rails, cars that can drive on roads, trains that run on tracks, road-rail vehicles that can traverse both, and parcels that can be transported by the aforementioned vehicles. Due to the pivotal role of types in the grounding algorithm, a rather complex type hierarchy was designed for this example, as shown in figure 2.2.

The problem (figure 2.3) specifies the objects, the initial configuration and the goal for the planning task. Crucially, the goal is to deliver the parcels to their respective destinations.

The graph in figure 2.4 illustrates the layout. One possible valid solution is the following plan:

```
(define
    (domain transport-domain)
    (:requirements :strips :typing :disjunctive_preconditions :equality)
    (:types
        city - object
        movable - object
        parcel - movable
        vehicle - movable
        car - vehicle
        train - vehicle
        road-rail - (multi car train)
    )
    (:predicates
        (road ?c1 - city ?c2 - city)
        (rail ?c1 - city ?c2 - city)
        (at ?m ?c)
        (in ?p ?v)
    )
    (:action DRIVE
        :parameters (?c - car ?from - city ?to - city)
        :precondition (and (at ?c ?from) (or (road ?from ?to) (road ?to ?from)))
        :effect (and (at ?c ?to) (not (at ?c ?from)))
    )
    (:action CHOOCHOO
        :parameters (?t - train ?from - city ?to - city)
        :precondition (and (at ?t ?from) (or (rail ?from ?to) (rail ?to ?from)))
        :effect (and (at ?t ?to) (not (at ?t ?from)))
    )
    (:action LOAD
        :parameters (?p - parcel ?v - (either car train) ?where - city)
        :precondition (and (at ?p ?where) (at ?v ?where))
        :effect (and (in ?p ?v) (not (at ?p ?where)))
    )
    (:action UNLOAD
        :parameters (?p - parcel ?v - vehicle ?where - city)
        :precondition (and (in ?p ?v) (at ?v ?where))
        :effect (and (at ?p ?where) (not (in ?p ?v)))
    )
)
```

Figure 2.1: PDDL domain specification of the exemplary transportation task. Multiple inheritance is supported with help of the keyword `multi`.

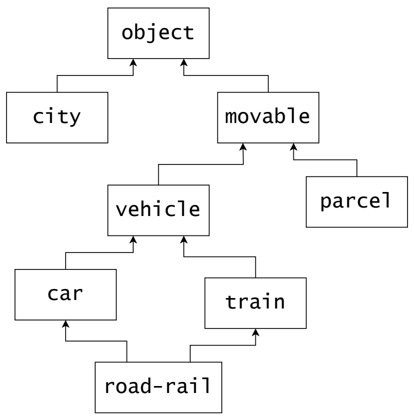

Figure 2.2: The transportation domain's type hierarchy. `object` is a mandatory type. Although unconventional, multiple inheritance is demonstrated by `road-rail` inheriting directly from both `car` and `train`.

1. `DRIVE c2 E D`
2. `LOAD p2 c2 D`
3. `DRIVE c2 D C`
4. `UNLOAD p2 c2 C`
5. `LOAD c2 t C`
6. `CHOOCHOO t C B`
7. `UNLOAD p2 t B`
8. `LOAD p1 v B`
9. `CHOOCHOO v B C`
10. `DRIVE v C E`
11. `UNLOAD p1 v E`

2.2 Propositional STRIPS

The STRIPS (Stanford Research Institute Problem Solver) format is a framework for specifying planning tasks, well-known for its simplicity and effectiveness in representing actions and states in a planning problem. STRIPS can be seen as a grounded version of PDDL, with some distinct specifications. The following formalism is implemented in *Verified SAT-Based AI Planning*[AK20], including proofs for determining well-formedness and plan validation.

STRIPS represents actions, states, and goals using a simplified set of constructs compared to PDDL. A STRIPS task $\langle \mathcal{V}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$ consists of the following:

- $\mathcal{V}$, the set of defined binary variables,
- $\mathcal{O}$, the set of operators, each consisting of a precondition, an add effect and a delete effect. Each component is a set of variables.

```
(define
    (problem transport1)
    (:domain transport-domain)
    (:objects
        A - city
        B - city
        C - city
        D - city
        E - city
        c1 - car
        c2 - car
        t - train
        v - road-rail
        p1 - parcel
        p2 - parcel
    )
    (:init
        (road A B)
        (road C E)
        (road E D)
        (road C D)
        (rails B C)
        (at c1 A)
        (at c B)
        (at p1 B)
        (at t C)
        (at p2 D)
        (at c2 E)
    )
    (:goal
        (and
            (at p1 E)
            (at p2 B)
        )
    )
)
```

Figure 2.3: PDDL problem specification of the exemplary transportation task.

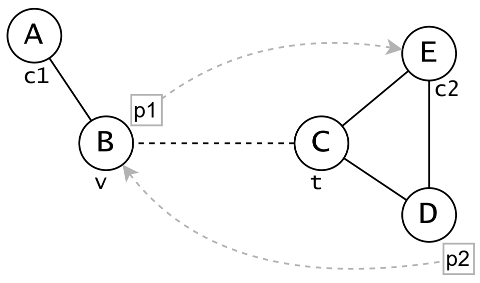

Figure 2.4: The layout the exemplary PDDL task. Cities `A` through `E` are connected via roads (solid edges) and train tracks (dashed edge). The goal is to deliver parcels `p1` to `B` and `p2` to `E`, using the cars `c1` and `c2`, the train `t` and the road-rail vehicle `v`.

- $\mathcal{I}$, the initial state; a valuation over $\mathcal{V}$.
- $\mathcal{G}$, the goal state.

2.2.1 Well-Formedness

A STRIPS task is well-formed if all variables referenced by operators, $\mathcal{I}$ or $\mathcal{G}$ are defined in $\mathcal{V}$. Additionally, and in contrast to PDDL, add and delete effects must be disjoint.

2.2.2 Semantics

The semantics of STRIPS are defined through the execution of operators. An operator o is applicable in a state s if every variable in its precondition is assigned true by s. The result of executing o is a modification of s where the variables in the add effect of o are assigned to true, and those in the delete effect to false. Similarly to PDDL, a plan is a sequence of operators that is valid if every operator is enabled at the moment it is executed, and the final state assigns true to all variables in $\mathcal{G}$.

STRIPS differs from SAS$^+$primarily in how it handles the representation of variables and domain values. SAS$^+$is a finite-domain representation, where each variable can take on any finite number of values. SAS$^+$also supports axioms, albeit with a slightly different set of restrictions. [Bäc92]

```
Variables:
    at-p1-A, at-p1-B, ..., at-p1-E, ... at-p2-E,
    at-c1-A, at-c1-B,
    at-c2-C, at-c2-D, at-c2-E,
    at-t-B, at-t-C,
    at-v-A, at-v-B, ..., at-v-E
Init:
    at-c1-A, at-c-B, at-p1-B, at-t-C, at-p2-D, at-c2-E
Goal:
    at-p1-E, at-p2-B
Operator drive-c1-A-B:
    PRE: at-c1-A            ADD: at-c1-B    DEL: at-c1-A
...
Operator choochoo-t-B-C:
    PRE: at-t-B             ADD: at-t-C     DEL: at-t-B
...
Operator load-p1-v-D:
    PRE: at-p1-D, at-v-D    ADD: in-p1-v    DEL: at-p1-D
...
Operator unload-p1-t-C:
    PRE: in-p1-t, at-t-C    ADD: at-p1-C    DEL: in-p1-t
...
```

Figure 2.5: A STRIPS task equivalent to the PDDL task defined in section 2.1.4. Most operators and variables are omitted, because there would be too many.

2.2.3 Example

Figure 2.5 is a manually optimized STRIPS representation of the PDDL task defined above. Note that since e.g. the train `t` can only run between cities `B` and `C`, those are its only necessary positional variables.

2.3 Datalog

Datalog is a declarative logic programming language consisting of parameterized facts and rules. It is commonly used for querying relational databases, but can also be employed to solve certain logical problems. The following formalization closely follows the one by Schlichtkrull et al. [SRN24].

In the context of planning and problem-solving, a Datalog program is commonly defined by a tuple $\langle \mathcal{F}, \mathcal{R} \rangle$. Similarly to PDDL, an atom consists of a predicate symbol and a list of arguments. An argument in a lifted atom can be either a constant or a variable, but ground atoms only contain constants. A lifted atom can be grounded with a valuation σ that assigns each variable to a constant. Both the set of predicate symbols and the set of constants are implicitly defined. The analog of a state is the model M, a set of ground atoms.

The *facts*, the set of ground atoms $\mathcal{F}$, represent known truths and effectively form the initial state of the logic program.

The set $\mathcal{R}$ defines conjunctive rules that allow the derivation of new facts from existing ones. A Datalog rule has the form

$$H \leftarrow B_1 \land B_2 \land \ldots \land B_n.$$

Here, H is the head atom and literals B_1 through B_m make up the body. A literal is a positive or negative lifted atom, or the equality or inequality of atoms and constants. The rule states that, for every valuation σ, if all B_i are satisfied in a solution model M, then H is true and must be in M. Since an empty body is always satisfied, such rules are often used instead of an explicit set $\mathcal{F}$ to encode the initially known facts.

A rule is called *safe* if all variables present in the head occur in positive atoms in the body. Unsafe rules effectively have a $\forall$-qualified head and can lead to infinitely large solutions, unless the domain of constants is explicitly restricted.

A Datalog program is *stratified* if its predicates can be organized into ordered *strata*. If the head atom's predicate is in the stratum n, then negative literals in the body can only reference atoms from strata below n. This avoids direct or indirect recursion through negation. This restriction is critical for ensuring the consistency and predictability of the logic program.

A key property of stratified Datalog is that is has a unique stable model, known as the canonical model, which can be constructed iteratively: Starting with the set of facts $\mathcal{F}$, one can iteratively apply the suitably instantiated rules to derive new facts until no more rules can be applied. This process is repeated for all strata, from lowest to highest. The outcome is deterministic and terminates when a fixed point is reached. The absence of negative dependencies in rule bodies ensures that the order of rule application does not affect the final set of facts. Thus, the canonical model is uniquely defined for any given set of facts and rules [CGT+89]. We only consider a special case of stratified programs: prohibiting the use of negative atoms altogether.

In *Fast Downward*, Datalog is used to derive the set of achievable ground atoms and applicable plan actions. Datalog predicates and variables rather intuitively correlate to ground PDDL atoms. Section 2.5.2 describes how additional Datalog predicates are constructed to encode the applicability of actions.

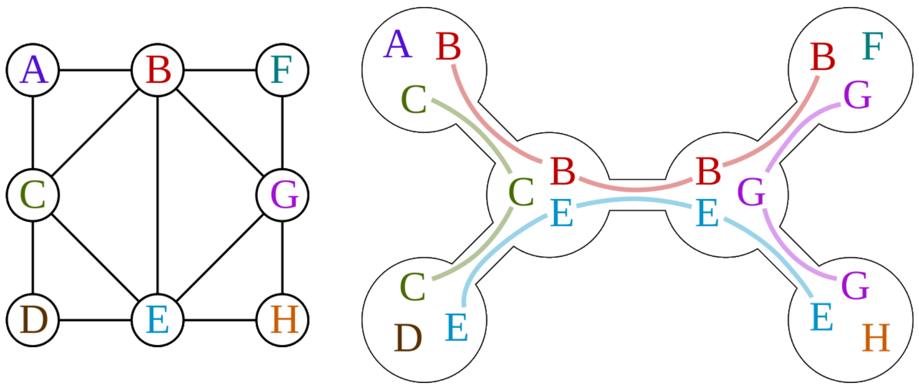

Figure 2.6: A graph (left) and its corresponding optimal tree decomposition (right). By David Eppstein [Epp07].

2.3.1 Primal Graph

For a Datalog rule r (or an atom a), let $vars(r)$ (or $vars(a)$) be the set of variables that occur in it.

The primal graph of a Datalog rule r is the graph $G_r = \langle V(G_r), E(G_r)\rangle$, where $V(G_r) = vars(r)$ and there is an edge for every pair of variables that jointly occur in an atom of the rule.

As is the case with many graph-based problems, a lower tree width (see below) generally indicates a simpler structure, which in turn simplifies the computation of the Datalog program and thus simplifies grounding [MW12].

2.4 Tree Decomposition

A **tree decomposition** of a graph $G = (V, E)$ is a tree T whose nodes are subsets $X_1, \dots, X_n \subseteq V$, called **bags**, satisfying the following conditions:

1. Each vertex of G appears in at least one bag. That is, $\bigcup_i X_i = V$.

2. For every vertex $v \in V$, the set of bags containing v forms a connected subtree of T.

3. For every edge $(v, w) \in E$, there exists a bag X_i such that $\{v, w\} \subseteq X_i$.

The **width** of a tree decomposition is the size of its largest bag minus one. The **treewidth** $\text{tw}(G)$ of a graph G is the smallest such width achievable by any tree decomposition of G.

A common task is to compute the tree decomposition corresponding to the treewidth. Since this problem is NP-complete, approximating algorithms are often employed, such as the heuristic algorithm implemented in the *htd* library, that we used in an earlier proof of concept.

Figure 2.6 illustrates tree decompositions using an example.

2.5 Grounding Process by Helmert

The grounding process developed by Helmert [Hel09] is a key component of the *Fast Downward* planning system. This process involves several independent steps, each designed to simplify the structure of the planning task. The input is a PDDL task Π.

2.5.1 Normalization

The initial step is normalization, which transforms the input task into an equivalent one with a simplified structure. A normalized PDDL task satisfies the following properties:

- There is only one type: `object`. Every constant and variable is of type [`object`](written as ω); with exactly one primitive type.
- The goal formula is a conjunction of literals.
- All action preconditions are conjunctions of literals.

Helmert additionally places restrictions on the structure of axioms and conditional effects. However, these are not relevant here, because these features aren't present in *AI Planning Languages Semantics*.

For our formalism, normalization occurs in three steps. First, types are removed and action preconditions are modified to incorporate the logic previously handled by type constraints. Minor modifications to Helmert's method had to be made to support Either types. For each primitive type t, a corresponding unary predicate `is_t` is constructed. Then, the types of objects and constants are converted into atoms of $\mathcal{I}$, taking into account super types. For example, $type(\texttt{p1}) = \texttt{Parcel}$ yields atoms `is_Parcel(p1)`, `is_Movable(p1)` and `is_object(p1)`. Next, action preconditions are suitably amended to enforce type constraints of the parameters. For instance, for an action with parameters $[x, y]$ with $type(x) = [\texttt{City}]$, $type(y) = [\texttt{Car} \mid \texttt{Train}]$ and precondition φ, the new precondition is $\texttt{is\_City}(x) \wedge (\texttt{is\_Car}(x) \vee \texttt{is\_Train}(x)) \wedge \varphi$. Next, all types aside from *object* are removed alongside with all occurrences of types, in i.e. predicate signatures, action parameters, $\mathcal{C}$ and $\mathcal{O}$, which are replaced by [*object*]. So far, after type normalization, the semantics are unchanged (see section 16).

Modifying type normalization to support constants with Either types is more complex and is the reason we don't support them.

Goal normalization is comparably simple. In case $\mathcal{G}$ isn't already a conjunction, Helmert simulates it with axioms. Without support for axioms in our formalism, however, this is achieved with the help of an auxiliary action. This action has an empty parameter list, its precondition is the original goal formula, and its effect is the addition of a new goal predicate with an empty signature. This goal predicate in turn becomes the new goal. The intuition behind this is that any state that satisfies the previous goal formula, now enables the goal plan action π_G and upon its execution, the new goal is ultimately satisfied.

Precondition normalization ensures that all preconditions are pure conjunctions of (potentially negated) ground atoms. First, each action precondition is replaced by its disjunctive normal form (DNF). Then, the action is split across the disjunctions, i.e. replaced by multiple smaller actions, each a copy of the original but with one of the conjunctive clauses as precondition.

The resulting task Π_N is normalized and differs from a STRIPS task mostly in that predicates and actions are still parameterized. In order to eliminate this key difference, we replace every action and predicate by all of its suitable instantiations. Which ones are indeed suitable is computed by subsequent reachability analysis.

2.5.2 Reachability Analysis

Following normalization, the set of reachable ground atoms and applicable ground actions is computed by generating a relaxed instance Π_+ of the task. All negative atoms in action preconditions and the goal are omitted, effectively assuming these negative literals to be true. Furthermore, delete effects are also excluded. The absence of negative atoms and effects simplifies the analysis significantly, allowing for a clearer understanding of action reachability. Relaxation produces a task where all originally reachable ground actions and ground atoms remain so, alongside potential new configurations.

The core of reachability analysis involves transforming the relaxed task into a Datalog program. This transformation assigns a new predicate symbol to each action to assert the potential applicability of the corresponding plan action. The program is then augmented with new rules: each rule places the new predicate in the rule head and the action precondition, now a conjunction of positive literals, in the rule body. Subsequently, the effects of actions are encoded with rules where the head is an affected atom and the body includes the action's applicability predicate.

Consider, for example, the following normalized and relaxed action:

$$\texttt{foo} = \langle \underbrace{[\texttt{x,y,z}]}_{\text{params}}, \underbrace{\texttt{P(x,y)} \land \texttt{Q(x)}}_{\text{pre}}, \underbrace{\{\texttt{R(x)},\texttt{Q(z)}\}}_{\text{adds}}, \underbrace{\{\}}_{\text{dels}} \rangle$$

The produced rules are:

$$\begin{aligned} \texttt{foo\_applicable(x,y,z)} &\leftarrow \texttt{P(x,y),Q(x)} \\ \texttt{R(x)} &\leftarrow \texttt{foo\_applicable(x,y,z)} \\ \texttt{Q(z)} &\leftarrow \texttt{foo\_applicable(x,y,z)} \end{aligned}$$

If some action parameter is not represented in the precondition, as is the case here with z, the resulting applicability rule is unsafe. Type normalization, however, ensures that this is not the case, as every variable is represented in a type predicate. This is the case even after precondition normalization (without proof). If the input task is already typeless, however, type normalization can be skipped and rule safety is no longer guaranteed. In this case, one can introduce a new unary dummy predicate alongside corresponding facts in $\mathcal{F}$ for every single constant. Since this predicate will be true for any constant, this does not change the semantics of the affected rules.

Depending on the complexity of the preconditions, the applicability rules may become quite large and are thus decomposed to aid the solving process of the Datalog task. This involves join decompositions, which conjoin multiple atoms in the body, thus reducing the number of predicates each rule must handle. Projection decompositions are also employed to decrease the number of variables within the rules. This is however not implemented in our software.

A key issue here is the manner in which the rules are joined. The complexity of the resulting program may vary greatly based on the order in which decomposition operations are carried out. Helmert's approach prefers to first project away as many variables as possible, and then chooses to join atoms with the lowest number of variables first[1]. This is main aspect that Corrêa et al. address.

Upon solving the Datalog program, the output consists of a list of all reachable ground actions and atoms. These results are then utilized to construct the final ground task. How this is done while taking into account negative and equality literals is described in detail in section 4.4.

Before this last step of grounding, Helmert employs a process called invariant synthesis. This step identifies invariant properties of the state in order to combine binary variables into finite-domain variables. However, for the purposes of generating a

[1]As Corrêa et al. point out, Helmert's paper proposes to join atoms with the most variables, but this has since changed in the implementation.

STRIPS output, which inherently deals only with binary variables, this step is omitted. Invariant synthesis is, however, orthogonal to grounding and could thus be applied subsequently, if required.

The grounding process, while complex, is integral in reducing the planning task to a form compatible with efficient automated solution techniques. By streamlining the task through normalization and limiting the size of the output via reachability analysis, the process ensures an effective preparation of the task for subsequent planning operations.

2.6 Grounding Process by Corrêa et al.

Corrêa et al. thoroughly analyzed the structures of planning tasks that *Fast Downward* fails to ground and then developed two enhancements to reachability analysis. The first is an enhancement to Datalog rule decomposition by employing Tree Decompositions. The second (called *iterated*) effectively splits reachability analysis into two phases, initially computing all achievable atoms and then all applicable ground actions with further use of Datalog programs.

These modifications demonstrate good space efficiency and are able to ground more PDDL tasks with the same amount of storage space. However, there is an inherent time overhead associated with this approach, especially *iterated*. The method is capable of grounding a broader range of problems compared to other techniques but does so at a slower pace.

In terms of PDDL format adaptations, this grounding approach omits some minor features present in Helmert's definition but incorporates significant elements such as the utilization of $\neq$ as a built-in predicate. This is similar to the approach taken in [AL20], where $=$ is treated as a built-in predicate.

Rule Decomposition with Tree Decomposition

One of the key features of Correa's approach is the application of tree decomposition techniques to the rule decomposition process. They note that the maximum treewidth of all produced rules should be minimized, in order to facilitate efficient solving.

Tree decomposition involves converting the primal graph of a rule into a tree structure, where each node of the tree corresponds to a subset of the graph's vertices. This method can greatly reduce the complexity of Datalog rules, particularly those with numerous interdependent variables.

The process begins by representing the dependencies within a rule as a primal graph. Each vertex in this graph represents a variable, and edges indicate which variables jointly occur in the same predicate.

Tree decomposition is then applied to this graph. The resulting tree structure is then used to generate rules with minimal treewidths.

Tree decompositions can be computed by a variety of algorithms. In their implementation, Corrêa et al. use lpopt [BMW20] for Datalog rule decomposition, which by default decomposes the primal graphs with the heuristic Minimum-Degree Elimination (MIW), implemented in the *htd* library [AR16].

The decomposition is achieved using sophisticated algorithms, often implemented in software solutions like lpopt, which are designed to find optimal or near-optimal tree decompositions. By decomposing the rules in this manner, the resulting Datalog problem becomes easier to solve.

Two-Step Reachability Analysis

The second modification to reachability analysis in the grounding process employs a strategic two-step approach that decouples the computation of all achievable atoms with the iteration of all applicable ground actions. Corrêa et al. call this method *iterated*.

First Phase: Generation of Reachable Atoms The initial phase of the analysis focuses exclusively on identifying the set of reachable atoms, without immediately grounding the actions. The motivation behind this is that the applicability rules have a large number of variables, since they of course contain the entire parameter list of their respective actions. For this reason, they are initially omitted from the generated Datalog program, and the effect rules directly have the precondition as their body.

As such, the action from example in section 2.5.2 would yield the following *simplified rules*:

$$\begin{aligned} \mathtt{R(x)} &\leftarrow \mathtt{P(x,y),Q(x)} \\ \mathtt{S(z)} &\leftarrow \mathtt{P(x,y),Q(x)} \end{aligned}$$

This simplified Datalog program is decomposed using the method described in section 2.6 and solved to efficiently yield the model $\mathcal{M}$ and thus the set of achievable atoms. The set of ground actions, however, needs to be determined separately.

Second Phase: Action Parametrization and Grounding

Given the model $\mathcal{M}$, a naive way to calculate the set of ground actions is to construct a Datalog program with $\mathcal{M}$ as the set of initial facts and all applicability rules. When solving this program, however, all discovered plan actions are kept in memory at once. Corrêa et al. thus construct a logic program designed for the iteration of each reachable ground action, without the need to keep previous results in memory. For each action defined in the PDDL task, a separate program is designed in a way such that each stable model describes exactly one valid parametrization. These solutions are then systematically enumerated with an ASP solver .

The separation of atom reachability determination from action grounding limits the space complexity of the grounder and allows for the grounding of more tasks.

Corrêa et al. further refine this method by specifically re-adding the inequality predicates to the second phase. As inequalities are negated atoms, they had been removed by the delete-relaxation. But since $=$ is a static predicate, it can safely be re-added to the action preconditions in the logic program, even after the computation of relaxed-reachable atoms. The refined method is called *iterated*$^{\neq}$. Our implementation by contrast doesn't need to remove negated equality atoms in the first place, because they are separate from predicates in *AI Planning Languages Semantics*.

2.7 Isabelle

Isabelle/HOL (subsequently referred to simply as Isabelle) is a powerful interactive proof assistant based on higher-order logic (HOL). It is used to formalize mathematical proofs through a process that ensures thoroughness and correctness. Isabelle is highly valued in both academic and certain industrial applications for its robust framework, which supports the formal verification of mathematical theorems, algorithms, and even programming languages. Many proofs are publicly available at the *Archive of Formal Proofs* (`https://www.isa-afp.org/`).

Users can define complex data structures and mathematical and logical expressions. Functions are defined using a syntax and functional programming paradigm akin to Haskell, and logical properties are written in intuitive mathematical syntax. This facilitates the expression of program properties, including the critical aspect of correctness as logical statements, which in turn can be proven within Isabelle. Individual proof steps are verified by various builtin algorithms and, if verification is successful, one can trust the rigidity of the proof. Of course, the formulation of correctness statements is still prone to human error, so they must be carefully inspected to ensure that they indeed reflect the program's intended functionality.

Notably, these functions can be exported as executable code, for instance in the language Standard ML (SML). This allows for a practical application of formally verified software in real-world environments

In this project, all functions are directly implemented and their correctness proven in Isabelle.

One important aspect of the syntax are locales. A locale defines variables and assumptions that can be referenced by functions and theorems inside it, and relationships between locales can be modeled as sublocales. This greatly helps reduce overhead in groups of theorems that share the same variables and assumptions, such as that a PDDL task is well-formed.

3 Related Work

In the realm of formal verification and formalization of planning languages, the work of Abdulaziz and Lammich (2020) provides formalizations of PDDL and SAS$^+$ within the Isabelle proof assistant. This contribution is particularly relevant as it directly supplies the input format for the current project and offers fully verified execution semantics and plan validation among other crucial properties. Their project is a baseline for understanding and manipulating PDDL tasks in the context of formal verification.

Equally relevant, Abdulaziz and Kurz (2020) have developed a verified SAT-based planning system that includes a formalization of propositional STRIPS within Isabelle. This work not only supports the output format of many contemporary grounders but also provides a formally verified translation from STRIPS into SAT, into which the output of this project can be piped directly. The solution of this satisfiability problem can be converted back into a valid plan for the STRIPS task.

More progress has been made in the field of plan validation, e.g. for temporal planning by Abdulaziz and Koller (2022) [AK22].

For PDDL, to our knowledge, there exists neither a verified solver nor a grounder. This project aims to close this gap.

Schlichtkrull, Hansen and Nielson (2024) formally verified the semantics of Datalog in Isabelle, including support for negated atoms. Unfortunately, to our knowledge, there exist neither a verified Datalog solver nor a model validator in Isabelle.

4 Implementation

This chapter details the concrete formalization of PDDL and the implementation of the grounding pipeline in Isabelle, including a summary of the relevant correctness theorems. In some highlighted aspects, this formalism differs from the formal definition described in section 2.1.

The input task Π is first normalized, producing the intermediate results Π_2 (after type normalization), Π_3 (after goal normalization) and finally the normalized instance Π_N (after precondition normalization). Then, the delete-relaxation Π_+ is generated and its reachable atoms and plan actions are computed. From these two sets and Π_N, the ground PDDL task Π_G is computed and converted to the propositional STRIPS instance Π_S. This same indexing notation also applies to the tasks' components. Finally, Π_S is encoded into a SAT formula Φ.

Every step in the pipeline supplies a function with which, given a solution for the output, a corresponding plan for the input can be reconstructed. This way, a solution to Φ directly leads to a solution of the original input Π.

Unfortunately, the verification of the software is incomplete. The bigger gap concerns reachability analysis. At the time of writing this thesis, there exists no verified Datalog solver in Isabelle, nor is there a model checker that could potentially verify the output of an external solver. Because of this, a pseudo-Datalog solver was developed to perform reachability analysis of Π_+. Due to time constraints, it could not be formally verified.

Secondly, the preservation of semantics between Π_G and Π_S was not fully verified.

The file `Grounding_Pipeline.thy` summarizes the main theorems, and this chapter's lemmas can be inspected there.

The file `Running_Example.thy` demonstrates the pipeline on the example from section 2.1.4.

4.1 PDDL Semantics

The starting point for this project is the PDDL formalization in *AI Planning Languages Semantics* [AL20]. It supplies the representation of PDDL tasks and defines well-formedness (WF). It also includes verified plan execution and a plan validator.

Many definitions and correctness lemmas are organized into the following locales, which are used extensively throughout our own grounding implementation:

```
locale ast_domain = fixes D :: ast_domain

locale ast_problem = ast_domain "domain P" for P :: ast_problem

locale wf_ast_domain = ast_domain + assumes wf_domain: wf_domain

locale wf_ast_problem = ast_problem P for P + assumes wf_problem:
    wf_problem
```

There are three main differences to the formalism laid out in section 2.1. The first is that there is no function that assigns types to action parameters and constants. Rather, these objects are tuples of an identifier and a type, and thus carry their type information themselves. Likewise, a predicate consists of its identifier and its signature, as a list of types.

This leads to the second difference: everything that needs to be referenced, i.e. constants, predicates, primitive types, actions and action parameters have an identifier. These IDs are either strings or wrappers for strings, but for the purpose of this thesis, we will often ignore the wrapping and just consider every ID to be a string.

It follows that for a PDDL task to be well-formed, these identifiers have to be unique. That is, all actions must have different IDs (but an action may have the same ID as e.g. a constant or a predicate).

This matters because during normalization and grounding, care must be taken to A: not produce auxiliary objects that share a name with existing ones and B: still be able to reconstruct the corresponding plan for the original input problem from a computed plan for the grounded instance.

The third difference concerns the well-formedness of actions. In the implementation by Abdulaziz and Lammich, action parameters are allowed to have non-existent types, i.e. ones that aren't defined in $\mathcal{T}$. If an action parameter x has an invalid type T, the action can still be well-formed as long as x does not appear anywhere in the precondition or the effect. Interestingly, this has no actual bearing on the semantics: the action simply cannot be validly instantiated, as no constant is an instance of T.

Unfortunately, this complicates type normalization. There is no valid type predicate for the non-existent type T, and thus the usual way of letting the precondition handle the type logic falls flat. One approach would be to, in case of such an invalid parameter, modify the precondition such that it can never be satisfied. The semantics of the PDDL task would be unchanged since the action cannot be instantiated, anyway. However, I decided on a more pragmatic solution: restrict the input such that all action parameters have valid types.

Secondly, as explained in section 2.1, we restrict the types of constants and objects to be primitive, i.e. to only have one type alternative.

These restrictions (in addition to the existing restriction that the input problem be well-formed) form the assumptions of the input problem and are formalized as follows:

```
abbreviation "\<omega> \<equiv> Either [''object'']"

definition (in ast_domain) wf_action_params :: "ast_action_schema \<
    Rightarrow> bool" where
  "wf_action_params a \<equiv> \<forall>(n, t) \<in> set (parameters a).
    wf_type t"

fun single_type :: "type \<Rightarrow> bool" where
  "single_type (Either ts) ⟷ length ts = 1"
abbreviation single_types :: "('a \<times> type) list \<Rightarrow> bool"
    where
  "single_types os \<equiv> \<forall>(_, T) \<in> set os. single_type T"

definition (in ast_domain) restrict_dom :: bool where
  "restrict_dom \<equiv> single_types (consts D)
                ∧ list_all wf_action_params (actions D)"

locale restrict_domain = wf_ast_domain +
  assumes restrict_dom: restrict_dom

definition (in ast_problem) "restrict_prob \<equiv> restrict_dom ∧
    single_types (objects P)"

locale restrict_problem = wf_ast_problem +
  assumes restrict_prob: restrict_prob

sublocale restrict_problem \<subseteq> restrict_domain "D"
  [...]
```

4.1.1 Additional Properties

The PDDL formalism of *AI Planning Languages Semantics* includes the correctness proof of a plan validator and of many supporting lemmas. However, there are some properties unrelated to validation that are relevant to this project and thus had to be proven.

Lemma 1 `wf_ast_problem.wf_plan_action_simple`**.**
A plan action is WF iff it references an existing action and its arguments match the action's signature.

Proof. The original formalism includes the additional condition that the resulting ground action after instantiation be well-formed. In a WF context, this is redundant since any WF action with a matching argument list is yields a WF ground action per `wf_ast_problem.wf_resolve_instantiate..` □

Although rather minor, this lemma is referenced frequently, as is the following property:

Lemma 2 `entail_adds_irrelevant/entail_dels_irrelevant.`
If the state t has no atoms in common with the formula φ, then adding or removing t from any state s does not change whether φ is satisfied, i.e.

$$s \Vdash\!\models \varphi \iff s \cup t \Vdash\!\models \varphi \iff s - t \Vdash\!\models \varphi.$$

Proof. This is proven by structural induction on φ. □

This relationship is particularly crucial for type normalization, as the state is extended with type predicates that must not interfere with any action preconditions.

4.2 Normalization

Normalization is a three-fold process involving adjustments to types, the goal, and action preconditions. This pre-processing step closely follows the procedure described in section 2.5.1.

4.2.1 Type Normalization

Since all predicate IDs must be unique, the IDs of the constructed type predicates must be distinct from the pre-existing ones. A type predicate has the ID of the corresponding primitive type but padded on the left with underscores equal to the length of the previously longest existing ID plus one.

For instance, in the example task, the longest predicates are `road` and `rail` with four letters each. Thus, the corresponding predicate for e.g. `car` is the prefix becomes "`_____car`". This admittedly ugly design was chosen because it simplifies verification.

The generation of type preconditions is rather straight-forward, although it should be noted that the functions used to create a conjunctive (or disjunctive) clause from a set of facts $a_1, \ldots, a_n$ produce $a_1 \wedge \cdots \wedge a_n \wedge \neg\bot$ (or $a_1 \vee \cdots \vee a_n \vee \bot$) due to the recursion's base case.

Finally, the supertype facts τ are constructed. For every primitive type, the corresponding set of super types is computed by starting with an empty type graph and iteratively adding edges while simultaneously updating, for each affected type, the set of super types and the set of derived types.

The pseudo-code is given in algorithm 1. The functions f (forward) and b (backwards) respectively assign every node x to the set of reachable nodes from x and the set of nodes from which x is reachable. These functions are initialized to return the singleton set x for every node x; the correct value for a graph without edges. Then, each edge (l, r) is iteratively removed from the input and working set E, while accordingly updating f and b. When adding a new edge (l, r) to a directed graph, for every node x from which l was reachable, i.e. $b(x)$, the set of reachable nodes from r, i.e. $f(r)$ becomes reachable. Thus, f must be updated to

$$f\,(x := f(x) \cup f(r) \text{ for } x \in b(l))\,.$$

The function b is updated analogously. Note that when updating f, the set $f(r)$ is unchanged, so there is no need to cache that before updating f.

Algorithm 1 `reachable_nodes`

```
Input: $E$, the set of edges of a directed graph.
Output: $f$, a function assigning each node to its set of reachable nodes.
  $f \leftarrow \lambda x.\{x\}$
  $b \leftarrow \lambda x.\{x\}$
  while $E \neq \emptyset$ do
      remove $(l, r)$ from $E$
      $f \leftarrow f\,(x := f(x) \cup f(r) \text{ for } x \in b(l))$
      $b \leftarrow b\,(x := b(x) \cup b(l) \text{ for } x \in f(r))$
  end while
  return $f$
```

The set τ is the union of the supertype facts for all constants in $\mathcal{C} \cup \mathcal{O}$ according to their respective types, and it is added to $\mathcal{I}$.

Plan Reconstruction Type normalization preserves the semantics of plans. Given a valid plan for the type-normalized problem, the very same plan solves the original task.

Correctness

First, we must prove the correctness of `reachable_nodes`.

Theorem 3 `reachable_iff_in_star`**.**
The function `reachable_nodes` *from algorithm 1 is given the edges E of a directed graph*

and computes a map that assigns every vertex to its set of reachable vertices (including itself). Equivalently, if E is regarded as a relation, the resulting map describes the reflexive transitive closure:

$$\texttt{reachable\_nodes}(E) \equiv E^*$$

Proof. The function was implemented recursively with an auxiliary argument S that aids in the proof:

$$\begin{aligned}
&\mathit{reachable\_nodes}(E) = \mathit{reach\_aux}(E, \emptyset, \lambda x.\{x\}, \lambda x.\{x\})\\
&\mathit{reach\_aux}(\emptyset, S, f, b) = f\\
&\mathit{reach\_aux}((l,r) \cup E, S, f, b) = \mathit{reach\_aux}(E, (l,r) \cup S, f', b'), \text{ where}\\
&\qquad f' = f\,(x := f(x) \cup f(r) \text{ for } x \in b(l)) \text{ and}\\
&\qquad b' = b\,(x := b(x) \cup b(l) \text{ for } x \in f(r))
\end{aligned}$$

The set S represents the set of processed edges at each iteration. This allows for the definition of the following property: If $f \equiv S^*$ and $b \equiv (S^*)^{-1}$, then

$$\texttt{reach\_aux}(E, S, f, b) \equiv (E \cup S)^*.$$

This is proven by induction over E and the correctness of `reachable_nodes` follows. □

The verification of type normalization is the most involved part of this project because every single aspect of Π, aside from the goal, is modified and the logic for ensuring correct typing is moved to a new system. As with every step in the pipeline, four key properties of the output must be proven:

1. It has the desired format (in this case being type-less).
2. It is well-formed.
3. It has a solution iff the input has one.
4. Given a solution to the output, the plan reconstruction function produces a valid plan that solves the input.

Theorem 4 `ast_problem2.prob_detyped`**.**
Π_2 is type-less.

Proof. This proof is trivial and follows directly from the definitions of the functions that comprise type normalization. □

Theorem 5 `ast_problem2.detype_prob_wf`.
Type normalization preserves well-formedness.

Proof. The main aspect of this is proving that atoms are still correctly instantiated after setting all types to ω, i.e. [`object`].

During type normalization, every function that assigns a type to a constant or variable is modified in a way such that

$$x \in \operatorname{dom}(type) \Longrightarrow type_2(x) = \omega.$$

This is combined with the fact that the arity of predicates is preserved during type normalization (`ast_domain2.t_tyt_params`):

$$sig(p) = L \Longrightarrow sig_2(p) = [(\omega)_{\times|L|}],$$

to prove `wf_ast_domain2.t_atom_wf`. It asserts that, any predicate atom that is WF w.r.t. Π and the type function *type* is WF w.r.t. Π_2 and $type_2$. From this finally follows the generalization to arbitrary formulas (`wf_ast_domain2.t_fmla_wf`).

Subsequently, these properties are combined to show the well-formedness of the resulting actions in $\mathcal{A}_2$ (`restrict_domain2.t_ac_wf`) and finally, of Π_2. □

Next, we show that τ correctly reflects the type system. The function `of_type`(a, b) determines whether a variable of type b can be instantiated with a variable or constant of type a and can be written as:

$$\texttt{of\_type}([a_1 \mid \ldots \mid a_n], [b_1 \mid \ldots \mid b_m]) \longleftrightarrow \forall a_i . \exists b_i . (a_i, b_i) \in \mathcal{T}^*$$

Lemma 6 `ast_domain.of_type_iff_reach`.

$$\texttt{of\_type}([a_1 \mid \ldots \mid a_n], [b_1 \mid \ldots \mid b_m]) \Longleftrightarrow \forall a_i . \exists b_i \in \texttt{reachable\_nodes}(\mathcal{T})(a_i)$$

Proof. This follows from the correctness of `reachable_nodes` and the above definition of the function `of_type`. □

Lemma 7 `restrict_problem.obj_of_type_iff_typeatom`.
A variable with type $[t_1, \ldots, t_n]$ can be instantiated with the constant c iff there exists some i with $\texttt{is\_t}_\texttt{i}(c) \in \tau$.

Proof. If follows directly from the previous lemma and the construction of τ. □

Theorem 8 `restrict_problem2.detyped_planaction_enabled_iff`.
The plan action π is enabled w.r.t. Π in state s iff it is enabled w.r.t. Π_2 in $s \cup \tau$.

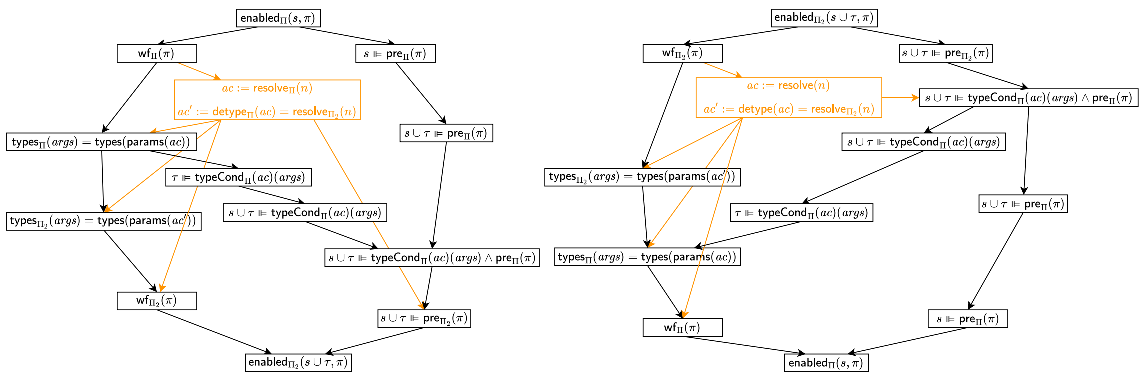
enabled$_\Pi(s, \pi)$
wf$_\Pi(\pi)$
$s \Vdash$ pre$_\Pi(\pi)$
$ac :=$ resolve$_\Pi(n)$
$ac' :=$ detype$_\Pi(ac) =$ resolve$_{\Pi_2}(n)$
$s \cup \tau \Vdash$ pre$_\Pi(\pi)$
types$_\Pi(args) =$ types(params(ac))
$\tau \Vdash$ typeCond$_\Pi(ac)(args)$
$s \cup \tau \Vdash$ typeCond$_\Pi(ac)(args)$
types$_{\Pi_2}(args) =$ types(params(ac'))
$s \cup \tau \Vdash$ typeCond$_\Pi(ac)(args) \wedge$ pre$_\Pi(\pi)$
wf$_{\Pi_2}(\pi)$
$s \cup \tau \Vdash$ pre$_{\Pi_2}(\pi)$
enabled$_{\Pi_2}(s \cup \tau, \pi)$
enabled$_{\Pi_2}(s \cup \tau, \pi)$
wf$_{\Pi_2}(\pi)$
$s \cup \tau \Vdash$ pre$_{\Pi_2}(\pi)$
$ac :=$ resolve(n)
$ac' :=$ detype$(ac) =$ resolve$_{\Pi_2}(n)$
$s \cup \tau \Vdash$ typeCond$_\Pi(ac)(args) \wedge$ pre$_\Pi(\pi)$
$s \cup \tau \Vdash$ typeCond$_\Pi(ac)(args)$
types$_{\Pi_2}(args) =$ types(params(ac'))
$\tau \Vdash$ typeCond$_\Pi(ac)(args)$
$s \cup \tau \Vdash$ pre$_\Pi(\pi)$
types$_\Pi(args) =$ types(params(ac))
wf$_\Pi(\pi)$
$s \Vdash$ pre$_\Pi(\pi)$
enabled$_\Pi(s, \pi)$

Figure 4.1: A graphical representation of the proof that type normalization preserves the enabledness of plan action $\pi = \langle n, args \rangle$ in state s.

Proof. This proof is split into the two directions, and a detailed graphical representation is given in figure 4.1. □

Lemma 9 `restrict_problem2.match_state_step/match_state_step'`.

$$exec(s, \pi) = t \iff exec_2(s \cup \tau, \pi) = t \cup \tau$$

Proof. The main part of this lemma involves proving that there is no overlap between τ and the effect of π, whether it is resolved in Π or Π_2. This proof is straight-forward since the IDs of the type predicates are generated with long prefixes to ensure that they are distinct from the existing predicates. □

From this also follows that the supertype facts are static and their emulated type logic remains intact throughout execution. This is finally used to prove that type normalization fully preserves the semantics of Π:

Theorem 10 `restrict_problem2.detyped_valid_iff`.
The plan πs solves Π iff it solves Π_2.

Proof. This is proven by induction over the plan sequence but in reverse order, i.e. starting from the last plan action executed. The base case follows from lemma `restrict_problem2.goal_sem`:

$$s \Vert\!\models \mathcal{G} \Leftrightarrow s \cup \tau \Vert\!\models \mathcal{G}_2,$$

which is easily proven since $\mathcal{G} = \mathcal{G}_2$ and type predicates can't be represented in the goal. Consequently, the addition of τ to s has no effect on whether or not the goal is satisfied.

The induction is completed using the previous lemmas. □

4.2.2 Goal Normalization

Goal normalization is rather straightforward. In a similar manner to the previous step, the ID of the goal predicate is "`goal`", padded to the left with underscores to be one symbol longer than the previously longest predicate ID. At this point after type normalization of the running example, the longest ID would be "`_____road-rail`", requiring a padding size of 11.

Likewise, the introduced goal action's ID is "`goal`", padded to be one symbol longer than the previously longest action ID. Although its precondition is simply $\mathcal{G}$, its type needs to be altered, as $\mathcal{G}$ is a formula of constants but action preconditions are formulas of constants *and* variables. Additionally, actions may only reference domain-level constants from $\mathcal{C}$ but not from the problem-level $\mathcal{O}$, whereas $\mathcal{G}$ may reference both. Thus, $\mathcal{C}$ and $\mathcal{O}$ are combined into $\mathcal{C}$ (leaving $\mathcal{O}$ empty) to ensure that the new action is well-formed.

Since the goal action depends on $\mathcal{G}$, it is not possible to apply goal normalization to a PDDL domain without the context of a corresponding problem.

Plan Reconstruction Any valid plan for the output needs to execute π_G, and all other actions are unchanged. Thus, the original plan is recovered by including all plan actions up to, but excluding, π_G.

Correctness

Verifying the implementation of goal normalization is much simpler compared to type normalization, although we must additionally prove that goal normalization preserves type normalization.

Lemma 11 `ast_problem.detype_prob_sel(4).`
$\mathcal{G}_3$ is a single literal.

Proof. This is trivial and follows directly from the construction. □

Technically, the requirement is that the goal formula is a pure conjunction, but this is of course satisfied by a formula of a single literal.

Lemma 12 `ast_problem.goal_norm_preserves_typeless.`
Goal normalization preserves type normalization.

Proof. This trivially follows from the fact that the types aren't modified and the added goal predicate and action are 0-ary and thus inherently type-less. □

Lemma 13 `wf_ast_problem3.degoal_prob_wf.`
Goal normalization preserves well-formedness.

Proof. Proving that the auxiliary goal action is WF is straight-forward. It also needs to be proven that all previous actions are still well-formed even after combining $\mathcal{C}$ and $\mathcal{O}$ into $\mathcal{C}_3$. This is achieved with the help of `ast_problem.wf_atom_mono`: adding new constants preserves the well-formedness of existing predicate atoms (and consequently, any existing formulas and actions). □

Lemma 14 `wf_ast_problem3.g_goal_sem_left/g_goal_sem_right.`

$$s \models \mathcal{G} \Leftrightarrow \pi_G \textit{ is enabled in } s \textit{ w.r.t. } \Pi_3$$

Proof. This follows from the definition of the goal action. □

Theorem 15 `wf_ast_problem3.valid_plan_left.`
If the plan πs *solves* Π_3*, then the subsequence of* πs *until but excluding* π_G *solves* Π_2*.*

Proof. The goal predicate only appears in the goal action and is not included in the initial state. Thus, any valid plan for Π_3 must apply π_G. At this point—immediately after the execution of plan πs so far—the current state must satisfy $\mathcal{G}$ per the previous lemma. From this follows that πs solves Π_2. □

Theorem 16 `wf_ast_problem3.degoaled_valid_iff.`
Π_3 *has a valid solution iff* Π_2 *has one.*

Proof. Both directions follow from the previous two lemmas. □

4.2.3 Precondition Normalization

To recap, all action preconditions are put into DNF and split into multiple actions across the disjunctions. Fortunately, *Propositional Proof Systems* [MN17] supplies fully verified functionality to transform any formula into an equivalent conjunctive normal form (CNF) as a set of disjunctive clauses. This can be leveraged since for any formula F,

$$DNF(F) = NNF(\neg CNF(\neg F)),$$

where NNF is the classic way of computing a formula's negation normal form by continuously applying DeMorgan's laws and eliminating double negations. Here, for action a with precondition $pre(a)$, the CNF of $\neg pre(a)$ is computed. The NNF of the negated CNF is then obtained by simply flipping all literals, and reinterpreting the result as a set of conjunctive clauses.

Every clause then corresponds to a single new action with a purely conjunctive precondition. These actions are numbered $a_0, \ldots, a_n$, and their IDs are a concatenation of their index's string representation and the original action's name. The prefix is padded with zeros such that the prefixes across all actions have the same length.

For example, assume that the action `foo` is split into eleven auxiliary actions and `bar` into two. The longest prefix is "`10`" and corresponds to `foo`. Thus, the prefixes for the split parts of `bar` are padded to a length of two as well, yielding "`00bar`" and "`01bar`".

Plan Reconstruction To obtain the original action from one of its parts, it suffices to remove the prefix, all of which share the same length.

Correctness

The proof that the action prefixes, which are natural numbers converted into strings by the function `show`, are distinct, was kindly aided by Maximilian Schäffeler [Sch]:

Lemma 17 `nat_show_inj`.

$$a \neq b \longrightarrow \mathit{show}(a) \neq \mathit{show}(b)$$

Proof. This is proven by induction over the length of digits of a and b. The required induction rule is modified (as `nat_induct_10`) to clarify why it works.

For any property P,

$$(\forall n < 10.\ P(n)) \land (\forall n.\ \forall m < 10.\ P(n) \longrightarrow P(10n + m)) \Longrightarrow \forall n.\ P(n)$$

This induction rule is itself proven by classic induction over n. □

The following three properties of the output format follow directly from the implementation of precondition normalization:

Theorem 18 `ast_problem_4.prec_normed_dom`.
The resulting task's action preconditions are pure conjunctions.

Lemma 19 `ast_problem.prec_norm_preserves_typeless`.
Precondition normalization preserves type normalization.

Lemma 20 `ast_problem.prec_norm_preserves_goal_conj`.
Precondition normalization preserves goal normalization.

Theorem 21 `wf_ast_problem4.split_prob_wf`.
Precondition normalization preserves well-formedness.

Proof. When an action precondition is transformed into a set of clauses that represent its DNF, each clause consists of literals of the original precondition formula. Since this formula is WF, so are all literals and consequently, any formula constructed from them. □

Let *split* be the function that maps each action to the set of actions it is split into, and *origin* be the one mapping each resulting action 'fragment' back to the corresponding original action.

The following equivalent statements describe how an action is enabled iff one of its fragments is enabled:

Lemma 22 `wf_ast_problem4.split_pa_enabled/restore_pa_enabled`.

$$\begin{aligned} &enab(s, \langle a, args \rangle) \iff \exists a' \in split(a).\ enab_3(s, \langle a', args \rangle) \\ &enab_3(s, \langle a', args \rangle) \implies enab(s, \langle origin(a'), args \rangle) \end{aligned}$$

Proof. They follow rather directly from the definition of action splitting. Here, a plan action is decomposed into $\langle a, args \rangle$, where a is the referenced action and *args* are the arguments. □

Lemma 23 `wf_ast_problem4.prenorm_valid_iff`.
Π_N *has a valid solution iff* Π_3 *has one.*

Proof. The execution semantics of a single plan action are formalized as:

$$exec_3(s, \langle a', args \rangle) = exec(s, \langle origin(a'), args \rangle) \text{ and equivalently,}$$
$$exec(s, \langle a, args \rangle) = t \iff \forall a' \in split(a).\ exec_3(s, \langle origin(a'), args \rangle) = t.$$

Since actions and their corresponding fragments only differ in their preconditions, and the goal is unchanged, the preservation of semantics follows with the previous lemma. □

Lemma 24 `wf_ast_problem4.restore_plan_split_valid`.
Plan reconstruction by removing the prefixes from actions IDs is valid.

Proof. For this, the equivalence of *origin* and directly removing the prefix from an action ID is proven, which is trivial. □

4.3 Reachability Analysis

This section details the procedure to obtain all relaxed-reachable ground atoms and actions of the normalized task.

Instead of directly generating a Datalog programs that produce the relaxed-reachable atoms and actions, the delete-relaxation Π^+ is constructed explicitly. Then, the Datalog programs that characterize the exact reachability of Π^+ are created. This extra step aids the proof.

4.3.1 Delete-Relaxation

The implementation of the delete-relaxation Π_+ follows directly from the procedure described in section 2.5.2. All negative action effects are simply removed. The input to this step is assumed to be well-formed and normalized (although the subsequent lemmas also hold without type normalization).

Theorem 25 `normalized_problem_rx.relax_relaxes/relax_normed`.
Delete relaxation preserves normalization and does not contain negated literals in action preconditions or the goal formula.

Theorem 26 `normalized_problem_rx.relax_wf`.
Π_+ *is well-formed.*

Proof. The well-formedness of Π^+ is proven similarly to that of Π_3. All literals left in the resulting action preconditions (and goal formula) of Π_+ are also present in Π_3, where they are WF. Consequently, any formula constructed from these literals is WF. □

Theorem 27 `normalized_problem_rx.relax_achievables/relax_applicables`.

Proof. There are two important aspects of the semantics of relaxed-reachability. Firstly, every plan-action that is reachable in Π is also reachable in Π^+. This follows from property 1: Every WF plan in Π is also WF in Π^+. The overapproximation of reachable atoms follows from property 2: For all plans πs: $\mathit{execs}(\mathcal{I}, \pi s) \subseteq \mathit{execs}^+(\mathcal{I}, \pi s)$. (this is in fact true for any state in place of $\mathcal{I}$.) Both of these properties are jointly proven by induction over the length of πs. □

4.3.2 Reachability Computation

The initial approach was to stay true to the procedure of *Fast Downward* and convert Π_+ into a Datalog problem. Its rules would then be decomposed. Ideally, this would include the refinements by Corrêa et al., which would likely necessitate an external solver for tree decompositions coupled with a verified checker for them.

This Datalog program would then be solved by an external solver, and the output tested by a verified model checker back in Isabelle.

The key problem was that although a Datalog formalization in Isabelle exists [SRN24], it does not include a model checker. Neither does it verify the fixed-point semantics of Datalog, a property crucial for verifying that a stable model indeed represents the set of reachable atoms and plan actions.

The next idea was to implement a model checker for Datalog in Isabelle, although that proved much more complex than initially estimated. The problem is that, for a model $\mathcal{M}$, it has to be checked if every rule is satisfied **for every possible variable valuation σ**.

The naive method would be to, for each rule, enumerate all possible parameterizations and check if each one is satisfied. However, this causes an exponential blow-up in combinations. This method's computational complexity would be as bad as that of a naive grounder to begin with.

An intuitive improvement involves enumerating all facts in $\mathcal{M}$. For each fact f, and rule r, only those valuations are considered, according to which f is in the body of r. This algorithm is already so close to a bottom-up solver that it was instead repurposed to enumerate all reachable atoms and plan actions in Π_+ directly, without first converting to Datalog. The implementation has not been verified, and we consider it to be a proof of concept.

Semi-Naive Evaluation

First, the preconditions of each action a are divided into two sets of literals: $\mathit{preds}(a)$, the positive atoms, and $\mathit{conds}(a)$, the equality and inequality literals. Since the input is relaxed, there are no negated atoms.

The algorithm `semi_naive_eval` iteratively discovers reachable plan actions and atoms. As a pre-processing step, the initial set $\mathcal{I}$ is combined into A_0 with the effects of any actions with empty *preds*, so long as the corresponding *conds* are satisfied. Similarly to empty Datalog rules, they represent facts that must be included in the solution. The corresponding plan actions are noted in P_0, and the set $\mathcal{A}'$ consists of all actions with non-empty *preds*.

The working set W is initialized with the initial facts. For every atom w from W, we

Algorithm 2 `semi_naive_eval`

Input: Π_+, a well-formed, normalized and relaxed PDDL task.
Output: A and P, the respective sets of reachable atoms and ground actions.

```
$A, W \leftarrow A_0$
$P \leftarrow P_0$
while $W \neq \emptyset$ do
    remove $w$ from $W$
    for $a \in \mathcal{A}'$ do
        for 1: $i < |conds(a)|$ do
            $\varphi \leftarrow conds(a)[i]$
            if $pred(\varphi) = pred(w)$ then
                $T \leftarrow \emptyset$
                if $compatible(T, args(\varphi), args(w))$ then
                    $T \leftarrow combine(T, args(\varphi), args(w))$
                    for $j \neq i; j < |conds(a)|$ do
                        $\psi \leftarrow conds(a)[j]$
                        for $f \in A | pred(f) = pred(\psi)$ do
                            if $compatible(T, args(\psi), args(f))$ then
                                $T \leftarrow combine(T, args(\psi), args(f))$
                                if $j$ is the last index (or $i$ is last and $j = i - 1$) then
                                    $N \leftarrow$ all argument lists that satisfy $T$ and $cond(a)$
                                    $P \leftarrow P \cup N$
                                    $A \leftarrow A \cup$ all effects of ground actions in $N$
                                    $W \leftarrow W \cup$ all effects of ground actions in $N$
                                end if
                            else
                                continue
                            end if
                        end for
                    end for
                end if
            end if
        end for
    end for
end while return $A, P$
```

compute all ground actions that include w in the predicate precondition. T is a sort of partial argument list, that assigns constants to only some variables that have already been fixed.

4.4 Grounding

The inputs to the grounding step are the normalized task Π_N, a an over-estimation A of the reachable atoms, and another over-estimation of the reachable plan actions P.

In ground PDDL, binary variables are represented by nullary predicates. Every ground atom from A is mapped to a nullary predicate with the string representation of its index as ID, including a representation of the atom in the form of a suffix, for readability. For each plan action in P, a grounded operator is produced by first instantiating the corresponding ground action, then replacing all ground predicate atoms by their corresponding nullary predicates, which are obtained via a lookup function. The ID of the operator is the index of the plan action, also including a corresponding suffix.

Unfortunately, this process is quadratic in the size of A, because the lookup was implemented in a naive, linear fashion. Curiously, testing reveals that this process is also non-linear in the size of P, likely due to the construction of the action ID mapping described below.

In any case, since STRIPS variable IDs in *Verified SAT-Based AI Planning* aren't required to be strings, it would be better to simply use the representation of ground atoms directly, which would eliminate the need for a lookup function altogether.

Plan Reconstruction This grounding step also produces a mapping from the ground action IDs to each corresponding plan action, which is used to restore the original plan for Π_N.

4.4.1 Correctness

The correctness theorems for this step are stated without proof, although they can of course be inspected in the implementation.

The locale `wf_grounder` places a number of restrictions on A and P:

1. A and P are implemented as lists and have to be distinct.
2. A and P contains at least all reachable atoms and plan actions, respectively.
3. All items in A and P are well-formed. If A and P represent the exact sets of reachable items, this condition follows.

4. All effects from ground actions in P are covered by A, i.e. they only reference atoms in A. This also follows if condition 2 is exact.

5. All preconditions are covered by A. This follows if, additionally, the preconditions of the input are normalized.

6. The goal formula $\mathcal{G}$ is covered by A.

Theorem 28 `wf_grounder.grounded_prob`.
Π_G *is grounded.*

Theorem 29 `wf_grounder.ground_prob_wf`.
Π_G *is well-formed.*

Theorem 30 `wf_grounder.ground_enabled_iff`.
π *is enabled in* s *w.r.t.* Π_N *iff the grounded analog to* π *is enabled w.r.t* Π_G *in the grounded analog of* s*.*

Theorem 31 `wf_grounder.valid_plan_left`.
If πs *is a solution to* Π_G*, then plan reconstruction yields a valid solution to* Π_N*.*

Theorem 32 `wf_grounder.valid_plan_iff`.
Π_G *has a valid plan iff* Π_N *has a valid plan.*

4.5 STRIPS Task Generation

The format of STRIPS from *Verified SAT-Based AI Planning* is very similar to that of normalized PDDL aside from two major differences:

Firstly, addition and deletion effects must not overlap. When translating a ground PDDL action a into a STRIPS operator, this requirement is ensured by removing any atoms that occur in $add(a)$ from $del(a)$, as $add(a)$ has priority.

Secondly, operator preconditions are limited to positive atoms only. This means that they cannot contain equality literals, the symbol $\perp$, or negated atoms.

Equality literals are removed by simply evaluating them. The literals $\perp$ and $\wedge\perp$ are simulated by the respective auxiliary variables '-' and '+', of which only '+' is added to the initial state. They are don't appear in any operator effects and are thus static.

Each predicate p gets two corresponding variables $+p$ and $-p$. Actions are translated into operators such that the resulting add effect contains the corresponding positive variables of the grounded add effect and the negative variables of the delete effect. The opposite applies to the resulting delete effect.

Plan Reconstruction We can't simply use STRIPS operator IDs to reconstruct the original plan action, because they don't have any. Instead, since the actions of Π_G are directly mapped to STRIPS operators, the operator with index i in the list of operators corresponds to the action with index i in $\mathcal{A}_G$.

4.5.1 Correctness

Theorem 33 `grounded_normalized_problem.wf_as_strips`.
If Π_G is grounded, normalized and well-formed, then Π_S is a valid STRIPS task.

Unfortunately, due to time limitations, we are not able to verify the preservation of semantics, nor the correctness of the plan reconstruction.

4.6 Output

Figure 4.2 displays the STRIPS instance of the transportation task, as produced by the grounder. In this case, the ground task produced actually contains only the exact set of executable operators and variables that can ever be assigned to true.

The following is a valid plan for the STRIPS instance:

1. `load1-p1-v-B`
2. `choochoo1-v-B-C`
3. `drive1-v-C-D`
4. `load1-p2-v-D`
5. `drive2-v-D-E`
6. `unload2-p2-v-E`
7. `drive2-v-E-C`
8. `choochoo2-v-C-B`
9. `unload2-p2-v-B`
10. `goal`

Finally, it is piped backwards through the pipeline's plan reconstruction functions to produce a valid plan for the original problem Π:

1. `LOAD p1 v B`
2. `CHOOCHOO v B C`
3. `DRIVE v C D`
4. `LOAD p2 v D`
5. `DRIVE v D E`
6. `UNLOAD p2 v E`
7. `DRIVE v E C`
8. `CHOOCHOO v C B`
9. `UNLOAD p2 v B`

```
Variables:
    is_object-A, is_object-B, ..., is_object-p2,
    is_city-A, is_city-B, ..., is_city-E,
    is_movable-c1, is_movable-c2, ..., is_movable-p2,
    ...,
    road-A-B, road-C-E, road-E-D, road-C-D, rails-B-C,
    at-p1-A, at-p1-B, ..., at-p1-E, ... at-p2-E,
    at-c1-A, at-c1-B,
    at-c2-C, at-c2-D, at-c2-E,
    at-t-B, at-t-C,
    at-v-A, at-v-B, ..., at-v-E,
    goal
Init:
    is_object-A, is_object-B, ..., is_object-p2,
    is_city-A, is_city-B, ..., is_city-E,
    is_movable-c1, is_movable-c2, ..., is_movable-p2,
    ...,
    road-A-B, road-C-E, road-E-D, road-C-D, rails-B-C,
    at-c1-A, at-c-B, at-p1-B, at-t-C, at-p2-D, at-c2-E
Goal:
    goal
Operator drive1-c1-A-B:
    PRE: is_car-c1, is_city-A, is_city-B, road-A-B, at-c1-A
    ADD: at-c1-B        DEL: at-c1-A
Operator drive2-c1-A-B:
    PRE: is_car-c1, is_city-A, is_city-B, road-B-A, at-c1-A
    ADD: at-c1-B        DEL: at-c1-A
...
Operator load1-p1-c1-A:
    PRE: is_parcel-p1, is_car-c1, is_city-A, at-p1-A, at-c1-A
    ADD: in-p1-c1       DEL: at-p1-A
Operator load2-p1-t-A:
    PRE: is_parcel-p1, is_train-t, is_city-A, at-p1-A, at-t-A
    ADD: in-p1-t        DEL: at-p1-A
...
Operator goal:
    PRE: at-p1-E, at-p2-B       ADD: goal       DEL:
```

Figure 4.2: The result of grounding the example task defined in section 2.1.4. Variable and operator IDs are modified to exclude the numerical prefixes, and most items are omitted. Some action names include an index due to being split during precondition normalization. Consequently, there are two variants of the action `DRIVE`: one where there is a road from the source to the destination, and another where the road is defined in the opposite way.

5 Conclusion

In this thesis, the PDDL grounder by Helmert (2009) was implemented in Isabelle. This includes the computation of all relaxed-reachable atoms and plan actions, but excludes invariant synthesis. The created program grounds PDDL tasks (in the format from *AI Planning Languages Semantics*) into propositional STRIPS (in the format from *Verified SAT-Based AI Planning*) and converts the result into a SAT formula with functionality by Abdulaziz et al. (Oct. 2020). Any solution to this formula can be converted back into a plan for the original PDDL task.

The normalization, relaxation and grounding steps are formally verified, but the reachability analysis and the conversion from grounded PDDL into STRIPS are not. The pipeline is highly modular and all steps can be executed independently of each other.

A major limitation is that the algorithm used to generate the ground PDDL instance from the set of reachable objects is highly inefficient. In addition, we don't use a formal Datalog format to perform reachability analysis. A lot more work is needed to be able to use a computationally efficient external Datalog solver, and confirm its output with a formally verified model checker.

In conclusion, this thesis reinforces the viability of formally verified planning systems and supports future developments in the formal analysis of planning tasks.

5.1 Future Work

Building on the work presented in this thesis, several directions for future research and development present themselves.

The most important next step is to complete the formal verification, especially of the conversion to the STRIPS format. Moreover, the computational efficiency, especially of the first grounding step, can be significantly improved with minimal effort.

The algorithm for reachability analysis could be verified, but due to the shared similarities between the relaxed PDDL task and Datalog, it is likely more promising to implement the same algorithm directly for Datalog, which has a wider range of applications. This would naturally suggest the implementation and verification of a conversion between relaxed PDDL and Datalog, to complete the grounding pipeline.

The implementation of invariant synthesis—the step that transforms the grounded problem into the finite-domain representation SAS$^+$—would be equally valuable. *Verified SAT-Based AI Planning* already includes a formalization of SAS$^+$ semantics, including a model checker and a reduction to STRIPS.

Finally, if a variant of typed Datalog [ZPS09] is used during reachability analysis, the step of type normalization could be simplified or even omitted.

Bibliography

[AK20] M. Abdulaziz and F. Kurz. "Verified SAT-Based AI Planning." In: *Archive of Formal Proofs* (Oct. 2020). `https://isa-afp.org/entries/Verified_SAT_Based_AI_Planning.html`, Formal proof development. ISSN: 2150-914x.

[AK22] M. Abdulaziz and L. Koller. "Formal Semantics and Formally Verified Validation for Temporal Planning." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 36. 9. 2022, pp. 9635–9643.

[AL20] M. Abdulaziz and P. Lammich. "AI Planning Languages Semantics." In: *Archive of Formal Proofs* (Oct. 2020). `https://isa-afp.org/entries/AI_Planning_Languages_Semantics.html`, Formal proof development. ISSN: 2150-914x.

[AR16] M. Abseher and M. Röbke. *htd*. `https://github.com/mabseher/htd`. Version 1.0.0 (beta1). 2016.

[ARA24] M. Abdulaziz, A. Rimpapa, and T. Ammer. *Isabelle-Graph-Library*. `https://github.com/mabdula/Isabelle-Graph-Library`. 2024.

[Bäc92] C. Bäckström. "Equivalence and Tractability Results for SAS$^+$ Planning." In: *Principles of Knowledge Representation and Reasoning*. Proceedings of the Third International Conference. 1992, pp. 126–137.

[Bic15] M. Bichler. "Optimierung Nichtgrundierter Answer Set Programme durch Regelzerlegung." BA Thesis. TU Wien, Nov. 2015.

[BMW20] M. Bichler, M. Morak, and S. Woltran. "lpopt: A rule optimization tool for answer set programming." In: *Fundamenta Informaticae* 177.3-4 (2020), pp. 275–296.

[Bra19] Bram28. *Exponential blow-up of CNF to DNF 2^n terms?* Mathematics Stack Exchange. 2019. URL: `https://math.stackexchange.com/q/3113090` (visited on 03/19/2025).

[CAA22] P. Carbonn, H. C. Alves, and T. Andras. *pyDatalog*. `https://github.com/pcarbonn/pyDatalog`. Version commit 45b0f3b. 2022.

[Cal+12] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca, and T. Schaub. "ASP-Core-2: Input language format." In: *ASP Standardization Working Group* (2012).

[CGT+89] S. Ceri, G. Gottlob, L. Tanca, et al. "What you always wanted to know about Datalog(and never dared to ask)." In: *IEEE transactions on knowledge and data engineering* 1.1 (1989), pp. 146–166.

[Cor+23] A. B. Corrêa, M. Hecher, M. Helmert, D. M. Longo, F. Pommerening, and S. Woltran. "Grounding planning tasks using tree decompositions and iterated solving." In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 33. 2023, pp. 100–108.

[Der+08] A. Dermaku, T. Ganzow, G. Gottlob, B. McMahan, N. Musliu, and M. Samer. "Heuristic methods for hypertree decomposition." In: *MICAI 2008: Advances in Artificial Intelligence: 7th Mexican International Conference on Artificial Intelligence, Atizapán de Zaragoza, Mexico, October 27-31, 2008 Proceedings 7*. Springer. 2008, pp. 1–11.

[Epp07] D. Eppstein. *A graph and its tree decomposition*. 2007. URL: `https://commons.wikimedia.org/wiki/File:Tree_decomposition.svg` (visited on 03/19/2025).

[Hel09] M. Helmert. "Concise finite-domain representations for PDDL planning tasks." In: *Artificial Intelligence* 173.5-6 (2009), pp. 503–535.

[HMS15] T. Hammerl, N. Musliu, and W. Schafhauser. "Metaheuristic algorithms and tree decomposition." In: *Springer Handbook of Computational Intelligence* (2015), pp. 1255–1270.

[How+98] A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. "Pddl - the planning domain definition language." In: *Technical Report, Tech. Rep.* (1998).

[HSS08] A. A. Hagberg, D. A. Schult, and P. J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX." In: *Proceedings of the 7th Python in Science Conference*. Ed. by G. Varoquaux, T. Vaught, and J. Millman. Pasadena, CA USA, 2008, pp. 11–15.

[Lif87] V. Lifschitz. "On the semantics of STRIPS." In: *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*. 1987, pp. 1–9.

[MN17] J. Michaelis and T. Nipkow. "Propositional Proof Systems." In: *Archive of Formal Proofs* (June 2017). `https://isa-afp.org/entries/Propositional_Proof_Systems.html`, Formal proof development. ISSN: 2150-914x.

[Mor+09] C. Morrison, D. Bryce, I. Fasel, and A. Rebguns. "Augmenting instructable computing with planning technology." In: *ICAPS'09 Workshop on the International Competition for Knowledge Engineering in Planning and Scheduling*. 2009.

[MW12] M. Morak and S. Woltran. "Preprocessing of Complex Non-Ground Rules in Answer Set Programming." In: (2012).

[Ofi15] Ofir. *A clique in a tree decomposition is contained in a bag*. Mathematics Stack Exchange. 2015. URL: `https://math.stackexchange.com/q/1228108` (visited on 03/19/2025).

[PWiki] *Planning.Wiki - The AI Planning & PDDL Wiki.* URL: `https://planning.wiki/` (visited on 03/19/2025).

[Sch] M. Schäffeler. *String inequality with show*. Comment on the Isabelle Zulip forum.

[SRN24] A. Schlichtkrull, R. Rydhof Hansen, and F. Nielson. "Isabelle-verified correctness of Datalog programs for program analysis." In: *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*. 2024, pp. 1731–1734.

[THN05] S. Thiébaux, J. Hoffmann, and B. Nebel. "In defense of PDDL axioms." In: *Artificial Intelligence* 168.1-2 (2005), pp. 38–69.

[WangD] D. Wang. *Temporal Planning Certification*. unpublished.

[ZPS09] D. Zook, E. Pasalic, and B. Sarna-Starosta. "Typed Datalog." In: *Practical Aspects of Declarative Languages*. Ed. by A. Gill and T. Swift. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 168–182. ISBN: 978-3-540-92995-6.