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:	March 17th, 2025

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

Munich, March 17th, 2025 Maximilian Vollath

Acknowledgments

Abstract

Grounding classical planning tasks is a fundamental yet complex problem in automated planning. The Fast Downward planning system is renowned for its effective approach to grounding first-order Planning Domain Definition Language (PDDL) tasks into the finite-domain representation SASP, employing reachability analysis to reduce the number of ground actions necessary for problem solving. Recent advancements by Correa et al. have further refined reachability analysis.

This thesis presents an implementation of the grounding procedure within the Isabelle proof assistant, where I have formally verified its correctness. The implementation closely follows the methodologies applied in Fast Downward, but I notably exclude the step of invariant synthesis from my process. This results in an output in the propositional planning language STRIPS rather than the finite-domain language SASP. Through this work, I demonstrate that the grounding procedures not only meet theoretical expectations but are also practically viable within a formal verification environment, thus laying groundwork for future explorations into verified planning systems.

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), Simplified Action Structures (SAS), and STRIPS are particularly noteworthy. PDDL is a higher-level, first-order language that includes concepts such as parameterized actions and predicates. Due to its complexity, PDDL is not well-suited suited for direct input into solvers. In contrast, STRIPS is a propositional planning language, meaning actions are not parameterized and the state consists of binary variables. Because of this simplicity, it is more feasible to design solving algorithms for tasks defined in STRIPS. In contrast to STRIPS, SAS$^+$is a finite-domain language, meaning variables can have more than two discrete values.

The process of transforming a task from a first-order representation into a propositional representation is grounding. In order to obtain a solution for a PDDL task, it is usually first grounded and then fed into a solver. A naive grounder, which simply instantiates every action and predicate with every possible combination of parameters, can be easily implemented. However, this approach leads to an exponential increase in ground variables and actions, rendering it infeasible for even modestly sized problems.

A good grounder must thus restrict the number of ground variables and actions produced. This is commonly done by employing reachability analysis, in order to omit impossible predicate and action instantiations from the ground problem. Note that it is typically infeasible to compute the exact set of reachable states or satisfiable ground actions. Instead, a superset is commonly produced.

A common grounding procedure was introduced by Helmert in 2008 and implemented as part of the Fast Downward planning system. For a relaxed version of the input PDDL task, the exact set of satisfiable actions instances is computed, using logic programming. In addition, Helmert employs an algorithm called invariant synthesis to produce finite-domain variables instead of binary ones, reducing the number of variables in the ground task. It is important to note that invariant synthesis is orthogonal to the grounding process. Correa et al developed two refinements to the reachability analysis which reduce the complexity of the generated logic program. Firstly, tree decomposition is employed to generate a more efficient form. Secondly, the set of satisfiable action instances is computed by first calculating the set of reachable states.

While implementations of these methods are publicly available, they are not formally verified. Formally verifying them is the aim of this project. The automated theorem prover Isabelle facilitates this process. In Isabelle, formulas and even entire programs can be expressed in a formal language. Within this framework, properties such as appropriately defined correctness specifications can then be proven. However, for this, the entire program has to be re-implemented in Isabelle, in a functional programming language similar to Haskell. Note that the correctness of an algorithm does not necessarily have anything to do with whether or not it is efficiently implemented.

While to our knowledge, no fully verified grounder exists, there are formally verified plan checkers for our languages of interest, and even a verified solver for STRIPS. One might then wonder why it is necessary to formally verify a grounder or a solver. Indeed, given a PDDL planning task, one could input it into an unverified grounder and an unverified solver, and simply verify the obtained solution with a verified plan checker. The key difference then, however, is that if the untrusted solver does not find any solution, one cannot be certain that such a solution indeed does not exist. Only if the entire procedure is verified, we can confidently assert that a solution does not exist. Additionally, other properties of the planning task can potentially be verified, such as the optimality of a given solution.

My contributions include the formal verification and optimization of Helmert's grounding algorithm and Correa's refinements, implemented in Isabelle. This includes formal verifications of graph algorithms that are used as subroutines: an enumeration of the transitive closure of a relation and an algorithm to produce a tree decomposition for a graph.

2 Background

Todo: General stuff about this project, maybe repeat that I do PDDL into STRIPS instead of into SASP.

2.1 Abstract Planning Language Syntax

2.1.1 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 versions and levels of PDDL and we are concerned with the abstract syntax specified in **[AbLa]**, which my implementation builds upon. It is very similar to the syntax used by Helmert but differs in a few ways which I will highlight.

PDDL tasks are commonly divided into domain and problem.

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. **[AbLa]** supports Either types. Hence, a type is a list of primitive type alternatives. 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. Although **[AbLa]** allow constants to have multiple types, I restrict them to singular types instead, in order to be consistent with Helmert and Correa.

$\mathcal{P}$ is the set of predicate symbols. Each predicate symbol additionally has a corresponding signature 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 that only refers to constants. Equality and inequality can be modeled by predicates. However, some formalizations such as **[AbLa]** (and Correa), consider equality (or inequality, respectively) a built-in binary predicate.

A state is a set of ground atoms and thus defines a valuation of these binary variables.

$\mathcal{A}$ 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 $dels(A)$. The parameter list defines variables and assigns a type to each one of them. The precondition is a formula over predicate atoms instantiated with domain constants and/or parameter variables. Helmert allows these formulas to contain first-order quantifiers, but **[AbLa]** restricts them to propositional formulas. The atoms in the effect are likewise instantiated with constants and/or variables. Helmert supports nested effect preconditions In other formalizations, the action parameters are often implicitly defined via the open variables in the precondition and the effect. However, I decided against that due to ambiguities that may arise from the use of Either types and for the sake of simplicity.

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 $\mathcal{C}$.

The initial state $\mathcal{I}$ is simply 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 (Todo: explain), even though Correa et al later omit them. Axioms are sets of rules that define new predicate atoms based on which atoms are true in a given state.

An important concept is well-formedness of a PDDL task. well-formedness, type graph unrestricted, object is predefined type

Semantics

Actions can refer to constants from $\mathcal{C}$ but not those from $\mathcal{O}$. Other than that, there is no semantic bearing on whether a constant is defined at the domain-level or the problem-level.

Example

running example: delivery problem

2.1.2 STRIPS

- grounded PDDL

- propositional logic only

Abstraxt Syntax

- Ab+Ku specification + implementation: no overlap between add/del, goal allowed to contain negative literals
- Ab+Ku specifics: goal allowed to contain negative literals
- vs SAS^+

Semantics

- parallel plans: not necessary
- Ab+Ku: if operator isn't enabled, it does NOOP.

Example

2.1.3 Datalog

- $\mathcal{F}$ facts, $\mathcal{R}$ disjunctive rules
- unique stable model "canonical model"
- constructed iteratively from facts

2.2 Grounding Process by Helmert

- Fast-Downward planning system
- Commonly used
- Invariant Synthesis (skipped because not necessary for STRIPS); orthogonal to grounding, so can just be applied to STRIPS afterwards.

2.2.1 Normalization

- remove types
- simplify formulas in conditions and goal
- normally invariant synthesis right afterwards, but can also be done after grounding.

2.2.2 Reachability Analysis

- Relaxed Task: negative literals assumed to be true, thus superset of reachable states.
- Datalog generation
- Rule decomposition
- (iterated) solving
- ground instance. Output normally: finite-domain. Here: STRIPS

2.3 Grounding Process by Correa

- (?) good space efficiency, but time overhead. Grounds more problems but slower
- PDDL format differences: omit minor features present in Helmert's definition, but the main thing is $\neq$ as built-in predicate, like how Ab+La has $=$.

2.3.1 Rule Decomposition with Tree Decomposition

- primal graph for rule
- tree decomposition
- tree width
- tree decompose this graph with lpopt

2.3.2 Two-Step Reachability Analysis

- avoid grounding actions at first to keep tree width low
- encode type predicates (or alternative unary predicates)
- iterated neq
- inequalities only considered in second phase to keep tree-width low (kind of what I'm doing, since Eq is a builtin predicate in Ab+La)

2.4 Isabelle

- Interactive proof assistant
- implementation with functional programming
- write proofs that are automatically validated
- code can be automatically translated into standard ML

3 Related Work

- Helmert + Correa quick mention
- Ab+La and Ab+Ku provide verified input/output formats and checkers. Ab+La includes SAS+ potentially for outlook. Ab+Ku includes solver for STRIPS Ab+Ku additionally provides solver
- lpopt thesis?: tree decomp proofs
- Otherwise not much formally verified work on grounding. But many grounders.

4 Implementation

- Parser for problem (not verified) into AST
- Ab+La as input format

4.1 Datatype Representation

Ab+La specifics:

- term: Union of variable and constant.
- instance with only constant terms is essentially already grounded.
- Formula atoms: either facts or Eq
- e.g. effects are "atom formula list" in favor of "PredAtom list". Restricted to list of facts by wf, not by types
- Either-type consts
- wrapper classes to ensure safety (can't confuse variable and const)
- objectT
- Closed-world and Entailment/Valuation!

4.2 Input Restriction

- single-typed constants + explanation. What if we don't restrict this?
- goal only conj (MVP)
- Well-formed action parameters (potentially an oversight)

4.3 Normalization

4.3.1 Type Normalization

- Create instance where only type is “object”
- unique predicate names
- generate supertype facts
- reachable nodes algorithm
- generate param preconds. If action params were allowed to not be wf, then action would not have valid instantiations, thus precond could just be False instead.
- remove types
- easier for me because no forall in effects
- Main theorem: equivalence of semantics

4.3.2 Formula Normalization

- Goal already conjunction. (axioms originally required to compile disjunctions into) Alternatively put goal into CNF, create dummy actions with clauses as preconditions, define dummy goal.
- Preconditions: turn formula into CNF, then remove negative atoms (for relaxed reachability), then split across disjunctions.
- Operator ID organization
- NNF sufficient relaxed reachability -> STRIPS formulas. But CNF needed for datalog
- Main theorem: equivalence of semantics w.r.t. corresponding actions.

4.4 Reachability Analysis

4.4.1 Relaxed Task Generation

- Just remove negative literals
- Main theorem: reachable states are superset of OGs

4.4.2 Datalog Generation

- rules are safe due to type-predicates. (every parameter is mentioned in preconditions).
- If not, dummy(x) is added (MVP)
- Main theorem: reachable states equivalent to relaxed instance

4.4.3 Datalog Rule Decomposition

- Join/projection rules
- normal form
- Performance-critical issue: How to choose joins.
- Helmert's Algorithm, Correa's Heuristic

correa: Helmert prefers atoms with many variables to join first, correa with few variables first.

- external datalog solver

4.4.4 Refinement: Tree Decomposition

- tree decomposition definition
- External algorithm (?)
- Algorithm external! No proof that algorithm ever terminates. BUT in principle, there is always a tree decomposition.

4.5 Grounded Instance Generation

- Interpret datalog solution: No solution -> goal not reachable
- Construct instance from set of reachable operators
- TO STRIPS: no overlap between add and del effects!
- STRIPS: Goal allowed to contain negative literals

- STRIPS: if operator isn't enabled, it does NOOP.
- piping into Ab+Ku solver

5 Proof Outline

5.1 Normalization

5.1.1 Type Normalization

5.1.2 Formula Normalization

5.2 Reachability Analysis

5.2.1 Relaxed Task Generation

5.2.2 Datalog Generation

5.2.3 Datalog Rule Decomposition

5.2.4 Refinement: Tree Decomposition

5.3 Grounded Instance Generation

6 Results

6.1 Efficient Implementation

- proving equivalence between mathematical specification and efficient implementation
- no proof of efficiency bounds

6.2 Code Generation

- short explanation of linking external software

6.3 Performance

- re-measure correa implementation
- obviously worse but maybe better than expected?
- Isabelle generates code into ML, which is less efficient (space + time) than compiled code (correa).
- Still, hopefully same order of magnitude

7 Conclusion

- I re-implemented it
- I proved it
- it runs, not fast but it runs

Outlook:

- Allow disjunctions in goal formula!
- support PDDL axioms
- implement invariant synthesis to change output to SAS^+
- consider types in datalog translation (Typed Datalog exists)

Glossary