Autonomous Online Hierarchical Conformance Refinement Planning using Answer Set Programming for General-Purpose Robots

By

Oliver Michael Kamperis

A thesis submitted to
the University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

Intelligent Robotics Group
School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham
December 2023

"Let us dedicate ourselves to what
the Greeks wrote so many years ago:
to tame the savageness of man and
make gentle the life of this world."

- Robert F. Kennedy (Indianapolis USA 1968)

ABSTRACT

This thesis proposes a novel approach to rapid online task and high-level action planning, called Hierarchical Conformance Refinement (HCR), which focuses on robotics applications. HCR planning is domain-independent, geared towards problems with many complex interacting constraints, and designed to minimise downtime and maximise productivity of robots. HCR is built in Answer Set Programming (ASP), a declarative knowledge representation and reasoning paradigm. ASP is highly effective for solving complex planning problems involving large amounts of descriptive domain knowledge. However, existing ASP planners perform poorly for problems with long minimal plan lengths. HCR tackles this weakness, by combining ASP with a novel mechanism for hierarchical refinement planning, and finds in the union of their complementary capabilities, the overcoming of their weaknesses and reinforcement of their strengths. The resulting technique enables a flexible divide-and-conquer method that exponentially reduces problem complexity and naturally supports online planning. This greatly improves speed and scalability to large problems with long plan lengths, whilst maintaining the high expressivity and generality of existing ASP planners.

Simulated experiments ran on a combined blocks world and navigation domain show that HCR planning significantly outperforms the classical approach to ASP based planning by exponentially reducing execution latency and total planning times in the minimum plan length of a problem. HCR planning reduced median execution latency by between 81 to 99% and total planning time by between 42 to 98%, over classical ASP based planning, for only between 0 to 11% reduction in plan quality, from the easiest to hardest problems tested. For the most complex problem tested, a robot equipped with the HCR algorithm can reduce total planning time from 607.0 to 14.1 seconds, and execution latency to less than 7 seconds. This makes ASP planning fast enough to be used for practical robotics applications.

DEDICATION

"To my parents George and Michele, who have always loved and supported me unconditionally; I am so very lucky and thankful to have you and for all the things you have done for me, I love you endlessly. To my siblings Sam and Sophie, thank you for being there when I need you, I know I can always rely on you, and know that you can rely on me too; we are a family forever. To my grandparents, uncles, aunts, and cousins, I have always enjoyed your company, and I am thankful for your love and acceptance; for those that have now passed, you live on in me and the whole family. To my friends Jake and Roxanne, who I have greatly enjoyed having dinners and watching films with during hard times; thank you for dealing with my unique personality, and caring about me when I needed you. To my friend Liam, who I have very fond memories of staying up late to watch silly videos or discuss the meaning of life and origin of the universe with; I often wish we could go back to those simpler times. To my friends Richard and Dom, who I have shared so many silly jokes and puns with; thank you for making me laugh and cheering my up when I am down. To all my other friends old and new that have come and gone, I thank you for all the happy memories we have shared, and I look forward to making many more. Finally, to all of the humanity, I do this work because I truly believe it contributes just a bit towards making our world a better place. We face so many challenges in this age; we are on the precipice of achieving greatness, or losing everything. Always use your time and efforts for the good of everyone, remember at all times how many things we have to be thankful for, and how many things we have in common. Try to understand and to comprehend those who are different to you, have only love and compassion toward one another, and help others with their troubles as you would like to be helped with yours."

- Oliver Michael Kamperis

ACKNOWLEDGMENTS

Thank you to my supervisors Marco Castellani and Yongjing Wang for giving me a huge amount of their time and effort throughout my PhD. Thank you to all the staff at the University of Birmingham for keeping everything running. Thank you to my family and friends for support and advice along the way.

Contents

List of Figures

List of Tables

Chapter One

Introduction

Autonomous robots are becoming highly capable and prevalent in the modern world. Robots are now being deployed into human-populated spaces, including our homes and workplaces, to perform practical tasks, previously done only by humans. However, despite this rapid and promising progress, many fundamental open research problems still exist in the field.

This thesis focuses on the development of domain-independent automated planning systems that can scale to the extreme size and complexity of physical automation problems. The core requirements for such a system, are that it be both fast and general enough to be useful for a wide variety of practical problems, where the available planning time is very small. More specifically, the aim of this thesis is to develop an online hierarchical satisficing planner, which generates discrete deterministic task and high-level action plans. Whereby, an online planner minimises downtime and maximises productivity of autonomous agents by planning partly overlapped with execution, and a satisficing planner reduces computational complexity and planning time by searching only for satisfactory quality plans (rather than optimal).

The central idea of this thesis towards achieving this objective, is that solving planning problems efficiently, requires a model and method for reasoning with them that promotes two of the core concepts involved in human decision making; abstract reasoning and problem decomposition. These concepts allow us to break down and solve arbitrarily large, complex, and unseen problems, into many smaller and more manageable parts. Those parts can then be dealt with distinctly and separately, allowing us to focus our reasoning and effort on only the part(s) of a problem that are currently most important. This is the ability that enables us to make progress towards even very long-term goals in our infinitely complex reality.

1.1 Problem Statement

Planning is a fundamental reasoning capability required by any general-purpose autonomous agent deployed into dynamic physical application domains, such as service robots for the residential, retail, or hospitality sectors. The complexity and variety of tasks these robots will be required to complete, makes it impossible to pre-program them to perform a fixed set of repetitive actions, as is done with classical industrial robots, such as the robotic arms used in manufacturing. Furthermore, when such robots are available in our homes and workplaces, people will expect them to act promptly on instructions given, and to complete all desired tasks as quickly as possible. General-purpose autonomous agents must therefore reason for themselves to rapidly generate high-quality plans that complete and achieve arbitrary tasks and goals, with little latency before execution, using only general techniques that can handle size and diversity of physical domains, without relying on significant existing experience.

In automated planning, an agent is given an abstract domain and problem description, and must use it to find a sequence of actions, which if executed in order, will transition some arbitrary given initial state to some desired goals. This requires the advanced high-level cognitive ability to reason about and simulate the effects of a series of related actions, on a series of states of the world, prior to execution, and into often large future horizons.

There is one major challenge, the complexity of the type of problems that occur in planning, typically explodes exponentially with both the minimum plan length (number of actions needed to reach the "closest" goal state) (Korf, 1987; Bylander, 1994a) and the problem description size (the number of actions and variables needed to describe a problem and plan) (C. A. Knoblock, 1990a). Specifically, in a search graph or tree, the time complexity of naive exhaustive search is worse case $O(b^d)$, where b is branching factor (the mean number of legal actions per state) and d depth of the shallowest goal state (the minimum plan length). It is difficult to conceive this exceptional complexity, however (Russell and Norvig, 2016) (from Page 80) provide a thorough description. As an intuitive example, consider the rela-

tively simple game of Chess[1]. Up to $10^{46.25}$ valid states are estimated to exist (Chinchalkar, 1996), and whilst each player has "just" ≈ 30 average moves per turn, after only 5 turns, over $\approx 6.9 \times 10^{13}$ possible games exist (Shannon, 1950). Therefore, even for small branching factors, the exponential increase in search space with search depth, makes problems unsolvable via any naive exhaustive search method. Clearly, almost all interesting practical problems, which will likely have many more legal actions, cannot be solved in this manner.

The field of automated planning has now seen attention for more than five decades. The majority of research has focused on developing methods to overcome its complexity. Despite this, existing planners are either too slow, or too specialised and not general enough, to be useful for most physical automation problems. Unfortunately, developing a general planner, that is appropriate for a wide range of problems, typically comes at the cost of greatly reduced computational efficiency, over more specialised techniques tailored specifically to only a few problems. Obtaining a planning algorithm that is both fast and general is clearly non-trivial. The relevant existing approaches, from which the concept of the novel approach contributed by this thesis was conceived, are now briefly reviewed to contextualise the core thesis contributions. Chapter 2 presents the full literature review.

1.2 Background and Motivations

Classical state space planners have been studied extensively. Classical planners primarily employ heuristic mechanisms to guide search and minimise problem complexity. For simple problems, such as path planning, where an accurate estimate of the cost or "distance" to the closest goal state can be found trivially, domain-specific heuristics can be optimally efficient and extremely effective (Hart, Nils J Nilsson, and Raphael, 1968). However, for more complex combined problems with many interacting constraints, such as those involving both navigation and manipulation, accurate domain-specific heuristics can be difficult to obtain

[1] Although Chess is a competitive game, its complexity is indicative of most interesting planning problems.

(without forming large machine learnt models), because there is rarely a direct or meaningful way to measure the distance to the goal (Helmert, 2003). Fortunately, domain-independent heuristics, which are typically based on the concepts of abstraction and problem decomposition (aligning with the ideals of this thesis), can produce very effective general heuristics because they can be extracted from any given problem description in a general way (Pearl, 1984). The downside, is that these heuristics often lead to limited expressiveness of a problem representation, since they are restricted to what can be understood by the heuristic.

Answer Set Programming (ASP) based planning has in contrast seen significant praise for its highly expressive and elaboration tolerant modelling language (Gelfond and Kahl, 2014a). ASP is a declarative non-monotonic logic programming paradigm that is now well established as an effective alternative to classical heuristic planning (Erdem and Patoglu, 2018). It is highly applicable for discrete deterministic planning, since it: a) allows domain knowledge to be written and extended easily through human-intuitive axiomatic logic rules; b) supports complex Knowledge Representation and Reasoning (KRR) capabilities (such as deriving the indirect effects of actions, reasoning about state constraints, or dealing with recursive relations); and c) can handle the large amounts of facts and relations that must be dealt with by a planning agent (Lifschitz, 2008). The satisfiability based general problem solver used by ASP can handle complex combined problems, because the solver can reason efficiently with large knowledge bases containing many highly interacting constraints, thus giving great freedom in the expression of a problem representation (Gebser, Kaminski, Kaufmann, and Schaub, 2019). Unfortunately, in comparison to classical heuristic planners, ASP planners perform poorly for problems with long plan lengths (Jiang et al., 2019), and the general heuristics for classical planning are not appropriate for ASP planners.

On a separate track, Hierarchical Refinement (HR) planning has sought to employ the human use of abstraction for problem solving, to greatly increase the efficiency of planners when scaling to problems with long plan lengths, at the expense of minor losses in plan quality (Sacerdoti, 1974). The use of abstraction has been studied extensively and is widely accepted

as an effective method for simplifying reasoning tasks for humans and AI (Newell, Simon, et al., 1972). Humans employ abstraction extensively when solving problems, e.g. planning and giving explanations (Giunchiglia and Walsh, 1992). Through abstraction one is able to focus on only that which is most important. Non-pertinent details are neglected to reduce complexity or assumptions made to cope with unknowns (Timpf et al., 1992). For planning, the concept is simple; only initially decide on the abstract goals or stages of a plan, allowing the details to be dealt with later and as they appear. This allows us to plan achieve short-term goals, which is exponentially simpler than planning to achieve long-term goals, and look for only reasonable/satisfactory quality plans, which is usually considerably easier than finding the optimal. If a long-term goal is unmanageable, the problem must first be broken down into multiple short-term goals (in some sequence) that eventually realise the original goal. Without this ability, the complexity of automated planning seems overwhelming. The goal of HR planning is to capture this ability in order to avoid using heuristics. This allows HR planning to reduce the complexity of planning without limiting expressivity or generality, if small losses in the quality for generated plans are allowed for (Washington, 1994).

Existing HR planners employ a divide-and-conquer approach to break large planning problems into many smaller sub-problems. The method generates a high-level abstract plan in a relaxed model of the planning problem, where a sub-set of action preconditions are ignored (C. A. Knoblock, J. D. Tenenberg, and Q. Yang, 1991). Each action of an abstract plan is then "refined" separately, using its start state and previously ignored preconditions, as the initial and goal states of a sub-problem at the next (more concrete) level. Solving this sub-problem generates an expanded sub-plan. The central idea is that this method is effective for solving large planning problems with long plan lengths, since they are divided into many linearly shorter and therefore exponentially simpler to find sub-plans (C. A. Knoblock, 1990a). Further, for complex problems, the necessary abstractions may be easier to obtain than the equivalent heuristic. However, this did not always prove true for three reasons: a) existing planners have very limited capacity to express abstractions, b) which leads to a very

rigid mechanism for plan refinement that forces sub-problems to be solved independently, and b) this requires an implicit assumption of independence between the sub-problems used, which often fails (Bacchus and Q. Yang, 1994). Ultimately, HR planners of this form cannot always find an appropriate abstraction for many interesting planning domains and problems, and they are rarely able to consistently produce effective divisions of a problem.

Recent research has however shown that ASP can perform HR-like planning in a flexible way using an alternative constraint addition based refinement method (S. Zhang, F. Yang, et al., 2015). Recent research also suggests that ASP is well suited for representing abstraction hierarchies, and that the variety and complexity of the abstractions that can be expressed naturally in ASP exceeds that of older HR planners (Saribatur, 2020; Sridharan, Gelfond, et al., 2019). Unfortunately, this research was limited in scope, lacking generality in the refinement method and versatility in the problem division mechanisms, with critical details of the theory missing, and very little of the implementations made public. Resultantly, these works on ASP based HR planning have not garnered additional research interest.

1.3 Summary of Hierarchical Conformance Refinement

This thesis contributes Hierarchical Conformance Refinement (HCR), a novel approach to discrete online task and high-level action planning, employing abstract reasoning and problem decomposition, to minimise down-time and maximise productivity of robots. HCR is an ASP based HR paradigm, that combines the expressiveness and elaboration tolerance of ASP with the speed and scalability of HR, for only minor losses in plan quality. The approach is human-intuitive, domain-independent, and geared towards problems with complex interacting constraints and long plan lengths. Little domain- or problem-specific knowledge is required by a HCR planner, other than a model of the domain's dynamics, the capabilities of the robot(s), and a specification of the abstraction hierarchy. A HCR planner, called ASH, The ASP based online HCR planner, has been fully implemented with this thesis.

HCR uses a new method of constraint addition plan refinement based on sub-goal generation and achievement, which for the first time supports online partial-planning. Online planning allows partial-problem solutions to be rapidly yielded to the robot(s), to greatly decrease execution latency and total planning times. This is enabled through a generalised concept of problem division, whilst simultaneously alleviating the problem dependencies that existed in past HR planners, in order to increase the quality of generated plans. Further, HCR supports an arbitrarily large hierarchy that allows the expression of any type of abstraction to which a state abstraction mapping can be defined (explained below). This makes HCR planning appropriate for a wide range of applications where the available planning time is very small, and only the fundamental knowledge of the domain dynamics is known.

1.3.1 Mechanism and Benefits of HCR Planning

HCR planning is conceptually simple. A given planning domain and problem is defined over an abstraction hierarchy, containing a unique model of the domain dynamics at each level, where higher levels are abstractions of the lower. Models at different levels of abstraction are connected through a state abstraction mapping, a function which maps all low-level states to one abstract state at higher-level and all abstract states are mapped to at least once. The intuition is that the constraints of a problem are gradually generalised or removed ascending this hierarchy. The planner then generates and incrementally refines plans, as constraints are gradually specialised or reintroduced descending the hierarchy, therefore breaking the task of accounting for them into many distinct steps. The concept, is that this separation makes it easier to account for the constraints than dealing with them all simultaneously.

The planning problems specified by each level are solved iteratively, in descending order. Only in the highest and most abstract model is a complete classical plan generated. Since each abstraction can reduce the problem description size and minimum plan length linearly, the abstract search space is reduced exponentially, making the complexity of an ab-

stract problem comparatively trivial to the original (C. A. Knoblock, 1990a). The abstract plan is then successively refined at all lower levels. The plan refinement process involves expanding plans with more actions, or specialising existing actions, to satisfy any specialised or reintroduced problem constraints that were previously ignored in the abstraction. Abstract plans increase linearly in length at each level of refinement. The length of the top-level classical plan thus increases exponentially with the depth of the hierarchy.

The abstract plan is refined under the primary component of conformance refinement planning, the conformance constraint. This requires that plans achieve the same effects and remain structurally similar at all levels. The constraint is formed by a sequence of sub-goal stages. Each stage is defined by the joint direct effects of the set of actions planned on a given time step at the previous abstraction level. These must then be achieved in the same order at the next level. The goal used in classical planning is then considered the final-goal. The idea is that, following a path through the sub-goal stages when solving a more concrete problem, should guide the planner to make monotonic progress towards achieving, or enabling actions that achieve, the final-goal. The structure of an abstract plan, once found, thus makes it easier to find conforming plans that solve the more complex/concrete problems.

More specifically, the conformance constraint provides four primary novel benefits;

1. It guides search by restricting the search space, this reduces search time simply by forcing low-level planning to pass through states that satisfy the constraint (by achieving the stages in the correct order), therefore eliminating from search any potential plans that do not satisfy the constraint or achieve conformance with the abstract plan.

2. It allows complete-problems to be divided, and sub-divided at all levels, into sequences of partial-problems, each defined by a contiguous sub-sequence of sub-goal stages from the previous level (a generalisation of sub-problems used in past HR planners);

   (a) Under the assumptions that minimal plan length scales linearly with problem size

(in terms of number of sub-goal stages included), and that problem complexity is exponential in the minimal plan length, then solving a sequence of partial-problems is exponentially simpler than solving the respective complete-problem.

(b) Since partial-problems are solved independently, the planner is able to yield executable partial-plans to the robot incrementally and online (continuously sending more actions to the robot), and exponentially faster than finding a complete plan. Since only the first partial-problem must be solved prior to execution (then progressively extended online, towards eventual achievement of the final-goal), the execution latency and downtime of a robot can be greatly minimised.

3. Its flexibility as an ordering constraint maintains high quality plans. Since all sub-goal stages included in the same partial-problem are considered and pursued simultaneously, their achievement can interleave. This allows the achievement of early sub-goal stages to be delayed if doing so better prepares for what is needed to achieve later sub-goal stages sooner, and reduce overall plan lengths. The dependencies that occurred between the sub-problems in past HR planners are therefore much alleviated.

4. It allows the progression of planning and search towards the final-goal (in terms of the number of sub-goal stages achieved) to be observed externally and online, providing highly desirable transparency and performance evaluation to human operators.

The downside of HCR planning is that optimal solutions are not guaranteed. Abstract plans are generated non-deterministically and in ignorance of the original problem. As such, there is no clear criteria for selecting the best abstract plan to refine. The requirement to achieve conformance with abstract plans of unknown quality, can lead refinement planning away from optimal solutions to more concrete problems. Further, the division of complete-problems into partial-problems, can expose dependencies between sub-goal stages in different partial-problems. These dependencies may reduce plan quality, if the solution to an early partial-problem leaves the planner in a state which makes it harder to solve a later one.

1.3.2 Abstraction Hierarchies in HCR Planning

HCR planning provides a novel ASP encoding that allows representation of an arbitrarily large abstraction hierarchy. The hierarchy supports any abstract model, including any action or state abstraction, to which a deterministic and exhaustive state abstraction mapping can be defined from the original model to the abstract. This is possible, because HCR refines plans based on action effects only, and therefore abstract and refined plans can be connected at different levels through the state abstraction mapping between action effects.

To construct an abstraction hierarchy, the lowest-level model, called the ground model, is designed first, and the abstract models are then designed iteratively "from the ground up", in ascending order. The ground model is a description of the actual problem to solve.

The abstract models are then created by applying an abstraction to another model, called its original, this may be the ground or another abstract model. An abstract model of a planning problem is a contraction on the original, with fewer states and/or more state transitions. Plans yielded from abstract models only need to be achievable via actions from the original, and are not executed in reality. This thesis explores abstraction in two contexts:

- Relax the problem, removing some of its constraints (typically negative effects or preconditions of actions), allowing transitions between more states, with the intention of reducing plan lengths (as less actions are necessary to achieve the final-goal).
- Apply a state abstraction, removing or generalizing actions and state variables (and related constraints), reducing both the problem description size and quantity of valid reachable states, with the intention of reducing search spaces and plan lengths.

The representation used in HCR planning is however more general and not restricted to these two abstractions. A designer can modify, replace, delete, or preserve domain laws however they wish, so long as a state abstraction mapping exists. Further, abstract models can be constructed easily, because actions, state variables, and domains laws, are explicitly

and quantitatively parameterised by abstraction level, whereby all such components that are not modified, replaced, or deleted, are automatically preserved in an abstraction.

HCR intrinsically supports condensed domain models, the novel type of action and state abstraction contributed by this thesis. Condensed models are intended to reduce problem description sizes by forming generalised and more granular action and state variables. They extend past work (Sridharan, Gelfond, et al., 2019), by adding a relation over class types, that specifies a logical coordination between at least two related classes that both subordinate from one super-class. One is defined as the ancestor class and at least one defined as the descendant class(s). Sets of entities of descendent classes, are combined into abstract descriptor entities of the ancestor class, which serve to represent the union of the descendants. For example, buildings and rooms are both ways of describing the location of an object (building and room being related classes, which both subordinate to location). If a ground model represents the location of objects based on the room they are in, a condensed model can be formed by only considering the location of objects relative only to what building they are in, therefore increasing the granularity of the representation. In particular, this class-based specification automatically generates all of the condensed model's abstract domain laws, including those that are modified, avoiding the need to manually re-write the abstract domain laws (which was necessary in past work using similar abstractions).

1.3.3 HCR Online Planning Systems

This thesis proposes two core decision making systems involved in online HCR planning; problem division strategies and online planning methods. A division strategy decides how problems are divided, and an online method decides when they are divided and subsequently solved. Both are separate external decision making systems, that plug into the algorithms used by HCR planning, and are therefore general in the sense they can be customised to use any desired algorithm to make decisions, or (external) knowledge bases to inform them.

During the course of online planning the division strategy and online method together construct and traverse a problem division diagram. This is a branching structure whose nodes contain partial-planning problems and the partial-plans that solve them, and whose arcs link abstract problems and plans, to their refinement problems and refined plans. Special traversal links then denote the order in which partial-problems are solved and partial-plans generated, allowing one to trace and observe the progression of online planning. Ultimately, this controls how quickly and how often ground-level plans are yielded to the robot for execution, and is therefore greatly influential in minimising down-times of the robot.

Note that since refinement problems refine abstract plans and abstract plans are generated during planning, refinement problems which are deeper in the diagram depend on problems which are shallower in the diagram. Similarly, since later partial-problems extend earlier problems, a partial-problem depends on all problems to the left of it in the diagram. Therefore, the diagrams must be constructed from left-to-right and from top-to-bottom.

Figure 1.1 is an example problem division diagram. Where the notation $x : y$ denotes problem number y at level x, and $a \rightarrow b$ denotes abstract plan length a expanding to refined plan length b. The black downwards links from parent nodes to their children, represent the generation of partial planning problems of the abstract plans in parent nodes. When a node is visited, its problem is solved, when it is expanded, the refinement problem of its abstract plan is divided. The red traversal links then denote the order nodes were visited in.

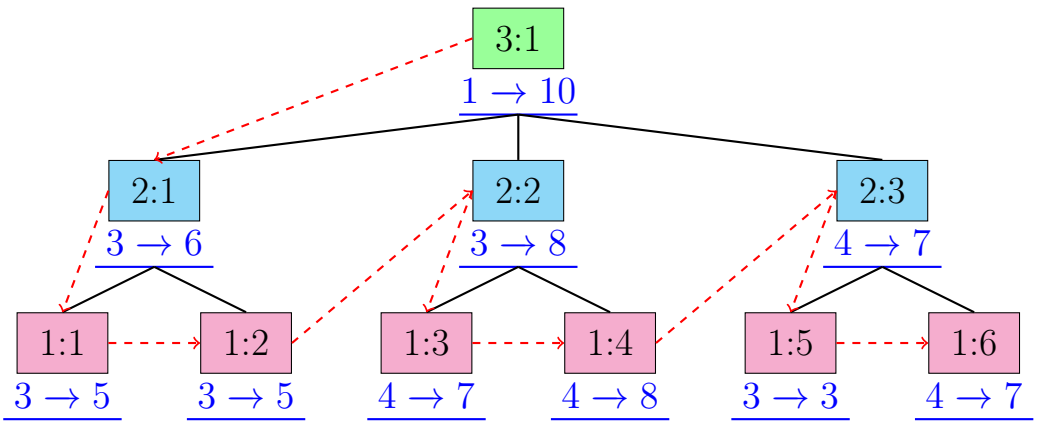


Figure 1.1: Problem Division Diagram Example.

1.3.4 ASH: The ASP based HCR Planner

The Answer Set programming based Hierarchical conformance refinement planner (ASH) is a domain-independent planner that implements the new HCR planning process proposed by this thesis. A complete software implementation is available at Kamperis, 2023.

The diagram in Figure 1.2 is a conceptual drawing of the structure of ASH, where the arrows indicate control and information flow between components. The following provides a high-level description of the concept and intuitions behind the design. From the top;

1. The top purple box are ASH's problem-specific inputs; the initial-state and final-goal, and various configuration settings containing input parameters for ASH's internal planning and search algorithms, and external decision making methods and strategies.

2. The left-hand blue boxes are a hierarchical planning domain definition, containing the robot's knowledge of the fundamental physical laws of the dynamic planning domain in which it operates, and various abstractions of them. This knowledge consists of the sequence of domain models arranged in an arbitrarily large abstraction hierarchy, where adjacent levels are linked via the upwards state abstraction mapping (between state variables). The ground-model, at level 1, is the most detailed model, defining all original domain laws in the most specific forms needed to solve the actual original input planning problem. Hierarchical levels $l > 1$ contain more abstract models obtained by; removing or generalising domain laws/problem constraints, or by applying action or state abstractions, from/to the ground-model. This can exponentially reduce the complexity and description size of the abstract models of a planning problem.

3. The central red box is ASH itself, containing primarily: a) its planning algorithms and sub-systems, for monolevel and hierarchical planning, sequential yield mode, and incremental ASP search; b) the various decision making systems involved in online planning; and c) its six operational modules that encapsulate ASH's domain-independent ASP

logic rules. These rules are used for defining (amoungst other things); the generation of a complete problem definition, and constraints on what constitutes a valid state, state transition, and solution to classical or conformance refinement planning problems.

4. The right-hand pink boxes indicate the flow of HCR planning. A plan is generated at each level in the abstraction hierarchy in descending order. Only at the top-level is a classical plan generated, at all lower levels a conformance refinement plan is generated by refining the abstract plan from the previous level (under a conformance constraint obtained from it), until a ground plan is obtained which solves the input problem.

5. The bottom green box are the planner outputs: a) the resulting hierarchical plan, a hierarchy of monolevel plans over all abstractions linked by a conformance mapping; b) and the experimental statistics and log files used for performance evaluation.

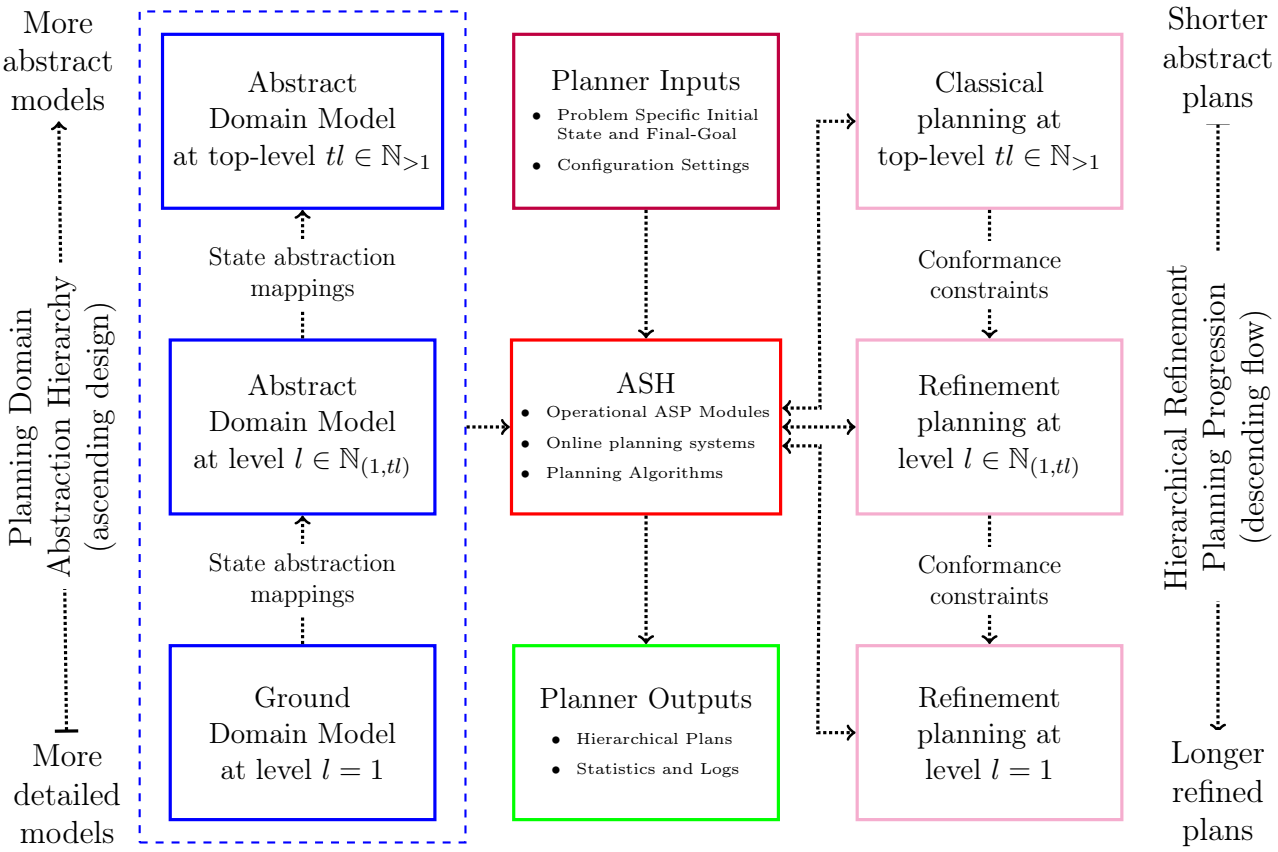


Figure 1.2: Structure of ASH.

1.4 Thesis Aim and Objectives

The aim of this thesis is to develop HCR planning, a new domain-independent online satisficing action and task planning paradigm. HCR combines ASP with HR planning, with the goals to reduce total planning and execution latency times of ASP based planners, to improve their scalability to larger problems with long minimum plan lengths. The motivation for improving ASP planning is that whilst it is powerful for expressing complex planning problems, it is not currently fast enough to be used in most practical applications.

The individual objectives of thesis to achieve its aim are as follows:

- Develop the theory of and fully implement the proposed HCR planning approach:
  - Overcome the primary limitation of ASP planners, by developing a general method to employ HR to improve scalability to problems with long plan lengths.
  - Overcome two main limitations of existing HR planners, by relaxing their rigid plan refinement and problem division methods to; improve generated plan quality, and support a wider range of abstract models to increase generality.
  - Enable support for online planning, whereby planning is partly overlapped with execution, in order to minimise the execution latency of robots.
- Experimentally evaluate the performance of HCR against classical ASP planning:
  - Validate: a) whether the proposed approach scales more effectively to problems with longer plan lengths than classical ASP planners whilst maintaining good plan quality; and b) that online planning reduces execution latency of robots.
  - Identify any limitations of HCR and paths for future work to overcome them.

Note that although HCR planning has been developed in this thesis with a focus on robotics, HCR is a general tool that is applicable for many other automation tasks, such as disassembly sequence planning for re-manufacturing and AI for video games.

1.5 Thesis Novel Contributions

The following lists the specific novel contributions of the thesis:

- The conformance refinement concept to iterative hierarchical plan refinement, whereby plans must achieve the same effects and remain structurally similar at all levels.
- The conformance constraint formed by sequences of sub-goals stages which must be achieved in order but can interleave to alleviate dependencies between them.
- The introduction of online partial-refinement planning using partial-problems formed by any contiguous sub-sequence of sub-goal stages, and associated decision making systems; problem division strategies and online planning methods.
- A generalised hierarchical representation that supports any abstract model to which an exhaustive deterministic state abstraction mapping can be defined.
- The class-based encoding that automatically generates condensed domain models.

1.6 Thesis Outline

The rest of this thesis is structured as follows. Chapter 2 presents the literature review, identifying the primary issues with past planning approaches related to HCR planning. Chapter 3 then: a) begins by highlighting the novelty of HCR planning that overcome those issues; b) then presents the test planning domain used in this thesis; c) before presenting the theory behind formal definitions for HCR hierarchical planning domains, planning problems, and what constitutes their solutions. Chapter 4 provides all details of the implementation of this theory, including the online planning systems, modules of ASH, and planning algorithms. Chapter 5 contains the experimental evaluation of this implementation. Finally, Chapter 6 concludes by stating the successfulness of achieving the thesis aims and objectives.

1.7 Theoretical Assumptions

The following describes the theoretical assumptions imposed by the scope of this thesis.

1. *Completeness of Knowledge Assumption:* The problem specification is assumed complete and consistent. The planner must know all relevant domain laws, the complete initial state, and the static structure of the domain (this includes all entities involved), before planning and with certainty. In many real-world scenarios, this assumption will not hold, since it is unlikely that the robot will know the initial domain state or structure without having to map out the domain, and as most dynamic domains are subject to significant entropy, it is unlikely that existing knowledge will be correct for long.

2. *Determinism of Execution Assumption:* Generated plans are assumed to never fail, because the plan's execution is deterministic, i.e. it will progress exactly according to the expected simulated sequence of states and state transitions. Hence, although the proposed approach and provided planner can yield partial plans online, it does not deal with plan failure, diagnostics, or re-planning. This assumption is inadmissible in most dynamic domains, primarily because the existence of exogenous actors (such as humans) will often change the state during execution unbeknownst to the robot.

3. *Uniform Execution Time and Cost of Actions Assumption:* All planned actions are assumed to take a uniform amount of time to execute and have uniform cost. Therefore, the current planner does not take into account (or optimise over) the execution time or resource cost of a plan, and the amount of actions in the plan is assumed to sufficiently represent its quality. This reduces plan optimisation to an action cardinality minimisation problem. This assumption is often impractical since, for many real-world applications, minimising execution time and resource costs will be of importance.

Future work proposed in Section 6.2 includes some ideas for extending the contributions of this thesis to relax these assumptions and handle the resulting problems.

Chapter Two

Literature Review

The field of automated planning has now seen attention for many decades. Its main research goal is the development of algorithms that are expressive enough to model and fast enough to solve the immensely complex planning problems that exist in practical applications.

This chapter reviews symbolic automated action and task planners which are relevant to the contributions of this thesis. The review focuses primarily on the strategy that planners use to construct a plan and on the representation used for modelling domains and problems. The exact method used to search the space of possible plans to find a solution is not the focus of this thesis. Symbolic action and task planning, particularly hierarchical planning, has seen relatively little recent research attention and therefore some references may seem dated. Most recent work in planning is closer to motion planning and usually uses probabilistic or machine learning based methods which are not strictly relevant to this thesis.

This chapter first provides a general background on classical planning and a review of abstractions used for heuristic classical planning. Second, it reviews hierarchical planning in two main areas; hierarchical task and goal network planning, and hierarchical refinement planning. Third, answer set programming as a tool for general problem solving is discussed. Before finally, answer set programming for classical and hierarchical planning is reviewed.

The review includes primarily theoretical research works. However, recent applied research in the field has seen interest, particularly for combined task and motion planning (Guo et al., 2023) and human-robot collaboration (Wang et al., 2018; Evangelou et al., 2021). These indicate a clear need for general, fast, and high quality planning algorithms.

2.1 Classical Planning

Classical planning domains are defined by a discrete state transition system, a finite graph whose nodes are states and edges are state transitions labelled by actions[1]. Solving a classical problem requires searching this graph for a linear plan (a single finite sequence of actions), which traces a path from the initial state to the goal state. Actions are planned sequentially (one per step), and their effects are deterministic, instantaneous and Markovian.

Classical planning typically focuses on domain-independence, where a planner should be able to interpret a description of any given planning domain (written in some modelling language), which serves as a high-level specification for such a transition system. The possible state transitions are described by "lifted" action schemas, that define only the preconditions and effects of actions[2]. State constraints are not supported as standard in classical planning. Therefore, ensuring that an invalid state is never reached requires carefully declaring action preconditions and effects such that no state transition can end in an invalid state.

The seminal work on classical planning was the Stanford Research Institute Problem Solver (STRIPS) (Fikes and Nils J. Nilsson, 1971), which was famously demonstrated on "Shakey", the world's first intelligent mobile robot (SHAKEY, 1966-1972). STRIPS could model simple robotics problems, such as performing navigation and reasoning about the position of domain entities over time, and, in contrast to earlier planners using atomic representations, it could handle the many facts and relations required for planning. STRIPS has since inspired many highly expressive modern planning paradigms, including the Planning Domain Definition Language (PDDL) (Ghallab, C. Knoblock, et al., 1998), which is now generally considered the standardised formalism for modelling planning problems.

[1]Formally, a state transition system is a tuple $Y = \langle S, A, T \rangle$, where S is the set of possible states, A actions, and T state transitions (labelled by actions from A) (Nielsen, Rozenberg, and Thiagarajan, 1990). If s is a state, a an action, and $t = \langle s, a, s' \rangle$ a state transition, then such a system can be defined by two functions; $actions(s) = \{a_1, a_i, ..., a_n\}$ which is the set of actions whose preconditions are satisfied in s, and $result(s, a) = s'$ which is the state s' resulting from applying the effects of a to s (defining a transition t).

[2]This implicitly defines the $actions(s)$ and $result(s, a)$ functions, and therefore a state transition system. For more details and examples of how this works (for PDDL), see Pages 366-369 of Russell and Norvig, 2016.

In PDDL an action schema represents a generalization used to generate all ground instances of that action by binding the schema's parameters to specific values. State variables are then used to represent attributes of the domain state whose value can change over time.

Equation 2.1 is an example PDDL description declaring a locomotive *Move* action. The effects allow the *robot* to move from location $from$ to location *to*, if the following preconditions are satisfied; the robot is located in $from$, and both locations are directly connected. Notice that both the positive and negative action effects must be specified explicitly.

$$\begin{aligned} &Action(Move(robot, from, to), \\ &\qquad \text{PRECOND: } In(robot, from) \wedge Connected(from, to) \\ &\qquad \text{EFFECT: } \neg In(robot, from) \wedge In(robot, to)) \end{aligned} \tag{2.1}$$

PDDL planners use heuristics to reduce search complexity. A heuristic function takes a state as input and estimates the remaining cost required to reach a goal state. This estimate is used to bias search towards expanding potential plans of lowest cost, as these are likely to reach a goal state the soonest. The best heuristics produce accurate estimates and are fast to evaluate. To ensure optimal plans, heuristics must however be admissible and consistent. An admissible heuristic is one that never overestimates the true cost of reaching a goal state from any given state. A consistent heuristic is one where the estimated cost of reaching a goal state from a state σ_1, is less than or equal to the cost of reaching a connected state σ_2, plus the estimate of the cost of reaching a goal from σ_2, for all states connected to σ_1.

PDDL enables heuristics to be domain-independent because its action schemas allow heuristics to be extracted automatically from the problem description. Heuristics based on abstractions are amoungst the most popular and effective towards this end (Bonet and Geffner, 2001; Hoffmann, Sabharwal, and Domshlak, 2006) and are reviewed below. Unfortunately, these heuristics can restrict the expressiveness of problem descriptions because they are limited to what can be dealt with by the heuristic. Further, some require a pre-processing step that is often computationally expensive and very rarely generalises.

2.2 Abstractions for Classical Heuristic Planning

The intuition of heuristics based on abstraction, is to extract and solve a computationally cheap abstract version of a problem, and then use its solution to provide heuristic estimates for plans that solve the original problem. Given a classical planning problem representation as a graph search, an abstraction can be formed simply by adding edges to and/or removing nodes from it. It is strictly easier to find a goal node from any given start node in such an abstract graph. The resulting plans must therefore underestimate the cost of an plan in the original graph that transition between the same nodes. For the abstraction to be effective, the time taken to find a solution to the abstract problem must be less than the time saved by following its heuristic guidance. However, such abstractions are difficult to find.

Relaxed models have been studied and used extensively in the development of general heuristics (Pearl, 1984). The most common examples are ignoring action preconditions or ignoring delete lists (the negative effects of actions). The prior creates a model in which a sub-set of (or potentially all) preconditions of actions are ignored (Sacerdoti, 1974). This means that more actions are enabled in any given state and therefore less actions are needed to reach a goal state (plan lengths are shorter). The latter creates a model where monotonic progress towards the goal is always possible (Hoffmann, 2005), because actions never "undo" the effects of other actions, and therefore actions can enable other actions but never disable them. This again means that less actions are needed to reach a goal state since: a) when a state that enables an action is reached, the action remains enabled for the rest of the plan, and b) when a goal is achieved, it can never be unachieved. Both types of relaxed model can be easily derived automatically from a PDDL description, and can form accurate heuristic estimates[3]. However, for large problems, relaxed models alone are rarely sufficient, since they do not reduce the number of states. Further, they are calculated by solving, an easier but still expensive, abstract planning problem, for all states that are searched through.

[3]Relaxed models are however prone to local minima, since if critical problem constraints are removed, the cost of the abstract plan can be far lower than the actual cost, reducing the effectiveness of the heuristic.

Reducing the number of states and quantity of state variables needed to represent the state can be achieved with state abstractions (Helmert, Haslum, Hoffmann, et al., 2007). State abstractions can be problem- or domain-specific, and can either remove state variables or group together a set of state variables into one generalised abstract variable. Typically, problem-specific state abstractions are more powerful, because they can eliminate state variables that are not relevant to a particular problem. This can produce a compact representation, but they are hard to obtain, since the concept of relevance is rarely clear without attempting to solve the problem. Domain-specific state abstractions are easier to obtain but less powerful, since they can only use general properties of a domain to form an abstraction.

A different but effective technique instead decomposes a problem into many sub-problems, each containing one sub-goal (Y. Chen, Hsu, and Wah, 2004b). For any given state, the remaining goals yet to be achieved can be split into separate sub-goals, and then pursued/solved independently. Given an assumption of independence between the sub-goals, the computational cost of solving the decomposed problem is the sum of the costs of solving the sub-problems, and the heuristic value approximated by the sum of the costs of the sub-plans. Unfortunately, this assumption often fails because naively splitting the complete goal into separate sub-goals, solving them independently, and then concatenating their solutions, rarely produces a valid plan that reaches the complete goal (J. Tenenberg, 1988; Smith and Peot, 2014). This problem can cause two affects: a) since an action in a later sub-plan can unachieve a goal achieved by an earlier sub-plan, the concatenated plan might not achieve all goals and is not a valid solution (although admissible for heuristic purposes); and b) two sub-plans can contain redundant copies of the same action, making the cost of the concatenated plan overestimate, and this be inadmissible for heuristic purposes. The discussion of sub-problem/-goal decomposition is returned to in the following review of hierarchical planning.

[3]Pattern databases (Culberson and Schaeffer, 1998; Edelkamp, 2014) can be constructed to obtain a heuristic knowledge base that can generalise to different problems in the same domain (avoiding repeatedly solving the same abstract planning problem), but these can be very expensive to obtain and store. Similarly, planning graphs (Blum and Furst, 1997), polynomial approximations of a search tree (similar to ignoring delete lists in that monotonic progress to the goal can be made), can be constructed prior to planning, and then searched for heuristic estimates, which is generally cheaper than abstract planning in a relaxed model.

2.3 Hierarchical Planning

Hierarchical planning focuses on solving complex problems through divide-and-conquer, by employing abstraction to break-down problems into many smaller problems. Amoungst the most prominent techniques in hierarchical planning are Hierarchical Task Network (HTN), Hierarchical Goal Network (HGN), and Hierarchical Refinement (HR) planners.

2.3.1 Hierarchical Task and Goal Network Planning

Hierarchical Task Networks (HTNs) (Erol, J. Hendler, and Dana S Nau, 1994b) and Hierarchical Goal Networks (HGNs) (Shivashankar, 2015) are based on the idea of hierarchical and recursive decomposition of tasks or goals (Bercher, Alford, and Höller, 2019), achieved by enriching classical planning descriptions with additional decompositional methods.

Hierarchical network planners however use a different planning strategy to classical. HTN planners are given; tasks, actions, and task decomposition methods. They start with an initial network of abstract tasks to complete, each of which can be thought of as a high-level description of an objective to be achieved. Where the network is essentially a directed graph containing tasks, actions, and ordering constraints over them. Their methods then represent pre-defined recipes for the possible ways to (recursively) replace these tasks with sequences of progressively more specific sub-tasks and eventually concrete actions. HGN planners are given; goals, actions, and goal decomposition methods. They start with a network of abstract goals to achieve, and their methods are similar, but instead replace goals with sequences of sub-goals, which are then achieved by concrete actions. In either case, the planner continues to decompose tasks and goals until a classical plan (containing only actions) that achieves the initial abstract tasks/goals is found. Importantly, the recursion depth of decomposition (number of times a task/goal can be replaced by a sub-task/goal or action) is arbitrarily large, allowing for a flexible and dynamic structure to the hierarchy mechanism.

HTN planning can be more intuitively illustrated by a decomposition tree, a structure borrowed by this thesis. Such a tree shows graphically how methods (labelling horizontal arrows) decompose tasks (at higher levels) into sub-tasks and actions (at lower levels). Figure 2.1 is an example tree using recursive methods for a simple object stacking problem.

HTN and HGN planning is considered more expressive than classical planning (mainly because they can express undecidable problems (Ghallab, D. Nau, and Traverso, 2004)) and can be very efficient if effective decompositional methods are used. However, they are limited by the brittleness caused by the reliance on (the completeness of) their methods (Erol, J. A. Hendler, and Dana S Nau, 1995). Large amounts of prior knowledge is needed to pre-define methods with the best strategies for decomposing/solving problems. Further, they are only optimal, if the methods pre-define optimal decompositions. Despite this, HTN planning is popular, easy to implement, and many solvers exist (Georgievski and Aiello, 2015a), including an ASP implementation (Dix, Kuter, and D. Nau, 2003). They are particularly popular in video games, as many developers argue that HTN planning produces more natural agent behaviour (J.-P. Kelly, Botea, Koenig, et al., 2007; J. P. Kelly, Botea, Koenig, et al., 2008). Of interest, Guerilla Games' Horizon Zero Dawn uses HTN planning to simulate and control teams of multiple heterogeneous robots (Guerrilla, 2017). HGNs are much less popular than HTNs, but are claimed to be more expressive and less brittle (Alford et al., 2016).

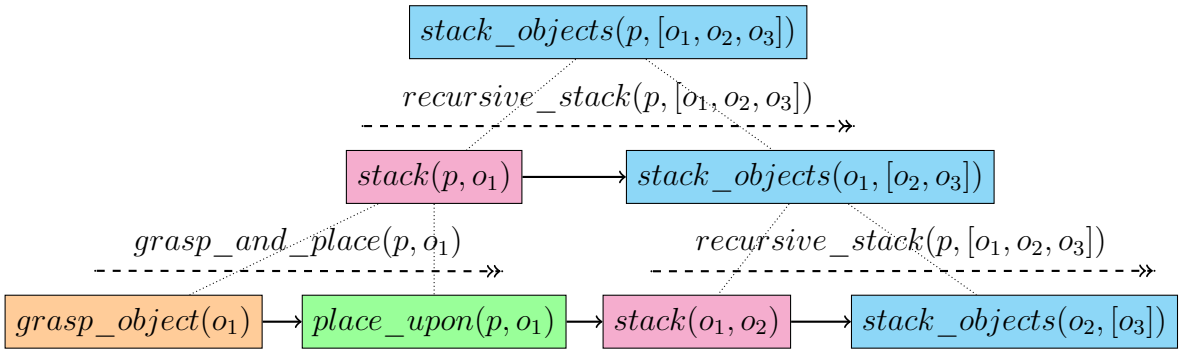


Figure 2.1: A Partial Decomposition Tree for an Object Stacking Problem. Where o_1, o_2, o_3 are objects, p a position, and [...] delimit some list. The method $recursive\_stack(p, [0_n, ...])$ recursively decomposes the task $stack\_objects(p, [o_n, ...])$ by planning to stack the first listed object in the given position, and then planning itself to stack the other objects on the first.

2.3.2 Hierarchical Refinement Planning

Hierarchical Refinement (HR) planners represent and solve a classical planning domain and problem over a hierarchy, containing multiple unique models of the domain and problem at different levels of abstraction (C. A. Knoblock, 1990b). Abstractions used in HR planning are obtained by removing the "details" of a problem, these are things which have sufficiently low importance, to be initially ignored in order to make it easier to obtain an abstract plan, and can be safely dealt with later by refining that abstract plan (Bacchus and Q. Yang, 1994). Many abstractions used in heuristic planners are also used in HR planners. However, unlike heuristic planners, HR planners don't just use abstract plans to guide search. They generate tangible abstract plans which are refined into actual solutions to the original problem.

Abstraction hierarchies for HR most commonly remove a sub-set of the constraints of a problem in each model. The planner first finds a classical plan which solves the most abstract and least constrained model of a problem. This forms a "skeleton" plan, that is then successively refined through all more concrete models, under increasingly strict constraints, until a plan that solves the original problem is obtained. Whilst the nature of the refinement process itself differs between HR planners, refinement must consider any constraints ignored in an abstraction, which are not satisfied by a skeleton plan, to be flaws that must be satisfied, by either inserting new actions or replacing existing actions in the skeleton plan.

Existing HR Planners: ABSTRIPS and ALPINE

The seminal HR planners were ABSTRIPS (an extension of STRIPS) (Sacerdoti, 1974) and ALPINE (C. A. Knoblock, 1991). The prior used relaxed abstract models which remove only action preconditions, and the latter used reduced models which remove preconditions and related facts. Both create abstractions over a STRIPS system by injecting a criticality function, "which assigns to each predicate occurring in a precondition formula an integer

value suggesting the difficulty of achieving [it]", the lower the criticality of a precondition, the longer it is ignored in the abstraction hierarchy, and therefore "the predicates having high criticality are those [...] considered first" (J. Tenenberg, 1988; Giunchiglia, 1999).

They both function similarly; within the highest abstraction, the planner plans to achieve the final-goal, with only the highest criticality action preconditions considered. Each action of the resulting skeleton plan then forms its own distinct sub-problem. The planner descends a level, reintroduces a sub-set of less critical preconditions, and calls itself recursively to refine the skeleton plan. Each call involves solving a sub-problem by inserting a sequence of new actions before the skeleton action (generating a sub-plan) to satisfy the reintroduced preconditions. Once all sub-problems at a given level are solved, their sub-plans are concatenated to form another longer skeleton plan and sequence of sub-problems. The planner descends another level, reintroduces even more preconditions, and refines the plan further. This continues until the plan satisfies all preconditions, and therefore the original problem is solved. If any sub-problem is unsolvable, the planner "backtracks" to the previous level, to find a new skeleton plan and tries to refine that (Ballard and Hartman, 1986).

HR planning using relaxed models can be illustrated by a refinement diagram. They show graphically how abstract plans (from the higher levels) contain plan flaws that require additional actions to be inserted (in the refinements at the lower levels) to satisfy the reintroduced preconditions (labelling horizontal arrows) previously ignored in the abstraction. Figure 2.2 is an example diagram for another simple object stacking problem.

As every abstraction can reduce the search space exponentially, the most abstract version of problem is trivial to solve via classical planning (C. A. Knoblock, 1990a). Refining the resulting skeleton plan into a solution to the original problem is then exponentially faster than directly solving it by classical planning, because the solution to each sub-problem involved in the refinement (a sub-plan) is linearly shorter, and therefore its complexity is exponentially less than the complete problem (Bacchus and Q. Yang, 1994; Korf, 1987). The

total complexity is the sum of complexities of all sub-problems, which is also exponentially less than the complete. If the time taken to find the abstract solution is less than the time saved by refining the abstract plan into a solution to the original problem (rather than solving the original directly), HR planning will out perform heuristic classical planning.

Limitations of Existing HR Planners

The trade-off, is that HR planners rarely find optimal solutions. The main issue is that, since abstract solutions are refined directly into actual solutions to the original problem, the quality of an abstract plan is of critical importance. Unfortunately, the criteria for what constitutes a "good" quality abstract plan are not obvious, as their quality is not known until after they are refined. Therefore, there is no clear way to decide which abstract plans are best to select for refinement, without having exhaustively refined them all.

Another fundamental problem with existing HR planners is the implicit assumption of independence between the sub-problems involved (C. A. Knoblock, 1992), which often fails in practice. Essentially, since sub-problems are solved independently and cannot "interact", their solutions are always "greedy" in that they are only locally optimal, and never consider the requirements of other sub-problems. The final complete plan is resultantly rarely globally optimal in any sense (C. A. Knoblock, J. D. Tenenberg, and Q. Yang, 1991). If there is enough dependency between sub-problems, the planner may fail to refine a plan. The problem is similar to that discussed in Section 2.2 with the sub-goal decomposition independence assumption for heuristic planning. If two actions that can undo each others effects exist, solving a later sub-problem (by achieving the preconditions of its action), might unachieve a goal achieved by an earlier action from the skeleton plan (J. Tenenberg, 1988). This makes the resulting plan invalid, as it does not satisfy all goals. There is therefore no guarantee that any given abstract plan will be refineable. Selecting which abstract plan to refine, and if or when to backtrack to select another plan to refine, can only be done on a heuristic basis.

Finding an effective abstraction hierarchy is also a significant challenge in HR planning (Bacchus and Q. Yang, 1994). Abstractions must be carefully balanced: extensive abstractions can increase abstract planning speed, but the resulting plans may be more challenging to refine, or lead planning further from the optimum. Unfortunately, the properties that make abstractions beneficial are not well understood, and as such, there is no immediate guarantee a given abstraction will improve planner performance (Smith and Peot, 2014). Further, many forms of abstraction continue to elude representation, with existing HR planners only creating abstractions and refining plans based only on action preconditions, which are not appropriate for all problems, and in particular, only reduce abstract plan lengths but not problem description sizes. This prevents the refinement process from replacing actions of a skeleton plan, and requires that actions present in the abstract models always be present in original. This prevents the use of action and state abstractions in HR planning, that are known to be of great importance to reducing abstract problem complexity in heuristic planning. Further, the requirements for actions of a skeleton plan to be present in a refined plan, is what restricts existing HR planners only to single action refinement sub-problems.

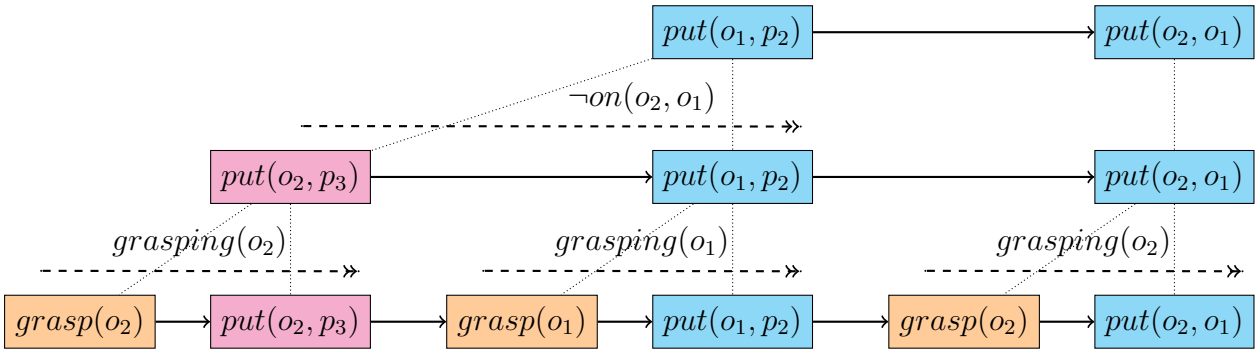


Figure 2.2: A Plan Refinement Diagram for an Object Stacking Problem. Where o_1, o_2 are objects, and p_1, p_2, p_3 are positions on a table where the objects can be placed. The initial state is $on(o_1, p_1), on(o_2, o_1)$ (objects stacked on p_1 with o_1 at the base and o_2 on top) and the goal is $on(o_1, p_2), on(o_2, o_1)$ (move stack to p_2). The middle-level removes a precondition requiring an object to be grasped before it is moved and the top-level removes a precondition preventing objects with another object on top of them being moved. The top-level classical plan puts o_1 onto p_2 immediately (despite that o_2 is on top of it) and then puts o_2 on top of o_1. In the refinement at the middle level, o_2 first needs to be taken from the top of o_1 with the additional pink put action before o_1 can be moved. In the refinement at the ground level, all actions then require one of the proceeding orange grasp actions to enable them.

2.4 Answer Set Programming

Answer Set Programming (ASP) is a declarative non-monotonic logic programming paradigm. It is well established as a prominent technique for knowledge representation and reasoning, and is particularly effective for combinatorial optimisation problems with large knowledge bases (Kaminski, Schaub, and Wanko, 2017). Its successfulness is typically attributed to the generality, expressiveness, and elaboration tolerance of the modelling language, and to the now widespread availability of highly efficient solvers (Gebser, Leone, et al., 2018).

The central concept is to represent a problem as a logic program, whose models, called answer sets, are the solutions to that problem (Gelfond and Lifschitz, 1988; Lifschitz, 2008). Such a problem is defined declaratively; whereby one defines only its rules and constraints, and the manner of searching for a solution is left unmentioned. ASP solvers can then find solutions to any given problem in a general way[4] (Gebser, Maratea, and Ricca, 2019).

In contrast to first-order logics, the non-monotonic logic used in ASP can support reasoning about unknowns (lack of support for truth does not necessarily imply falsehood) and the ability to update beliefs as new knowledge is gathered and integrated into a program (Gebser, Janhunen, et al., 2015). In ASP, the logical connective *not* symbolises negation as failure (*naf*), also called default negation. Unlike classical negation, which stands for "p is false", *naf* stands for "there is no reason to believe p is true" (this is consistent with p being false or unknown) (Clark, 1978). This supports defeasible processes, such as abductive and commonsense reasoning, where defaults and assumptions can be used to resolve inconsistencies resulting from uncertain or incomplete knowledge (Gelfond and Lifschitz, 1991).

Note 2.4.1 *The reader does not need a deep understanding of ASP or non-monotonic logic to understand the content of this thesis. The overview presented in this section, and a basic understanding of first-order logic, discrete mathematics, and set theory, will be sufficient.*

[4]This thesis is not concerned with the specifics of how ASP solvers work, or their history/development; it is only concerned with how they can be used to solve complex problems efficiently and in a general way.

2.4.1 Syntax and Semantics

An ASP logic program Π contains a set of logic rules of the abstract form $head \leftarrow body$. An answer set of a program is a set of atoms $\zeta = \{a_1, \dots, a_q\}$ which simultaneously satisfy all program rules. Whereby, if rule's body is satisfied, then so must its head. If a program has an answer set it is *satisfiable*, otherwise it is *unsatisfiable* (Gelfond and Lifschitz, 1988).

The body of a rule is a (possibly empty) logical conjunction and the head is either; empty, a single atom, a disjunction of atoms, a choice set, or an aggregate (Calimeri, W. Faber, et al., 2012). The following are the general forms of all standard types of rules:

$$a_0 \leftarrow \tag{2.2}$$

$$a_0 \leftarrow a_1, \dots, a_n, not\ a_{n+1}, \dots, not\ a_o \tag{2.3}$$

$$a_0 \mid \dots \mid a_m \leftarrow a_{m+1}, \dots, a_n, not\ a_{n+1}, \dots, not\ a_o \tag{2.4}$$

$$\{a_0, \dots, a_m\} \leftarrow a_{m+1}, \dots, a_n, not\ a_{n+1}, \dots, not\ a_o \tag{2.5}$$

$$\#agg\ c_1\ \{a_0, \dots, a_m\}\ c_2 \leftarrow a_{m+1}, \dots, a_n, not\ a_{n+1}, \dots, not\ a_o \tag{2.6}$$

$$\leftarrow a_1, \dots, a_n, not\ a_{n+1}, \dots, not\ a_o \tag{2.7}$$

Where a is an *atom* of form $p(t_1, \dots, t_r)$ or its classical negation $\neg p(t_1, \dots, t_r)$, p a *predicate* of arity r, and t a term. A term is a constant, variable, integer-valued mathematical expression, or function symbol. A constant is either an integer or a lowercase string, and a variable is an uppercase string. A function symbol is a symbol $d(t_1, \dots, t_r)$. Predicate and function symbols will be abbreviated to $p(\bar{t_r})$ and $d(\bar{t_r})$ respectively, where $\bar{t_r}$ is a r-length vector of terms. The atoms $a_0, \dots, a_m$ are the head atoms of the rules, and $a_{m+1}, \dots, a_n$ and $not\ a_{n+1}, \dots, not\ a_o$ are *positive* and *negative body literals*, respectively (a literal is negative *iff naf* negated). The notation $\#agg$ denotes an aggregate head, where $\#agg \in \{\#count, \#sum\}$ expressing cardinality and weight constraints, and c_1 and c_2 are upper and lower bounds (see Dell'Armi et al., 2003 and Page 31 Gebser, Kaminski, Kaufmann, Lindauer, et al., 2022b).

A rule with a head atom and no body is a *fact* (2.2), all answer sets must satisfy its head as the body is always satisfied. A rule with exactly one head atom and at least one body literal is a *normal rule* (2.3), an answer set must satisfy its head if its body is satisfied. Where intuitively (but informally), a positive literal or head atom is satisfied *iff* it is in the answer set $a_i \in \zeta$, and a negative literal is satisfied *iff* it is not in the answer set $a_i \notin \zeta$. In planning, normal rules may be used for example to define action effects, e.g. the axiom "when a robot moves its location changes". A rule with multiple head atoms, is either a *disjunctive rule* (2.4) where the | denotes "exclusive or" such that exactly one of the head atoms must be satisfied if the body is, or it is a *choice rule* (2.5) where any sub-set of the atoms $\{a_0, \ldots, a_m\}$ (including the empty set) can be satisfied if the body is. An *aggregate rule* (2.6) is essentially a cardinality or weight constrained choice rule. A rule with no head atom and at least one body literal is an *integrity constraint* (2.7), no answer set can satisfy its body as the head can never be satisfied. In planning, an integrity constraint may be used as state constraints, e.g. the axiom "two objects cannot be in the same place".

Finding Answer Sets

Finding the answer sets of an ASP program is typically done in two steps; grounding and solving. In program grounding, all non-ground rules (containing variables) are replaced by ground versions by substituting the variables with constants, obtaining a ground program which can be solved. The ground program is typically of between polynomial to exponential size with respect to the original non-ground input program. In program solving, a search is then ran to find answer sets that satisfy the set of ground rules. Solving ASP programs to compute answer sets is founded upon the concepts of Boolean Constraint and Propositional Satisfiability (SAT) solving, and based upon the classical Davis-Putnam-Logemann-Loveland (DPLL) (Davis, Logemann, and Loveland, 1962) procedure, but current modern solvers now employ conflict-driven solving procedures (Gebser, Kaufmann, and Schaub, 2009) based on Conflict-Driven Clause Learning (CDCL) (Marques-Silva and K. A. Sakallah, 1999).

This thesis uses the modern ASP system *Clingo* series 5 (Gebser, Kaminski, Kaufmann, and Schaub, 2014) which integrates the *Gringo* series 5 grounder and the *Clasp* series 3 solver. Clasp supports parallel search via shared memory multi-threading (Gebser, Kaufmann, and Schaub, 2012b), and parallel prioritised optimisation of disjunctive logic programs (Gebser, Kaminski, Kaufmann, Romero, et al., 2015). Clingo provides an API with bindings in C, C++, Lua and Python, giving imperative control over the grounding and solving processes (Gebser, Kaminski, Kaufmann, Lindauer, et al., 2022a; Kaminski, Schaub, and Wanko, 2017); this enables incremental (multi-shot) solving capabilities (Gebser, Kaminski, Obermeier, et al., 2015; Gebser, Kaminski, Kaufmann, and Schaub, 2019), allowing program groundings to be re-used and successively expanded with additional knowledge.

Examples of ASP Programs and their Answer Sets

The following are some minimal sufficient examples of ASP programs and their answer sets. Where a program is written as a set of rules and a dot "." terminates each rule.

The program $\{a.\}$ is satisfiable and has one answer set $\{a\}$, because a is a fact. The program $\{b \leftarrow a.\, a.\}$ containing a normal rule, is satisfiable and has one answer set $\{a, b\}$, because b can be derived from a. Removing a, the program $\{b \leftarrow a.\}$, is also satisfiable and has one answer set $\{\,\}$, but this is the empty set because b cannot be derived since a is not known to be true. The program $\{\leftarrow b.\, b \leftarrow a.\, a.\}$ containing an additional integrity constraint, is however unsatisfiable and therefore has no answer set, because deriving b from a would violate the integrity constraint. The program $\{\{b\} \leftarrow a.\, a.\}$ is satisfiable and has two answer sets $\{a, b\}$ and $\{a\}$, this is because deriving b from a is a choice, and therefore can, but does not have to be present in an answer set. The program $\{\leftarrow b.\, \{b\} \leftarrow a.\, a.\}$ is satisfiable but now again only has one answer set $\{a\}$. The program $\{a(V) \leftarrow b(V).\, b(1).\, b(2).\}$ containing variables (non-ground rules), is grounded to the program $\{a(1) \leftarrow b(1).\, a(2) \leftarrow b(2).\, b(1).\, b(2).\}$ (with variables expanded), is satisfiable and has one answer set $\{a(1), a(2), b(1), b(2)\}$.

2.4.2 Classical Planning with ASP

ASP is highly applicable for discrete deterministic planning because it supports the complex reasoning capabilities (such as deriving the indirect effects of actions, reasoning about state constraints, or dealing with recursive relations) and can handle the large amounts of facts and relations, that must be dealt with by a planning agent. Conversely, these can be very difficult to manage in imperative programming languages such as C++ or Python.

In ASP, a planning domain is modelled axiomatically, as a program declaring only the physical laws that govern the dynamic domain the robot operates in. This allows the robot to reason for itself about how it can act upon and transition the state. A classical ASP planning problem can then be obtained by adding an initial state and goal condition to the program. The solver searches for a plan that satisfies the domain laws, and transitions from the initial state to some goal state (solving the planning problem), using general heuristic methods (Gelfond and Kahl, 2014a). For example, consider the simple planning domain:

$$\{move(Robot, Room, I)\} \leftarrow \tag{2.8}$$

$$in(Robot, Room, I) \leftarrow move(Robot, Room, I) \tag{2.9}$$

$$\leftarrow move(Robot, Room_1, I), in(Robot, Room_2, I-1), \tag{2.10}$$
$$not\ connected(Room_1, Room_2)$$

$$\leftarrow not\ \#count\ 1\ \{in(Robot, Room, I)\}\ 1 \tag{2.11}$$

$$\leftarrow in(Robot, Room_1, I), in(Robot, Room_2, I) \tag{2.12}$$

$$\{in(Robot, Room, I)\} \leftarrow in(Robot, Room, I-1) \tag{2.13}$$

These specify an action schema (where I denotes time) such that: the robot can plan a move action that changes its location only between rooms connected to the room it is currently in (rules 2.8-2.10), the robot must be in some room at any given time but cannot be in two rooms at once (rules 2.11-2.12), if it does not move it stays in the same room (rule 2.13).

The paradigm is therefore unlike classical rule based expert systems. The program rules do not tell the robot what to do or how to do it, but instead tell it what it can do, what its capabilities are, and what constraints its domain imposes upon it. This axiomatic modelling allows compact expression of complex dynamic domains in an intuitive manner, including those with many interacting constraints, in a way that allows new knowledge to be added without having to modify existing knowledge[5]. A robot equipped with such a model, then has the capacity to generalise from these axioms to solve arbitrary problems without the need to express the search strategy or obtain a domain-specific heuristic.

Classical planning with ASP has proven effective in many applications (Esra Erdem, 2016; Grasso, Leone, and Ricca, 2013), such as; robotics (Erdem and Patoglu, 2018), healthcare (Dodaro, Galatà, et al., 2018; Erdem, Erdogan, and Öztok, 2011; Gebser, Guziolowski, et al., 2010) and industry (Falkner et al., 2018). This includes; planning and monitoring for simulated teams of house keeping robots (Aker, Patoglu, and Erdem, 2012), a delivery robot system built on an ASP extension of the Robot Operating System (ROS) (Quigley et al., 2009; Andres, Obermeier, et al., 2013; Andres, Rajaratnam, et al., 2015), coordinating teams of unmanned aerial vehicles (Balduccini, Regli, and Nguyen, 2014), general PDDL planning (Dimopoulos et al., 2017), and real-world service robots (Khandelwal et al., 2017).

Despite their success, existing approaches to planning using ASP have continued to prove intractable for problems with very long minimal plan lengths (see Jiang et al., 2019 for a thorough empirical evaluation of the performance of ASP based planning). Whilst ASP performs well on highly constrained problems (where the constraints of the problem can be exploited to eliminate large amounts of the search space), the size of the ground logic program always explodes exponentially in the length of the plan. In contrast, classical heuristic planners tend to perform better for problems with long plan lengths[6], but are less effective at handling large problem descriptions and are less expressive than ASP. Notably,

[5]E.g. expanding the above to allow object manipulation, does not require changes to the current rules.

[6]Note that it is difficult to directly compare classical ASP planners with heuristic planners (based on the PDDL) since they use different encodings (it is hard to define the exact same problem) and search strategies.

classical heuristic planners lack the capacity to express state constraints. The other main flaw of ASP is its poor performance when dealing with action costs, and its inability to handle continuous costs. ASP is therefore typically only effective for problems where the primary criteria for plan quality are the length of and quantity of actions in a given plan[7].

2.4.3 Search Strategies for ASP based Planning

There are two ways to search for a solution to a planning problem represented by an ASP program: one-shot and incremental (sometimes called multi-shot) search.

In one-shot search, the ASP grounder and solver are called only once. The maximum search length must be pre-defined and fixed prior to search, to obtain a finite-sized grounding. One-shot search is easy to implement, but the fixed search length is problematic for two reasons: a) the plan length that solves a given problem is not known and hard to predict before search, and b) optimisation is needed to guarantee the minimal length plan is found. If the search length is less than the minimal plan length, then the planner will fail, and the program grounding cannot be re-used for another solve (with a longer search length).

Incremental search allows the ASP grounder and solver to be called multiple times, by incrementally expanding the program grounding, therefore removing the need to know or predict the maximum search length. Search starts with a search length of 1 and increases to some limit $k \in \mathbb{N} : k \geq 1$. For each step $i \geq 1$, the program is then solved to check for a solution of that length. At the first step where the program is satisfiable, the planner returns having found the guaranteed minimum length plan[8] (without optimisation). Because the program grounding is re-used and expanded only to the minimum search length, this ensures a minimum size grounding and overall grounding time. However, the implementation is more complex, requiring extra imperative code to control the incremental solve.

[7] If other quality criteria are needed then they must be expressed as a discrete preference relation over actions (since enforcing a preference relation can be reduced to a combinatorial optimisation problem).

[8] This is conceptually very similar in nature to how an iterative deepening depth-first search works.

2.4.4 Hierarchical Planning and Abstraction in ASP

Representing and reasoning with abstraction hierarchies for HR planning in ASP has seen attention in the literature, particularly within the context of ASP based "action languages"[9]. There exists only a few such publications which are relevant to this thesis. However, these have provided substantial inspiration and foundations for its core contributions.

Representing Transition Systems at multiple levels of Abstraction

Sridharan, Gelfond, et al. (2019) proposed the REBA system, which represents a planning domain as two "tightly-coupled" state transition systems simultaneously, both of which could be used for planning. REBA's focus was on the state representation, using their (rather unfortunately named) "refinement" models, written in the action language $\mathcal{AL}_d$ (Gelfond and Inclezan, 2013). The system used a domain-specific state abstraction, in which the abstract-level was a "coarse" resolution model, and the original-level a "fine" resolution model, of the domain, each connected by a many-to-one upwards state space mapping called "bridging axioms". This allowed for example, a representation in which rooms are defined as being composed of a set of cells. Whereby, at the coarse level, the location of objects is considered only relative to what room they are in, and at the fine level, the location is considered relative to specifically what cell of the room they are in. This creates an abstraction which reduces the quantity of possible states, without entirely deleting properties of the state representation. Sridharan, Gelfond, et al. (2019) showed that ASP can model more powerful state abstractions than those common in most heuristic and HR planners.

The nature of planning in REBA is quite different to HR planning. ASP is not used directly to refine plans. Instead the technique is to find a coarse resolution ASP plan and then use probabilistic decision making to implement each abstract action at the fine resolution[10].

[9] An action description language (Gelfond and Lifschitz, 1998) is a high-level natural language notation for ASP programs used for describing state transition diagrams of dynamic systems (Babb, 2015).

[10] This extended similar past work (S. Zhang, Sridharan, and Wyatt, 2015; S. Zhang and Stone, 2017).

This allows navigation planning to be broken down to finding a coarse ASP plan between rooms, which is then refined to a fine probabilistic plan between the cells of those rooms.

The system, only supported a two-level hierarchy, primarily because abstraction levels were not represented quantitatively or explicitly within predicates. Instead the prefixes "course" and "fine" had to be manually added to function symbols representing state literals. Further, REBA's abstractions were defined only by relations between entity constants, and not their class types. As a result, all abstract domain laws for the coarse resolution had to be re-written manually. REBA's creators claim that it would be possible to extend their theory to representing more than two levels. However, this would imply some ability to refine ASP based plans at the higher levels, since probabilistic decision making can only occur at the lowest level. No clear mechanism or semantics for either such technique was provided. Unfortunately, the available code snippets for REBA do not fully implement the proposed system and do not correspond to the theory. ASP planning occurs only at the fine level, whilst the coarse level is entirely defined in terms of the fine, which cannot be acted upon by the agent. Moreover, no mechanism for changing the level at which planning occurs is obvious, without having to manually rewrite large amounts of the provided code.

Constraint Based HR Planning with Action Language BC

S. Zhang, F. Yang, et al. (2015) used a three-level abstraction hierarchy written in the action language $\mathcal{BC}$ (Lee, Lifschitz, and F. Yang, 2013) for HR planning in a logistics domain. A novel constraint addition based approach to abstract plan refinement was used, in which abstract actions "bottlenecked" search whilst solving the original problem. This was done by encoding abstract action effects as integrity constraints that must be achieved before fixed time steps during plan refinement. Resultantly, abstract actions did not have to be included in more concrete plans, and the planner was able to replace them with better quality actions found during plan refinement. The method first generates a plan at the most abstract level

of 1, and then refines that plan in ascending order, until the most concrete level of 3 is reached[11]. S. Zhang, F. Yang, et al. (2015) propose two algorithms: PlanHG, which refines all actions in the abstract plan in one call by satisfying all state constraints simultaneously; and PlanHL, which refines each abstract action separately by recursively calling the planner in a depth-first manner. It is claimed that this technique gave "orders of magnitude" faster planning, for a loss in plan quality of only approximately $\approx 11.25\%$ on average.

S. Zhang, F. Yang, et al. (2015) proved that hierarchical plan refinement and problem division for HR in ASP is possible. Their primary contribution is the effectiveness and flexibility of the constraint based plan refinement technique, which significantly reduces overall planning time whilst maintaining good plan quality. Unfortunately, they only support sub-problems that refine single abstract actions (causing exaggeration of the dependency problem) and they failed to leverage problem division for online planning. The major flaw with their theory was however, that it required the pre-computation of a step bound estimation function. This function was used to determine the time step by which each abstract action effect must be achieved in a refined plan. This function requires the maximum length of a refined sub-plan for all abstract actions in a given problem to be known prior to planning[12], which appears to require generation of all possible state transitions over all levels in the hierarchy. This would clearly be intractable to compute for any non-trivial problem. The cost of obtaining this function is unfortunately not included in the results.

There is also no description given of how the abstraction hierarchy is represented or constructed. This implies that a separate ASP encoding is used for each abstraction level, but this cannot be verified as the encoding is not provided. They further appear to have designed their abstraction hierarchy in a reverse manner, that is, they have started by choosing a well performing abstract model, and then designed more concrete models that are appropriate for the abstract model. However, this is counter intuitive, as an abstraction

[11]This is the opposite of how an abstraction hierarchy is usually defined, typically the highest level is the most abstract level, and hierarchical planning progresses in descending order (i.e. from top to bottom).

[12]They refer to this as the minimal number of steps needed to optimally achieve the effects of the action.

hierarchy should be constructed in an upwards manner. One must start with a model of the actual practical problem to solve, and abstractions should be created to simplify its solving. Finally, since no mapping was used between the state representation at different levels of the hierarchy, the integrity constraints defining the effect of abstract actions must also be present in the original model, limiting state abstractions to only those that remove state variables, and preventing those that modify them (such as REBA's models).

Unfortunately, since no implementation, complete domain and problem encoding, and no methodology for generating or storing their critical step bound estimation function, was published with the work, it has not been possible to directly continue this research.

Generating Generalised Plans by Abstraction under ASP Semantics

Saribatur (2020) and Saribatur and Eiter (2018a) explored abstraction for ASP based planning in the context of the computation of generalised abstract plans which "get to the essence of a problem [...] by disregarding irrelevant details and computing abstract plans [...] with the potential to understand the key elements of a [ASP] program that play a role in finding a solution [to a planning problem]". Their discussion focuses on: a) the concept of relevance, whereby irrelevant details must be removed from a problem to keep its complexity manageable; and b) the notion of generalisation, in which the common and distinguishing properties of objects are used to group or split them apart in the reasoning process to further manage complexity and ease abstract reasoning. Saribatur (2020) claim their method is close to the human use of abstract thinking for problem solving, which is core to efficient planning.

As an example of the theory, consider a blocks world problem with multiple tables and sets of blocks to be stacked, encoded as an ASP program. A problem-specific abstraction can be formed over object constants (entities) by generating an abstract program, which groups together all tables that are not part of the goal (and therefore irrelevant to the problem), into one constant. This would allow abstract planning to avoid considering all irrelevant tables

in detail, reducing the problem description size. Saribatur and Eiter (2021) then propose an abstraction mapping which links all atoms in the original and abstract ASP programs. This method allows abstract plans to be linked to their refinements directly through their actions (which requires also abstracting over the time domain). The work of Saribatur and Eiter (2021) is however only theoretical and has not been implemented. Yet, its foundations on previously implemented abstractions for ASP programs (Saribatur, Schüller, and Eiter, 2019; Eiter, Saribatur, and Schüller, 2019) supports the soundness of the theory.

2.5 Summary

ASP has been proven highly appropriate for representing and solving discrete deterministic planning problems. ASP is effective for problems with many complex interacting constraints and large problem descriptions. Unfortunately, ASP based planning is fundamentally limited because it performs quite poorly for problems with very long minimum plan lengths.

Hierarchical planning, problem decomposition, and the use of abstraction, are highly prominent in many aspects of the research on automated planning, and have been proven as powerful tools in reducing complexity of problem solving. Past research has attempted to combine ASP based planning with these techniques to overcome its limitation. This has shown that ASP is appropriate for representing hierarchies and reasoning with abstractions, and that HR planning can be elegantly achieved in ASP via a constraint addition approach. However, existing approaches have been obscure, impractical, and had poor generality.

This thesis proposes and fully implements the HCR planning paradigm, a novel approach for combining ASP and HR planning. The core goals are to overcome the limitations of past works and improve the speed, performance, versatility, and scalability of ASP based planning towards being effective for practical applications. The first section of the following chapter describes the specific details of how HCR planning improves upon past work.

Chapter Three

Fundamentals of HCR Planning

This chapter introduces the core fundamental theoretical foundations used in Hierarchical Conformance Refinement (HCR) planning. The design is described and exemplified in the context of the implementation contributed by this thesis, the Answer Set programming based Hierarchical conformance refinement planner (ASH), introduced in Section 1.3.4.

3.1 Preliminary Definitions and Terminology

In classical planning, a plan is a single linear finite-length sequence of discrete actions. Such a plan is henceforth called a monolevel plan. In HCR planning, plans are also arranged in hierarchies, consisting of multiple monolevel plans at different levels of abstraction. Such an arrangement is called a hierarchical plan. The structure of hierarchical plans is such that; higher levels contain short and generalised actions and plans, and lower levels contain longer expanded and more specialised actions and plans. The ground-level plan is the desired solution to the original planning problem, which is executed by the robot(s). The objectives are to, minimise the time taken to find and maximise the quality of, the ground plan.

A classical plan is always complete, whereby it starts in the initial-state and must achieve the goal. The single "end" goal used in classical planning will henceforth be referred to as the final-goal. This is because HCR planning is also enriched with sequences of sub-goal stages. These are intermediate goals, which must be achieved by a plan in the correct order, towards the path to the final-goal. Resultantly, HCR problems and plans can also be partial.

That is, a partial-plan can solve only part of the complete problem, because a partial-plan can achieve only a sub-sequence of the sub-goal stages. Complete refinement plans achieve the final-goal from the initial state, and also pass through a sequence of intermediate states that satisfy the sub-goal stages in order. Partial refinement plans do not have to start in the initial state or reach the final-goal. They can start from any sub-goal stage achieving state, and end in any later sub-goal achieving state, progressing the planner partly towards the final-goal. A sequence of partial-plans can then be concatenated to form a complete plan. HCR planning therefore occurs over two axes; the vertical and the horizontal. The vertical axis represents the descending progression of abstract plan refinement over the abstraction hierarchy (from top-to-bottom), and the horizontal axis represents the ascending progression of partial-plan extension along the sub-goal stage sequence (from left-to-right).

3.2 Novelty of HCR over Past Work

HCR planning combines the strengths of ASP and HR planning, with the goal to achieve both generality and speed. It overcomes the significant limitations of existing HR planners, including those implemented in ASP, whilst generalising the concept to support rapid online planning. The following details the novel contributions of HCR over past work in depth.

Previous HR planners such as ABSTRIPS and ALPINE (discussed in Section 2.3.2) refined abstract (skeleton) plans based on achieving preconditions of abstract actions. This limited them to using abstractions based only on the removal of action preconditions, and supported only single-action refinement sub-problems. HCR planning instead refines abstract plans based on achieving the effects of abstract actions, a process enforced by the conformance constraint. This is a planning constraint that require a refined plan to achieve the same effects and remain structurally similar to an abstract plan. The conformance refinement method is inspired by past work on constraint based HR planning in ASP (discussed in Section 2.4.4). This past work did allow constraints other than preconditions to be removed in abstractions,

and supported state abstractions. However, the rigid integrity constraint based method that guided plan refinement caused three main problems: it a) only supported single-action refinement sub-problems; b) required a constraint on the achievement time of abstract action effects to be pre-calculated prior to plan refinement; and c) only supported abstractions that removed, but not modified or generalised, problem constraints, actions, and state variables. The following section details how the conformance constraint overcomes these limitations.

3.2.1 Refinement Based on Conformance Constraints

HCR planning proposes a fundamentally novel mechanism of plan refinement and problem division. In past HR planners, a complete abstract plan that achieves the final-goal was only generated at the most abstract level. Abstract plans were then refined directly, whereby the refinement process required inserting new actions to satisfy any previously ignored problem constraints. This required the same actions to exist in the abstract and original models, and only allowed refinement planning to expand plans by adding actions. In HCR planning, a complete plan (possibly by concatenating multiple partial-plans) that achieves the final-goal is generated at all levels. Abstract plans are refined by conformance, whereby abstract action effects are passed as sequences of sub-goal stages to refinement planning. The refinement process requires generating a completely new plan that both achieves the final-goal, as well as the sequence of sub-goal stages in the same order as the actions that produced them, on the path towards the final-goal. This approach is more flexible, because it only tells the planner what needs to be achieved, and in what order, rather than forcing the planner to include particular actions in a plan refinement. In addition, it removes the need to apply fixed constraints on the achievement time of abstract action effects. By introducing a state abstraction mapping over state variables, action and state abstractions can change the entire problem representation used in an abstract model. This means that completely different actions can exist in the abstract and original models, allowing refined plans to expand in length, and existing actions to be specialised into or replaced by more detailed variants.

The primary novel contributions of the conformance constraint are:

- *High quality satisficing planning*: The flexibility provided by the goal-sequence enables the interleaving property. This allows the simultaneous consideration and pursuit of sub-goal stages included in the same problem. The interleaving property enables the achievement of earlier sub-goal stages in the same problem to be delayed, to better prepare the resulting state for achieving later sub-goal stages sooner, if doing so reduces overall plan lengths. The dependencies that were highly exaggerated by existing HR planners are therefore heavily alleviated. The goal-sequence used in HCR planning also removes the restriction where each individual abstract action had to be refined as a separate sub-problem, and makes problem division optional, as follows:
  - When problems are not divided, a complete refinement problem is solved. Whereby, all abstract actions are refined simultaneously and a global problem is considered. Dependencies between sub-goal stages are thus entirely eliminated, by allowing global interleaving, increasing search efficiency for little loss in plan quality.
  - When complete problems are divided, they can be broken into partial-problems, each refining a contiguous sub-sequence of sub-goal stages, thus still constituting a multiple action refinement problem. However, a series of local problems must now be considered, and dependencies are only partly alleviated, since only dependent sub-goal stages in the same partial-problem are allowed to interleave. Consequently, solutions to partial problems can be non-optimal, since the last sub-goal stage in a partial-problem is "greedily" achieved, without consideration for the requirements of sub-goal stages in later partial-problems or the final-goal. This can cause later partial-plans to unachieve components of the final-goal achieved by earlier partial-plans, reducing the quality of the refined plans. However, a general heuristic mechanism is employed, which biases planning during early partial-problems to pre-emptively achieving final-goals where possible. This helps to achieve better quality partial-plans, as they are extended to the final-goal.

- *Rapid online planning*: Complete refinement planning improves the planning speed only from the restriction on the search space provided by the conformance constraint. Online partial refinement planning gains much greater speed by reducing the minimum plan length of each partial-problem linearly, and thus their complexity exponentially. By initially only generating a partial-plan at each level also allows the planner to rapidly propagate to the ground-level. A robot can begin execution with just the first partial-plan, which can then be extended to the final-goal online (during execution). This can reduce execution latency and total planning times exponentially as follows. Let the complexity of a complete planning problem be $O(b^d)$ where b is the average branching factor and d is the plan length, which is exponential in d. Then partial-planning can reduce the complexity to $O(n * b^{d/n})$ where n is the number of partial-problems. This is equivalent to $O((d/p) * b^p)$ where $p = d/n$ is the average partial-plan length, which is linear in d, if n increases proportionally so that p remains constant given d.

- *Support for concurrent action planning*: Concurrent action planning allows more than one action to be planned per time step. This can occur if a set of actions, which would form a sub-sequence in a sequential action plan, are arbitrarily ordered, whereby their order could be permuted and still be a valid plan. HCR provides a set of general constraints to handle concurrent action planning. This is beneficial because it compresses plan lengths and thus may reduce planning complexity and time. Moreover, because it does not impose a strict ordering over achieving the effects of arbitrarily ordered actions, it provides flexibility to plan refinement, which may increase plan quality.

3.2.2 Generalised Hierarchy Representation

HCR planning aims to support a large variety of abstract domain models, to make the approach appropriate for as wide range of domains and problems as possible. To support action and state abstractions able to modify actions, state variables, and domain laws,

Sridharan, Gelfond, et al. (2019) and Saribatur (2020) propose different methods of mapping together arbitrarily different plans between abstractions. HCR planning instead introduces a new state abstraction mapping between state variables in different domain models. This maps actions of an abstract plan to those of their refined plans through their effects (encoded as sub-goals), and never directly through actions. The mappings allow reasoning about the achievement of sub-goals defined by an abstract state representation, even when an action or state abstraction is used. Such an abstraction can modify actions, state variables, and domain laws, such that the abstract representation is entirely different to the original. The hierarchical representation of HCR therefore supports any abstract model to which a deterministic and exhaustive state space mapping exists, whereby each original state maps to exactly one abstract state, and each abstract state is mapped to from at least one original state. This greatly increases the generality of HCR over previous HR planners.

HCR planning can support an arbitrarily large abstraction hierarchy and allows the entire hierarchy to be represented by one ASP program. This is because, unlike previous systems, abstraction levels are represented explicitly and quantitatively in predicates representing actions, state, and domains laws. Any action, state variable, or domain law that is not modified, replaced, or removed in an abstract model, is preserved implicitly, and has its abstract version generated automatically from the non-ground ASP program, without the need to manually re-write it. The planning algorithms can then automatically ground the hierarchy at any abstraction level to use any domain model(s) needed for planning.

This thesis further contributes condensed abstract domain models, as introduced in Section 1.3.2 and discussed further in Section 3.5.2. They apply action and state abstractions that automatically modify actions and state variables for the abstract model, to increase the granularity of the representation. Condensed abstract domain models extend past literature (Sridharan, Gelfond, et al., 2019), primarily to ease the specification of the models through a class-based relation. This relation enables the abstract generalized versions of the actions, state variables, and domain laws in the condensed model to be generated automatically.

3.2.3 The Operational Modules of ASH

ASH is designed to be modular to promote extensibility. The operational modules of ASH are its domain-independent ASP parts. The modules can solve any hierarchical planning domain or problem given to ASH in a general way. The modules include constraints that define how the search for plans occur in a declarative manner. These constraints are called planning constraints, as they are general definitions for what constitutes a state, state transition, and a plan. They differ from problem constraints, which are definitions for the nature of and what constitutes a solution to a specific planning problem. This design greatly reduces the implementation complexity of the necessary imperative search and control algorithms[1].

ASH is built in Python and the Clingo ASP system (Gebser, Kaminski, Kaufmann, Lindauer, et al., 2022b; Gebser, Kaminski, Kaufmann, and Schaub, 2014). This obtains a light-weight and highly portable planner, which combines the declarative KRR and general problem solving power of ASP, with the clear and concise imperative control of Python through the Clingo API (Gebser, Kaminski, Kaufmann, Lindauer, et al., 2022a). For the purposes of this thesis, Clingo gives two particularly powerful features of interest:

1. *Incremental solving*: which allows planning when the maximum planning horizon (minimum plan length) is not known[2] (Gebser, Kaminski, Kaufmann, and Schaub, 2019).
2. *Selective grounding via program parts*: which allows modularisation, parameterisation, and dynamic modification (during incremental solving), of generalised ASP programs.

The operational modules of ASH are program parts which group together the ASP rules that define how HCR planning works based on their function. Most importantly, these rules encode the definitions of what it constitutes for a plan to be the solution to a classical or conformance refinement planning problem[3]. The modules function as follows:

[1] Usually, in classical planners, the search algorithm is written entirely imperatively, which can cause proliferation in code complexity and reduction in elaboration tolerance as features are added to the planner.

[2] See Section 2.4.3 for a brief explanation of the benefits of incremental ASP solving over one-shot solving.

[3] These definitions are presented later in Section 3.6 and the rules of the modules in full in Section 4.4.

1. *Instance Relations Module* ($\mathcal{IRM}$): Creates the instance relation constraints, containing type constraints and ancestry constraints. These control what entity constants replace variables occurring in rules. In particular, it ensures the correct and automatic generation of the abstract domain laws of condensed domain models.

2. *State Representation Module* ($\mathcal{SRM}$): Ensures that the state representation is always complete and valid. When refinement planning, it ensures the state is simultaneously represented over an adjacent pair of abstraction levels, allowing reasoning about conformance with abstract plans containing action or state abstractions.

3. *Plan Generation Module* ($\mathcal{PGM}$): Generates actions, ensuring state transitions are legal, and plans are "classical complete" whereby they achieve the final-goal. Further, it contains the general concurrency conditions, that define when a set of actions can be planned on the same time step (this is desirable because it compresses plans and better represents the ordering constraints over the actions in a plan).

4. *Conformance Refinement Module* ($\mathcal{CRM}$): Ensures that refined plans conform to the abstract plan from the previous adjacent abstraction level which it refines, by generating and enforcing the conformance constraints. This requires that plans are "conformance complete" whereby they achieve all included sub-goal stages in order.

5. *Optimal Planning Module* ($\mathcal{OPM}$): Optimises plans under two metrics of plan quality; the minimisation of the number of actions during concurrent action planning, and maximisation of pre-emptively achieved final-goals during online planning.

6. *Final-Goal Generation Module* ($\mathcal{FGM}$): Generates a conforming final-goal definition across the whole abstraction hierarchy from only a ground-level specification. This therefore ensures that all plans at every level correctly end in a state where all state literals in the final-goal map to each other. This avoids the need for the designer to manually specify the final-goal individually at every level. The module also helps to diagnose problems in the specification if a consistent final-goal cannot be found.

3.3 Appropriate Domains for HCR Planning

A domain and problem are appropriate for HCR planning for a meaningful abstraction of them can be formed. HCR planning is therefore designed for complex problems with many complex interacting constraints. These are the problems that promote the use of abstractions, because abstractions are effective for simplifying reasoning about and accounting for those constraints. It is important to note that the nature of the abstraction hierarchy must currently be specified manually by a human designer. This may be a significant knowledge engineering challenge, since it is likely that the most appropriate abstraction hierarchy will change substantially between different domains of interest. The proposed framework, logic encoding, and provided implementation, seek to help to ease this task significantly.

The idea is that abstract planning should deal with only the most fundamental constraints of a problem. This allows the planner to form an abstract solution which represents some meaningful ordering over achieving the core stages of a plan that can solve the original problem (under its full constraints). The sub-goals should tell the planner not only how to achieve the final-goal, but how to "approach" the final-goal, by enabling actions that account for the fundamental constraints of the original problem. Whereby, the desire is that a plan would still have to take the same path to the final-goal, even if the sub-goals were not there. Importantly, a meaningful sequence of sub-goal stages are those that cannot hold simultaneously, and must be achieved in order to reach the final-goal. If the sub-goals can all hold simultaneously, this indicates that the planning constraints do not actually require that one sub-goal needs be achieved before the other, and the abstraction is ineffective. The development of general abstraction hierarchies that would give meaningful sub-goals for any given planning problem is however a question not within the scope of this thesis.

The following section describes the example and test domain used in this thesis, which is designed to contain the type of constraints that exist in interesting practical problems, and are appropriate for HCR planning but also exemplify its current limitations.

3.4 The Blocks World Plus Example and Test Domain

The Blocks World (BW) is a classic manipulation problem in robotics (Gupta and Dana S Nau, 1992). The original BW consists of a set of cuboid blocks, that are initially stacked or scattered on a table. Each block is given a unique number, and the objective is to stack the blocks into tower(s) on the table in (ascending or descending) numerical order. Whilst simple conceptually, BW problems are renowned for being challenging to solve, particularly for sub-goal based planners, mainly because they involve interacting constraints. In particular, a block cannot be moved if there is another block on top of it, and as a result, the order of moving/stacking the blocks is critical towards solving the problem in the fewest moves.

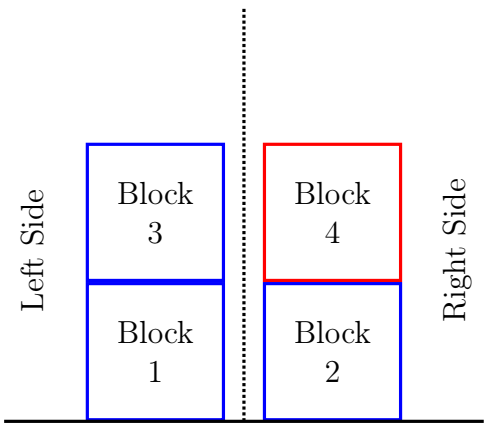
Path planning (navigation) and object transportation (logistics) are also typical problems in robotics (Helmert, 2003). This requires finding the shortest path between locations whilst carrying cargo. Whilst also simple, this is needed in almost all physical domains.

To demonstrate and experimentally test ASH, a novel extension of the original BW, called the Blocks World Plus (BWP) domain is proposed. The BWP is a combined manipulation and path planning/transportation problem, where the robot must collect the necessary blocks from around the domain, before solving the BW part of the problem. The complexity of the BW part is increased by colouring the blocks, and requiring them to be stacked in descending order towers of unique colour. The complexity of path planning is increased by adding doors between certain rooms, requiring the robot to have a free hand to open them.

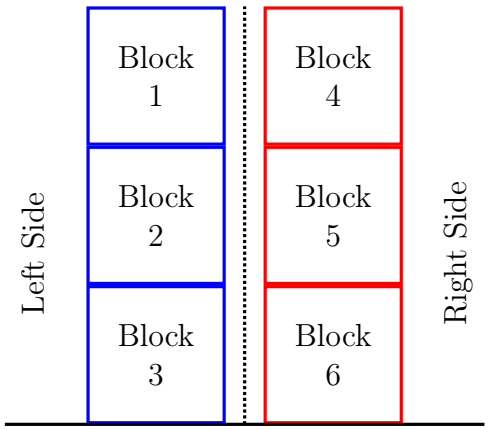
Figure 3.1c shows the structure of the BWP domain. It is composed of four rooms: a starting, store, and puzzle room, each divided into two cells, and a hallway divided into three cells. The dashed lines are connections, which may be blocked by a closed doorway. In cell 1 of the puzzle room there is a table, with two sides, to stack the blocks on. Four blocks start on the table in the configuration of Figure 3.1a and two red blocks start in cell 0 of the store room. The objective is to stack the blocks into the configuration of Figure 3.1b.

The initial configuration presents a classical reasoning challenge in planning, known as the "Sussman anomaly" (see Page 398 Russell and Norvig, 2016). This occurs when the final-goal is naively split into non-interleaved sub-goals, whereby achieving one sub-goal can cause another to be unachieved. For example, moving block 2 atop block 3, before moving block 3 from the top of block 1, will prevent block 1 from being moved. Such a predicament will require the blue tower to be completely rebuilt in order to finish solving the problem.

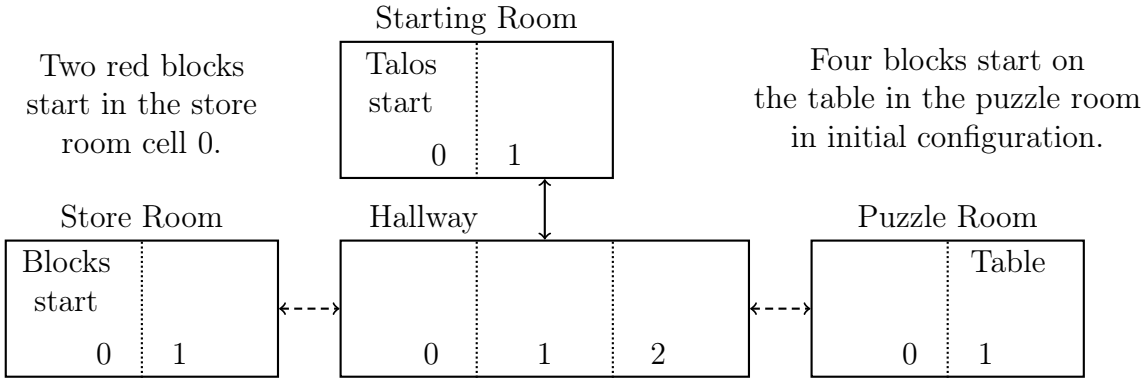
Talos, an anthropomorphic mobile robot, starts in cell 0 of the starting room. Talos has two prehensile manipulator arms, each of which is an assembly composed of an extensible limb and a hand that can grasp objects. To grasp an object, Talos must first extend a limb and align the hand attached to that limb, with the object. However, due to safety concerns, Talos is not allowed to move unless, he has first retracted all of his manipulator arms.



(a) Initial Configuration.



(b) Final-Goal Configuration.



(c) Static Domain Structure.

Figure 3.1: BWP Example Domain.

3.5 Hierarchical Planning Domains

A hierarchical planning domain is a declarative knowledge base, which describes the physical properties, laws, and structure of a dynamic domain over an abstraction hierarchy.

Definition 3.5.1 (Hierarchical Planning Domain) A hierarchical planning domain, defined over the range of abstraction levels[4] $l \in [1, tl]$, is defined by the tuple:

$$DD^{tl} = \langle H^{tl}, S^{tl}, R^{tl}, M^{tl-1} \rangle \tag{3.1}$$

Where each element is a finite set of facts or logic rules, the *class abstraction hierarchy* H^{tl} is a class typing system for entity constants, the *domain sorts* S^{tl} declare the *actions* available to the robot(s) and the *state variables* that represent the physical properties of the domain state, the *domain laws* R^{tl} declare dynamic and static axiomatic laws that define how the domain (and its abstractions) physically behave, the *state abstraction mappings* M^{tl-1} link different state representations between adjacent pairs of abstraction levels in the hierarchy.

Definition 3.5.2 (Domain Model) Every individual level in the abstraction hierarchy is considered to be a unique planning domain model, defined formally as follows[5]:

$$DM^{pl} = \begin{cases} \langle H^{tl}, S^{tl}(pl) \cup S^{tl}(pl+1), R^{tl}(pl) \cup R^{tl}(pl+1), M^{tl-1}(pl) \rangle & pl \in [1, tl) \\ \langle H^{tl}, S^{tl}(pl), R^{tl}(pl), \emptyset \rangle & pl = tl \end{cases} \tag{3.2}$$

Where if $pl = 1$ then the model is *ground*, otherwise it is *abstract*. If $pl = tl$ then the model is *top-level abstract*. For an *abstract model* at $pl > 1$, the model at level $pl-1$ is called its *original model*, this is the model from which the abstract was designed. The top-level abstract model differs from all others because it includes the definition of the domain sorts (which defines the state representation) at only one abstraction level (defining a classical representation),

[4]The notation $l \in [1, tl]$ and $l \in [1, tl)$ denotes $l \in ([1, tl] \cup \mathbb{N}) : tl \in \mathbb{N}$ and $l \in ([1, tl) \cup \mathbb{N}) : tl \in \mathbb{N}$.
[5]Where some $X^l(pl)$ is simply the sub-set of facts or rules from X^l related to the model at level pl.

whereas all others models include the sorts over a pair of two adjacent abstraction levels, the abstract and original, simultaneously (defining a refinement representation, explained in more detail below). All non-top-level original models must include the state abstraction mapping to its abstract model. Whilst this mapping is needed for refinement planning in the original model, it is intuitively only defined when creating the abstract model.

3.5.1 Actions and State

A planning domain must provide a representation of actions and state, and the constraints that underlie them, to allow reasoning about and generation of plans that solve arbitrary problems in that domain. The following formalises the vocabulary towards this end.

A planned action is one that a robot intends to execute at a given time step and abstraction level, which usually causes a physical state transition and progresses the robot towards some goal. A state transition is a change between two different states on successive time steps at the same abstraction level, caused by the direct and indirect effects of a set of planned actions. In sequential action planning, exactly one action must be planned on each state transition until the goal is achieved. Whereas, in concurrent action planning at least one action must be planned on each state transition. A state is a finite complete set of state literals that hold simultaneously at a given time step and abstraction level. A state literal represents the temporal assignment of a state variable from one of its argument tuples (from its domain) to one of its values (from its codomain). As described previously, a state variable is a function that describes some property or feature of the domain. Therefore, a state literal represents its value in a state, at a given time step and abstraction level.

There are three physical laws that underpin all others in maintaining a valid state as a plan progresses towards some given goal. These are defined in full later in Section 4.4.2. For now, it is sufficient to understand that these laws function as follows;

- the *continuity constraint* keeps the state complete and consistent, where all argument tuples for all state variables are assigned exactly one value at any given time step,
- the *law of inertia* prevents state variables changing value unless affected by the robot,
- and the *closed world assumption* states that anything not known to be true is false.

These laws therefore define what changes and what stays the same when an action is executed. This allows ASH to focus on concise descriptions of the effects of actions, whereby the result of an action's execution is specified only in terms of what changes, and everything else implicitly stays the same. It is therefore only necessary to specify the positive and direct effects of actions. Where, the direct effects are fluent state literals explicitly added to the state when an action is executed, whilst contradictory literals are implicitly deleted automatically[6].

3.5.2 Planning Domain Models

A planning domain model serves as a specification for the laws of a domain at a distinct level of abstraction. They each specify an ASP encoding which describes a unique discrete state transition system, for some ground or abstract model of the planning domain, which can be used to reason about planning problems in that model of the domain. Recall that the higher levels of the hierarchy contain increasingly abstract models, and the lower more concrete. The abstract models are obtained by removing or generalising problem constraints, or by applying action or state abstractions. This reduces abstract plan lengths or the description size of the abstract models of a planning problem. The models therefore create a specification for a hierarchy of state transitions systems, where higher levels are reductions on the lower, and whose states (at different levels) are linked directly by the state abstraction mappings. Where a reduced state transition system has; fewer states (as occurs in state abstractions), or more possible state transitions (as occurs with relaxed models), respective to the original.

[6]In planning paradigms, such as PDDL (Ghallab, C. Knoblock, et al., 1998) actions require an explicit "delete" list (the action's negative effects, literals removed from the state when an action is executed).

The state abstraction mapping used in HCR planning exists to allow representation of any action and state abstraction that modifies the actions and state variables used in the model, and not just those that remove them. HCR planning can in fact support any abstract model to which such a state abstraction mapping can be defined. In state abstractions where the abstract state variables are modified (such as in condensed models described below), the original level state variables are replaced by different abstract versions of the state variables. Since conformance refinement requires that refined plans achieve the same effects as the abstract plan they refine, and the effects of the abstract plan may be defined by abstract state variables which don't exist at the original level. To reason about the achieving those same effects therefore requires; maintaining the representation of, and providing the state abstraction mapping to, the abstract version of the state during plan refinement (linking different versions of the state variables[7]). This is why non-top-level models at levels $pl < tl$ (which can be used for refinement planning) must include the domain sorts (containing available actions and state variables) and domain laws for both the current refinement planning level pl and previous abstract level $pl + 1$. This allows the state to simultaneously be represented over a pair of adjacent abstraction levels called the *state representation levels*. To achieve this, any laws containing the special constant $sl \in \{pl, pl + 1\} : pl \in [1, tl)$, are expanded to automatically[8] obtain the versions for state representation levels[9] pl and $pl + 1$.

For a correct specification, the states of the transition systems described by the models must be fully linked, whereby the state abstraction mapping is exhaustive and deterministic. This is such that all abstract states must be mapped to from at least one original level state, and all original level states must map to exactly one abstract state. This is important, as it ensures all abstract states are accessible from some original level state, which is needed to be able to achieve the same effects of abstract actions during refinement planning[10].

[7]Note that even if a state variable does not change then both copies are persevered in the current theory.

[8]In past work using abstraction hierarchies in ASP, it was not possible to do this automatically because abstraction levels were not represented explicitly in predicates used to encode state literals and actions.

[9]Intuitively, if $pl = tl$, then this does not occur, because the model is solved by classical planning.

[10]This requirement does not ensure that all abstract states are reachable in the original model, as the original level state which links to the abstract might not be reachable in the original version of a problem.

The BWP domain has three abstraction levels; the ground model (level 1), and two abstract models: the first (level 2) is a condensed model of the ground, and the second (level 3) a relaxed model of the condensed. These models are described in Sections 3.5.2 and 3.5.2. The diagram in Figure 3.2 shows the structure of ASH loaded on the BWP domain. Notice that both the conformance constraints at both levels are defined by the condensed state representation, but the abstraction mappings are different. Both the condensed and relaxed (above the condensed) models use the condensed state representation, which is different to the more detailed state representation used in the ground model. Therefore, both models generate conformance constraints containing sub-goal stages that are defined by state literals in the condensed representation. The state abstraction mapping between these two abstract models is therefore direct and one-to-one. Whereas, the mapping between the ground and condensed model is many-to-one, from the large quantity of detailed states of the original model to the smaller quantity of more granular states of the condensed model.

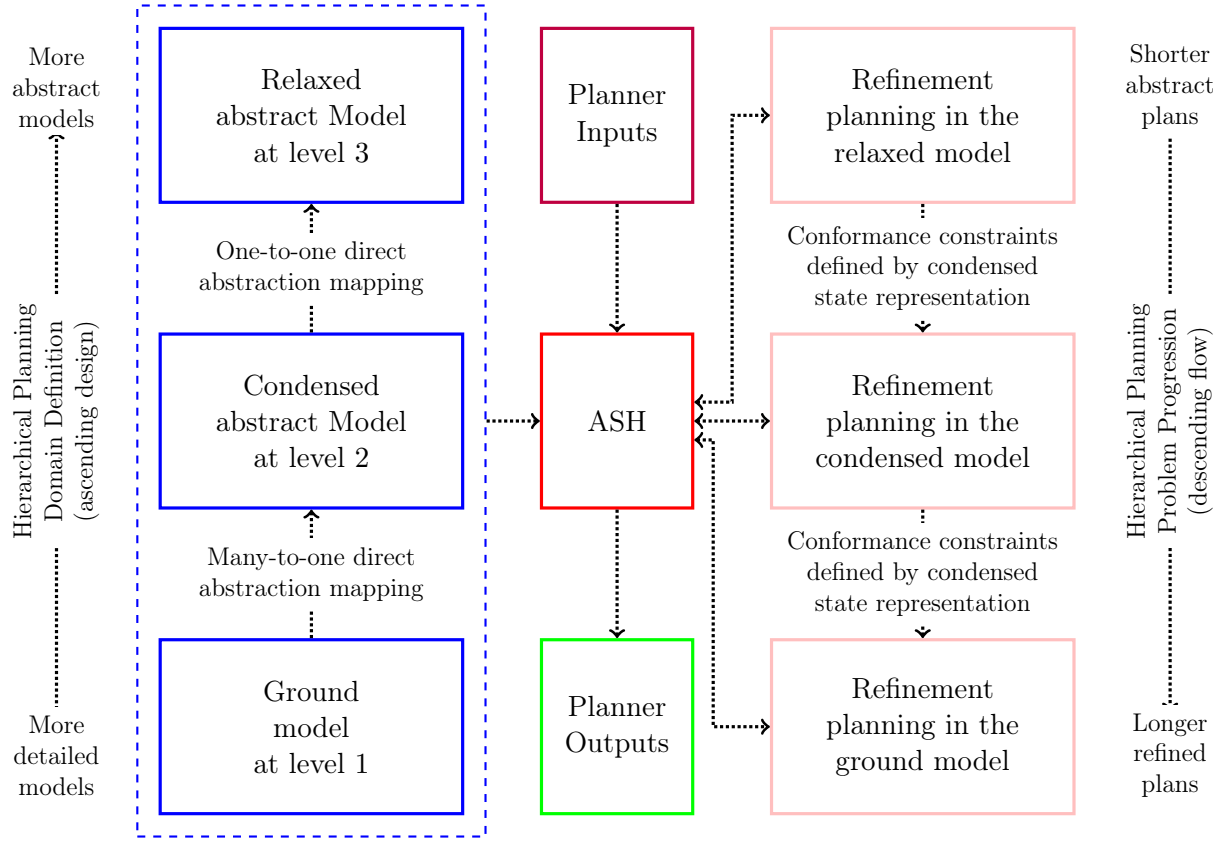


Figure 3.2: Structure of ASH for the BWP Domain.

The Ground Model

The ground model is the most detailed model of the domain and problem. It represents the actual problem to be solved and its solution is the plan sent to the robot for execution. It is therefore also the level at which the initial state and final-goal is given. When constructing an abstraction hierarchy, the ground model should always be designed first, and appropriate abstract models are then designed iteratively "from the ground up", in ascending order.

The ground model is not the same as the concrete model. The term concrete refers to the (usually continuous time and space) models used at execution-time by a robot's low-level actuator and sensor controllers. The ground model still maintains a level of abstraction over the concrete; ground-level actions should be the most abstract discrete instructions that can be understood and directly executed (in a general manner) by the robot's controllers. It must therefore be only as expressive as needed, to account for the nature of the domain and capabilities of the robot, and no more expressive. Irrelevant detail should always be avoided to minimise complexity and obtain a minimally sufficient description. For example, a grasp action is typically appropriate for the ground model, because the robot can be equipped with a specialised algorithm for trajectory control of its manipulator arms, and with dexterous grasping of arbitrary shaped objects[11]. In contrast, a single action that commands the robot to move an object from one location to another is unlikely to be detailed enough, this still requires some planning to achieve and therefore should be handled by the ground model.

Unfortunately, a ground model that is sufficiently expressive for most general purpose robotics applications (such as service robots) is usually too complex to be manageable for classical ASP planning. These applications necessitate generality, because they require handling a very wide range of different problems in a given domain. The abstract models, abstract planning, plan refinement, and problem division of HCR planning are therefore of importance, because they reduce problem and domain model complexity in a general way.

[11]The concrete grasping algorithm may need specific knowledge of the object's properties to grasp it, but this should come from the robot's visual apparatus or external knowledge bases, not the ground model.

Abstract Models

An abstract model is a simplification or generalisation of its original model and problem. The effects of actions planned in an abstract model only need to be achievable via actions from the original model and are not executed in reality. The initial state and final-goal of the abstract problem is automatically abstracted from that of the original.

When creating an abstract model, one should remove some of the details of the original, but preserve its most fundamental constraints. This can greatly reduce abstract planning complexity and time, by reducing the size of the problem description and/or length of plans, with respect to the original model. Abstract plans should have a structure that guides search and allows problem divisions to be made, to reduce planning time in the original model by more than it takes to find the abstract plan. Recall that a core concept is to break-down a problem's constraints by progressively removing them when ascending the hierarchy during its construction. These constraints are progressively reintroduced when descending the hierarchy during plan refinement, expanding plans and specialising actions to account for them. This makes it easier to account for the constraints than considering them all at once.

Finding an effective hierarchy does pose a significant knowledge engineering challenge. In particular, abstractions must be carefully balanced. If an abstract model removes too many constraints, its abstract plans might become of too poor quality, not guide search enough, or not allow meaningful divisions to made, and thus be too difficult to refine. If it removes too few, it might become too difficult to find its abstract plans. The most effective abstractions may also vary between domains. However, since abstract models simplify or generalise their originals, they are typically appropriate for a larger range of problems than the originals. Abstract plans are therefore also generalisations that can be refined in multiple ways to solve many different original problems. This makes abstract models general problem solving tools. However, it also makes it much less clear for; a) the designer to select the best abstract model to use, and b) the planner to select the best abstract plan to refine.

Relaxed Abstract Domain Models

Relaxed models are a simple yet often effective technique for creating abstractions of a planning domain. As such, they are very well established in the literature both for heuristic search and HR planning (see Sections 2.2 and 2.3.2). A relaxed model of a domain is obtained by applying simplifications which remove (i.e. relax) the enabling constraints of actions. The intention is to increase the number of applicable actions per state, by ignoring constraints that would usually forbid them, and resultantly increase the quantity of connected states. This makes it easier to find a state that satisfies the final-goal, because fewer actions (and shorter plans) are needed to achieve, or enable those that achieve, the final-goal.

This thesis uses a specific type of relaxed model which permits only the deletion of action preconditions, and everything else is preserved. Specifying a relaxed model simply requires flagging the sub-set of action preconditions to be deleted. The rest of the domain laws are automatically copied to the relaxed model. The state representation in the relaxed model is identical to the original, and the state abstraction mapping is trivial to define, direct, and one-to-one (since state variables map to themselves). Relaxed models of this form have an important property known as the upward solution property. This property states that any solution to the original model of a planning problem is also a solution to its relaxed model. This property holds because the original solution satisfies all constraints of the relaxed problem, plus those that were deleted to obtain it (J. Tenenberg, 1988).

The relaxed model used in the BWP domain removes all preconditions of locomotive actions, allowing Talos to move instantly between any two locations with one action (even if they are not directly connected), and removes a sub-set of the preconditions of manipulation actions, allowing Talos to grasp objects without having to first extend its manipulators arms. During plan refinement, a relaxed abstract locomotive action will for example expand to a sequence of locomotive actions, travelling along a path between the start and target locations of the relaxed action.

Condensed Abstract Domain Models

Condensed models apply an action and state abstraction to a planning domain and problem by removing or generalising actions and state variables. They are a natural and ubiquitous method for creating abstractions, as the concept comes from how humans describe the world at different levels of detail. The intention is to reduce both abstract problem description sizes and plan lengths, by abstracting away related sets of small and detailed entities, called descendants, and replacing them with larger abstract descriptor entities, called ancestors. These ancestors represent the combination of a set of their descendants, which are its parts or pieces (things it is composed of or made from), and which together give the ancestor certain properties as a whole. This reduces the number of entities needed to express a problem, resulting in fewer actions, state variables, and therefore fewer states and shorter plans.

Since the state representation in a condensed model is a reduced generalised form of the original model, the state abstraction mapping is many-to-one. This is because multiple original level state literals (containing descendent entities) may map to any given abstract state literal (containing ancestor entities), since ancestors can have multiple descendants. Whilst condensed models result in a more granular description, the abstraction mapping is still direct, due to the connection given by the ancestor-descendent relationship. This also minimises loss of information during plan refinement, because a condensed abstract plan is a generalisation of an original level plan, and its refinement is (one of) its specialisations.

Condensed models are specified through a class-based relation on the types of entities, in which a sub-set of original level class types (descendent classes) are deleted and replaced by abstract classes (ancestor classes)[12]. The ancestor classes gain an inheritance from all the super-classes of the descendent class types (called class inheritance overriding). These

[12]The class based specification of condensed models is conceptually similar to abstractions by analogical mappings proposed by Tenenberg J. Tenenberg, 1988. However, the key difference is that condensed models aim to reduce both the number of entities (i.e. constants) needed to express a problem through a class-based relation, whereas Tenenberg's abstractions are concerned specifically with reducing the number of class types (which he refers to as unary sort predicates) needed to express the entities within the problem.

class overrides therefore specify a logical coordination between two concrete classes, the ancestor and (one of) its descendent classes (some specific concepts), both of which can be subordinated to the same abstract class, the super-class of the descendent (some generic concept). More specifically, the two concrete classes are sub-classes, which are related to each other by a relationship such that instances of one class are (partially) composed of (a set of) instances of the other class, and both inherit from and are both more specific than the abstract super-class. This differs fundamentally from standard inheritance, which is solely a logical subordination, where one concept is strictly more specific than the other.

The condensed model used in the BWP specifies rooms as the union of their constituent cells, the puzzle table as the union of its two sides, and Talos' manipulator arm assemblies as the union of an upper limb and a grasper hand. Specifically, a manipulator arm assembly is an ancestor, which serves as an abstract descriptor for the combination of its descendant parts that compose it; an extensible limb, and a hand that can grasp objects. In the condensed model, the parts are abstracted away, and the arm is viewed as a single whole which has the abilities to extend and grasp objects, derived from its parts, simplifying the description. Similarly, a room is interpreted as an abstract descriptor for its constituent cells. In the condensed model, these cells are abstracted away, and Talos only considers locations relative to whole rooms. A condensed abstract navigation plan will therefore be a sequence of move actions between such rooms, and the plan's refinement will then expand into a "guided" sequence of move actions between the constituent cells of those same rooms[13]. This concept can significantly reduce the reasoning complexity of planning. This is because the complex search in the original model's problem space is guided by the requirement to achieve conformance with the abstract plan. In this case, the requirement to move through the same rooms as the abstract plan "focuses" search on only the relevant parts of the original problem space.

[13]The conjecture is that the existence of a condensed plan, implies the existence of a specialisation of that plan at the original level, obtained by expanding actions related to ancestors into actions related to their descendants. Vice versa, the existence of an original plan implies the existence of a generalisation at the abstract (a contraction of the original plan). However, this is not formally proven, and left to future work.

3.5.3 Refinement Trees

Refinement trees are structures that represent conformance refinements of plans. Studying refinement trees help us understand how the structure of an abstraction hierarchy affects plan refinement[14]. Refinement trees show how plans are expanded and actions specialised, to account for the planning constraints introduced at each level of hierarchical refinement[15].

The head of the tree is an abstract action set, which creates a sub-goal stage. Its branches lead to its children, a refined (expanded and specialised) sub-plan containing original actions. The children: achieve the same effects as those in the head (and its sub-goal stage), enable actions that do so, or are more detailed specialisations of the head actions.

Two example refinement trees for the BWP domain are presented below. The top-level green nodes are actions planned in a relaxed model, middle-level blue nodes the condensed, and bottom-level pink nodes the ground. The horizontal dashed arrows denote the creation and achievement of a sub-goal stage between the vertical action connections they cross.

Sub-Figure 3.3 is a navigation planning tree for the robot Talos (denoted by t) moving to puzzle room pr starting from starting room sr. In the relaxed model, mobility constraints are ignored, and Talos can move instantly to any location, therefore he reaches the puzzle room with one action, creating one sub-goal stage. In the condensed model, Talos may move only between connected rooms, hence (assuming the puzzle room door is open) he reaches the puzzle room with two actions, by moving through the hallway, creating two more sub-goal stages. In the ground model, the individual cells of the rooms are represented, and Talos must move between cells of these rooms; firstly reaching cell 1 of the hallway, then cell 2, and finally cell 0 of the puzzle room, achieving the middle-level sub-goal stages in sequence. In

[14]It is important to note that refinement trees are not constructed during planning, but can be extracted from a hierarchical plan after planning to visualise refinements, and are a purely representative structure.

[15]These are similar to the decomposition tress used in HTN planning, except instead of applying methods to decompose the abstract task in the head into sub-tasks and actions in a pre-defined manner, HCR planning infers a sub-goal stage from the abstract actions and the refinement planner then dynamically generates a new sub-plan which achieves that sub-goal stage under the stricter constraints of the original problem.

the abstract planning phase, the effect of each abstract action specifies some desired location L or way-point that must be reached as part of the solution to the problem. Thesse abstract actions are planned without consideration of how location L will be reached. The simplifying assumption of the abstraction therefore being that there exists some complete path that can reach L. The refinement of an abstract locomotive action requires finding a minimal path to L by planning an expanded sequence of possibly more specialised locomotive actions.

Sub-Figure 3.4 is a manipulation planning tree for Talos grasping a block b. In the relaxed model, Talos is freely able to grasp the block with his manipulator arm assembly *arm* with one action. In the condensed model, the arm must be extended prior to grasping the block, requiring an additional action to achieve the top-level sub-goal stage, and again creating two more. In the ground model, the arm is decomposed into its parts; a limb and hand. Extension of the manipulator arm is achieved by extending its descendant limb, and grasping of the block is achieved by grasping it with its descendent hand. However, grasping the object requires one additional alignment action to enable it. In this case, the effects of a grasp action in the abstract planning phase specify that a given object should be obtained for later use in the problem solution. A grasp action is planned without consideration of where exactly the object is, or what other objects may need to be moved to reach it. The assumption being that there is some way in which the object can be obtained. The refinement then requires planning a sequence of actions that enable the robot to grasp the object. These actions could involve moving other objects that obstruct Talos from reaching or grasping the desired object, or moving to an exact location from which the object is reachable.

Refinement trees therefore give us a peek at what is actually happening in conformance refinement, and show us how the structure of the abstraction hierarchy affects the refinements. They also hint at some of the main challenges in conformance refinement. In particular, it is difficult to predict the length of a sub-plan that an abstract action will refine into. This is true for multiple reasons. The constraints removed by an abstract model will either not affect all actions, or not affect all actions equally. Therefore, two different abstract

actions are likely to refine to different length sub-plans for the same abstract-original model pair. Further, problem constraints (particularly the preconditions of actions) can be conditional on the state, therefore a constraint that it relevant to a particular action in one state, may not be relevant to the same action in a different state. This means that the sub-plan length is highly dependent on the state at the end of the previous sub-plan, which is difficult to predict reliably, and it is only guaranteed to satisfy the previous sub-goal stage. This means an action that occurs more than once in an abstract plan, might refine to different length sub-plans. Finally, as aforementioned, there are usually multiple solutions to planning problems, and HCR planning does not currently have a mechanism for choosing the optimal solution. HCR is therefore intrinsically non-deterministic in the selection of which abstract plan to refine, as well as in the selection of which partial-plan to accept as the solution to partial-problems. The quality of the abstract and early partial-plans selected will impact the quality of refined plans and later partial-plans, and their sub-plan lengths.

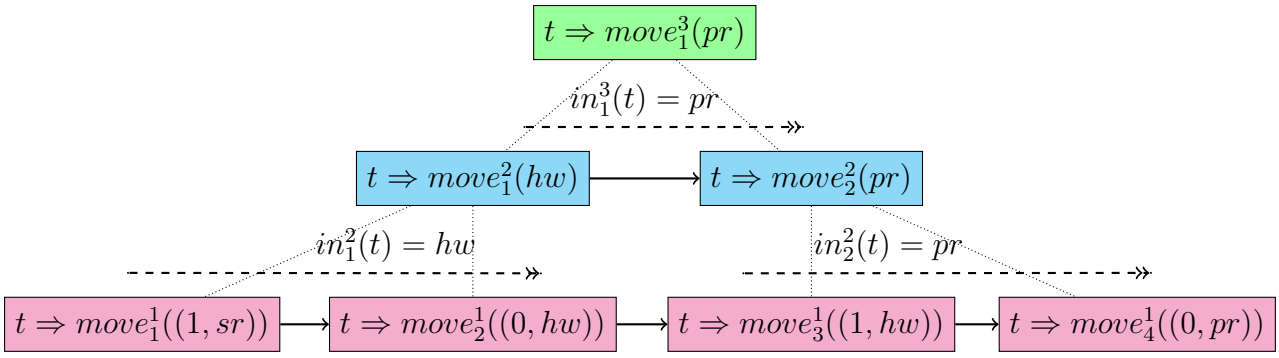


Figure 3.3: A Refinement Tree for a Locomotive action.

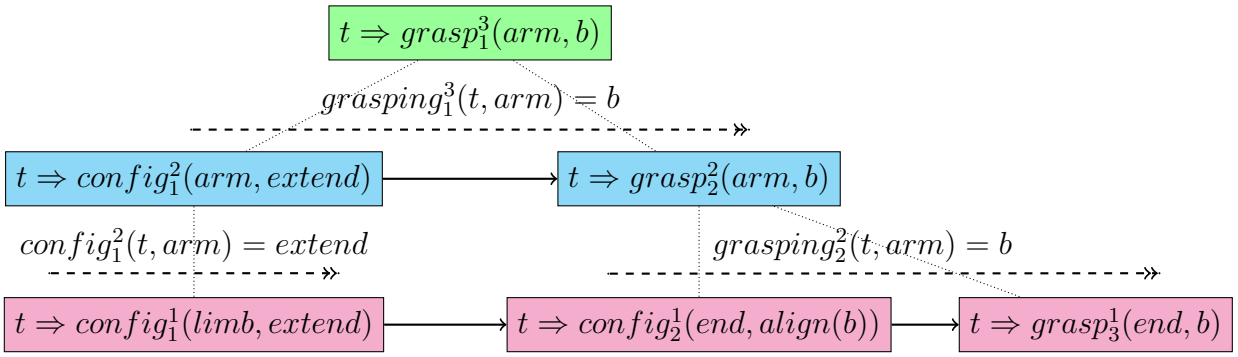


Figure 3.4: A Refinement Tree for a Manipulation Action.

3.5.4 Class Hierarchy

The class hierarchy is a formal structuring of class types used to identify all entities that exist in a planning domain. The class hierarchy used in HCR planning is unusual, as it is an inheritance hierarchy declared over an abstraction hierarchy. This creates three-dimensional directed graph, where the nodes are class types and the edges relations between classes. Furthermore, the class hierarchy contains intrinsic support for the specification of condensed abstract domain models, based on a novel class inheritance overriding mechanism.

Every real and tangible thing in the domain is an entity, and all entities have at least one class type. Class types are organised such that; a class that inherits from another is a sub-class, and the class that it inherits from is its super-class. Entities are an instance of their own base class type (the class type they are declared at) and all of the super-class types from which their base class inherits. For example, the class *object* is the set of all real things possessing a physical body and must have a location, and the class *location* is the set of locations objects can be located in. There are many sub-types of *object* and *location*. The sub-types of *object* can be *robot*, *door*, *block*. The sub-types of *location* can be; *building*, *room*, *cell*. Note that intangible concepts, such as *boolean* values can also be entities.

Definition 3.5.3 (Class Hierarchy) Formally, the class hierarchy is defined by the tuple:

$$H^{tl} = \langle \Psi, \Phi, K^{tl}, I, O \rangle \tag{3.3}$$

Where all elements are a finite set of facts: Ψ is the set of entity constants of the form ψ^{κ}, defining an entity whose name is ψ and whose most base class type is κ; Φ is the set of ancestry relations of the form $\phi(\psi_1, \psi_2)$ indicating that the entity ψ_1 is an ancestor of the descendent entity ψ_2; K^{tl} is the set of class types of the form κ^l, defining a class type whose name is κ declared at level $l \in [1, tl]$; I is the set of inheritance relations of the form $\iota(\kappa_1, \kappa_2)$, indicating that κ_1 is a super-class of κ_2; and O is the set of override relations of the form $o(\kappa_1, \kappa_2, \kappa_3)$.

Overrides are novel, they indicate that entities of class κ_1 are ancestors that have descendants of class κ_2, from which the ancestors derive an inheritance to the override class κ_3. For any override $o(\kappa_1, \kappa_2, \kappa_3) \in O$, there must also be inheritance relations $\{\iota(\kappa_3, \kappa_1), \iota(\kappa_3, \kappa_2)\} \subseteq I$, and the classes must be declared such that $\{\kappa_3^x, \kappa_1^y, \kappa_2^z\} \subseteq K^{tl}$ where $x \geq y > z$.

These elements are encoded in ASP by facts of the form:

An entity	$\epsilon(\psi^\kappa)$	is:	$entity(\kappa, \psi)$	(3.4)
An ancestry relation	$\epsilon(\phi(\psi_1, \psi_2))$	is:	$ancestry\_relation(\psi_1, \psi_2)$	(3.5)
A class type	$\epsilon(\kappa^l)$	is:	$class(l, \kappa)$	(3.6)
An inheritance relation	$\epsilon(\iota(\kappa_1, \kappa_2))$	is:	$super\_class(\kappa_1, \kappa_2)$	(3.7)
An override relation	$\epsilon(o(\kappa_1, \kappa_2, \kappa_3))$	is:	$override\_class(\kappa_1, \kappa_2, \kappa_3)$	(3.8)

More specifically, the three-dimensional graph representing the class hierarchy is defined as follows. Its nodes are labelled by class names. Its directed vertical arcs define the downwards propagation of classes across the abstraction hierarchy. Its directed horizontal arcs define inheritance relations between super- and sub-classes at the same level. Its undirected vertical arcs define override relations between inheritance arcs at different levels. Notice that multiple inheritance is allowed, whereby a sub-class can have many super-classes.

Class types exist at the level at which they are declared and all lower levels, such that for a class declared as κ^l, a class κ^m is added automatically to K for all $m \in [1, l]$. Therefore, any class that is not declared at the top-level of the hierarchy is said to have been deleted at all levels above its declared level. Any entity declared with a deleted class as their base class will also be deleted at those levels. A entity is then an instance of its base class, and all its super-classes, at all abstraction levels at or below the level at which the base class is declared. Unless, the entity is an ancestor and an override relation exists which breaks its inheritance from one of its super-classes and passes that inheritance to one of its descendant entities. This concept is non-trivial and is best described by the examples that follow.

The class hierarchy graph is represented declaratively in ASP via instance relation constraints. These are logic predicates used to constrain what entity constants replace entity variables occurring in non-ground ASP rules. The desire and need for this is described in Section 3.5.5. An instance relation constraint can be any of the following:

- A type constraint $insta\_of(l, \kappa, \psi)$ which says that entity ψ is an instance of class κ at abstraction level l. The short-hand $isa^l(\psi, \kappa)$ will be used to denote this.
- A ancestry constraint $child\_of(l, \psi_1, \psi_2)$ which says that entity ψ_2 is the child (direct descendant) of ψ_1, and $desce\_of(l, \psi_1, \psi_2)$ which says that entity ψ_2 is any (possibly indirect) descendant (child, or child of child, etc) of ψ_1, at abstraction level l. The short-hands $chi^l(\psi_1, \psi_2)$ and $des^l(\psi_1, \psi_2)$ respectively will be used to denote this.
- A sibling constraint[16] $siblings(l, \psi_1, \psi_2)$ which says that entity ψ_1 and ψ_2 are siblings, whereby they are both the child of some same ancestor entity, at abstraction level l. The short-hand $sib^l(\psi_1, \psi_2)$ will be used to denote this.

The generation of these constraints over different abstractions is fully automatic. However, unlike abstractions of the state space, which are defined in an upwards manner (i.e. abstract spaces are defined in terms of ground spaces), the class hierarchies are generated in a downwards manner. The generation of the hierarchy can be thought of as the discovery of class types from K and their respective entity instances from Ψ as the details of the ground level planning problem are progressively uncovered as the abstraction level is incrementally reduced. The generation of the instance relation constraints is defined as follows:

- All entities are instances of their own class at the same abstraction level:

$$isa^l(\kappa, \psi) \Leftarrow \psi^\kappa, \kappa^l$$

- And are instances of all their super-classes at the same abstraction level:

$$isa^l(\kappa_1, \psi) \Leftarrow isa^m(\kappa_2, \psi), \iota(\kappa_1, \kappa_2), \psi^\kappa, \kappa^l$$

[16]The siblings relation is used when two related descendants of an ancestor are in a problem constraint.

- As well as the next lower level if its type is not overridden by one of its descendants:

$$isa^{l-1}(\kappa, \psi) \Leftarrow isa^{l}(\kappa, \psi), \neg overridden^{l-1}(\kappa, \psi)$$

- An override type of an ancestor entity is overridden at a given level if one of its descendants inherits from that override type at that level:

$$overridden^{l-1}(\kappa_3, \psi_1) \Leftarrow isa^{l}(\kappa_1, \psi_1), isa^{l-1}(\kappa_2, \psi_2), isa^{l}(\kappa_3, \psi_1), isa^{l-1}(\kappa_3, \psi_2),$$
$$des^{l-1}(\psi_1, \psi_2), o(\kappa_1, \kappa_2, \kappa_3)$$

- An entity is a child of another at all levels they exist if there is an ancestry relation between them:

$$chi^{l}(\psi_1, \psi_2) \Leftarrow \phi(\psi_1, \psi_2), isa^{l}(\kappa_1, \psi_1), isa^{l}(\kappa_2, \psi_2)$$

- An entity is a descendant of another if either: the entity is the child of the other entity; or by transitivity if the entity is a descendant of some different entity, which is itself a descendant of the other entity:

$$des^{l}(\psi_1, \psi_2) \Leftarrow chi^{l}(\psi_1, \psi_2)$$
$$des^{l}(\psi_1, \psi_3) \Leftarrow des^{l}(\psi_1, \psi_2), des^{l}(\psi_2, \psi_3)$$

- Entities are siblings if they are both the child of the same entity at the same level:

$$sib^{l}(\psi_2, \psi_3) \Leftarrow des^{l}(\psi_1, \psi_2), des^{l}(\psi_1, \psi_3), \psi_1 \neq \psi_2$$
$$sib^{l}(\psi_2, \psi_1) \Leftarrow sib^{l}(\psi_1, \psi_2)$$

In a condensed model, the ancestor classes inherit from the override classes because the descendent classes are no longer present, having been deleted to create the abstract model. This is such that the ancestor classes can act as the abstract descriptions for the combination of the original descendent classes. In the original model, instances of the ancestor class yield the override classes back to their descendants from which they originally derived them. The ancestor classes remain present in the class hierarchy, and can be referred to explicitly, but are no longer linked to the override classes by an inheritance relation. Descendent classes affected by the model can always be safely deleted from the hierarchy, if all the super-

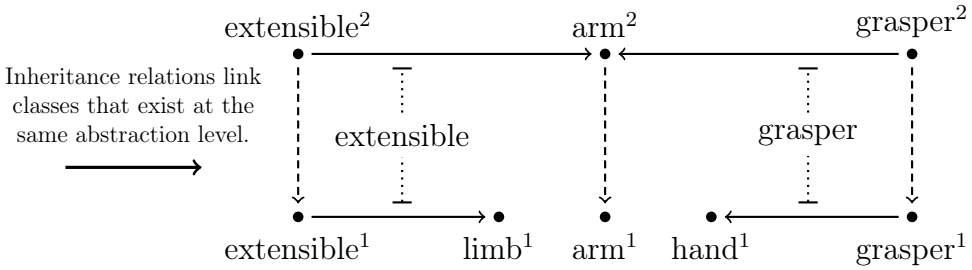
classes from which they inherit are overridden, and therefore the inheritance relations from all their super-classes are transferred to their ancestor classes. All system rules containing the override classes are generalised to entities of the ancestor classes, and none are removed. A state variable or action whose domain or range is defined by an override class is called a condenser variable or action respectively. If the domain is defined by an override class, then fewer literals are needed to represent a state. If the codomain is reduced, fewer possible literals will exist overall, and therefore fewer possible states exist. If the domain of an action is defined by an override class, fewer possible actions will exist, leading to lower branching factors. If some of the super-classes of a descendent class are not overridden, then the descendent class can still be deleted. However, any rules containing that descendent class as a type constraint will be deleted from the domain laws. A descendent class can have only one direct ancestor class, but it can have multiple indirect ancestor classes, if its direct ancestor also has ancestors. For example, room is the direct ancestor of cell (through the override class location), and building may be the direct ancestor of room, and thus by transitivity, building is the indirect ancestor of cell (all of which are locations). An entity of a descendent type must have exactly one ancestor entity for each of its override types[17]. An entity of an ancestor type must have at least one descendant entity for each of its descendent types[18].

Part of the class hierarchy graph for the BWP is given in Figure 3.5. This describes the composition of: (3.5a) Talos' manipulator arms as an assembly of an extensible upper limb and a grasper hand; (3.5b) rooms as sets of cells; and (3.5c) the puzzle table having two sides. These graphs formally represent the concepts described in Section 3.5.2. Consider Sub-Figure 3.5a as an example: manipulator arms have extensible upper limbs, both can be considered capable of extending and retracting themselves, the arm only as a result of having the extensible upper limb. There is therefore a logical coordination between the arm as a

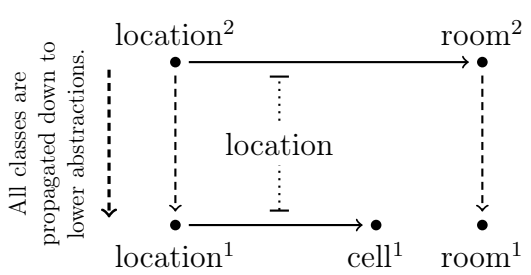
[17]If no ancestors existed, then there will be no version of the descendent entity at the abstract level to yield its inheritance to the override type to (since class types are never propagated up the hierarchy).

[18]If no descendants existed, the inheritance of the override class to the ancestor class does not get overridden, and the ancestor entity continues to be the descriptor used at the original level. This would not make sense for the semantics, as abstraction hierarchies are created in an upwards manner and therefore ancestor entities should only be added as abstract descriptors for original level entities that already exist.

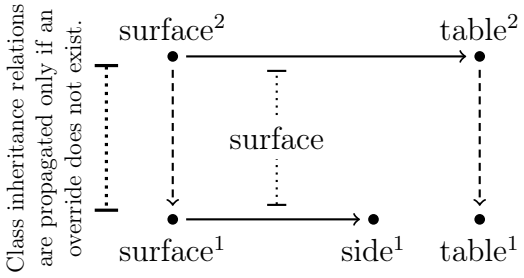
whole and its limb, both of which subordinate from the abstract concept of being extensible. Similarly, manipulator arms have attached grasper hands, both can be considered capable of grasping objects, the arm only as a result of having the hand. More formally; level 2 is the condensed model, where the ancestor class *arm* inherits from its two override classes, *extensible* and *grasper*, as indicated by the solid horizontal arcs because its descendant classes have been deleted. At this level, all arms are therefore considered to be extensible and capable of grasping objects, irregardless of their components. Level 1 is the ground model, the classes from the previous level are propagated down, and the two descendant classes *limb* and *hand* are introduced. The dotted vertical arcs indicate the overrides $o(arm, limb, extensible)$ and $o(arm, hand, grasper)$ between the ancestor and descendant classes. These break the inheritance relation of the override classes to the ancestor class *arm* at level 2, and yields them back to its descendant classes, such that *limb* and *hand* now inherit from *extensible* and *grasper* respectively at level 1. Therefore, at this level, given that we now represent an arm's components explicitly, the limb specifically is considered extensible and the hand is capable of grasping objects, rather than the arm as a whole having these properties.



(a) Sub-Graph for Talos' Manipulator Arm Composition.



(b) Sub-Graph for Location Composition.



(c) Sub-Graph for Surface Composition.

Figure 3.5: Part of the Class Hierarchy Graph for the BWP containing Override Relations.

3.5.5 Domain Sorts

The domain sorts declare the robots' available actions and the domain's state variables, for each domain model in an abstraction hierarchy. Where state variables can be fluent (dynamic and whose values can change in time) or static (fixed and cannot change). The domain laws then use these actions and state variables to specify the domain's dynamic behaviour.

To achieve an intuitive and compact encoding, the sorts are specified using functions of typed variables. Specifically, a sort is a possibly multivariate (n arity) named function of several class typed entity variables (called the parameter variables) to either a class typed entity variable or simply a boolean (called the value variable). Where the parameter and value types define the domain and codomain of the function respectively. For actions and statics, the value variable must be boolean[19]. Where the boolean value of an action simply defines whether it was planned or not. The general forms of these functions are as follows:

$$action\_name : parameter\_type_0 \times ... \; parameter\_type_n \rightarrow boolean \tag{3.9}$$

$$fluent\_name : parameter\_type_0 \times ... \; parameter\_type_n \rightarrow value\_type \tag{3.10}$$

$$static\_name : parameter\_type_0 \times ... \; parameter\_type_n \rightarrow boolean \tag{3.11}$$

For example, an action defining a robot's ability to move (by locomotion) and a state variable defining the location of objects can be denoted respectively as follows:

$$move : robot, location \rightarrow boolean \tag{3.12}$$

$$in : object \rightarrow location \tag{3.13}$$

The literal meaning of these declares that; a robot can move to any location, and all objects must have a location. For the ASP solver to reason with these, they must be grounded into atoms by replacing the parameter and value types with constants (this is described later).

[19]Statics can only take boolean values because the value of statics cannot change. Therefore, non-boolean statics cannot make the representation more compact, as negative literals are not represented explicitly.

Definition 3.5.4 (Domain Sorts) Formally, the domain sorts are defined by the tuple:

$$S^{tl} = \langle A^{tl}, F^{tl}, C^{tl} \rangle \tag{3.14}$$

Where all elements are finite non-empty sets, A^{tl} is the set of *actions* of the form $\kappa_r \Rightarrow a^l(\bar{\kappa_n})$, F^{tl} is the set of *fluent state variables* of the form $f^l(\bar{\kappa_n}) = \kappa_v$, and C^{tl} is the set of *static state variables* of the form $c^l(\bar{\kappa_n})$. Where κ_r is the robot type (the type of robot that can execute this action[20]), a, f, and c are unique function names (strings over a fixed alphabet), $\bar{\kappa}$ a vector $(\kappa_0, \kappa_i, ... \kappa_n)$ of parameter types of length n (function domain of arity n), κ_v a value type (function codomain), and $l \in [1, tl]$ the abstraction level the sort is defined at.

Actions are split into sequential and concurrent actions. Only one sequential action can be planned on each time step. Whereas an arbitrarily large set of concurrent actions can be planned on each time step, if they meet certain concurrency constraints. These constraints enforce that a set of concurrently planned actions must be executable by the robot (if ground) or their effects achieved during refinement planning (if abstract), either simultaneously or sequentially in any order. Fluents are split into inertial and defined fluents (as in Gelfond and Inclezan, 2013). Inertial fluents can be affected directly by actions, and obey the law of inertia, whereby their value cannot change unless affected (directly or indirectly) by actions. Defined fluents cannot be affect directly by actions, and are instead defined entirely in terms of other fluents, but therefore can be changed indirectly by actions.

These sorts must be encoded into ASP as non-ground logic rules of the forms:

$$\text{An action} \quad \epsilon(\kappa_r \Rightarrow a^l(\bar{\kappa_n})) \text{ is:} \quad action(pl, P, R, a(\bar{T_n})) \leftarrow body(a)^{pl} : P \in \{se, co\} \tag{3.15}$$

$$\text{A fluent} \quad \epsilon(f^l(\bar{\kappa_n}) = \kappa_v) \text{ is:} \quad fluent(sl, B, f(\bar{T_n}), \Upsilon) \leftarrow body(f)^{sl} : B \in \{in, de\} \tag{3.16}$$

$$\text{A static} \quad \epsilon(c^l(\bar{\kappa_n})) \text{ is:} \quad static(hl, c(\bar{T_n})) \leftarrow body(c)^{hl} \tag{3.17}$$

[20]The robot type parameter is separated out for generality and to build-in support future work that will extend to multiple heterogeneous robot problems (problems were the robots have differing capabilities).

Where $\bar{T_n}$ is a vector of entity variables $(T_0, T_i, ...T_n)$, *se* and *co* denote a sequentially and concurrently executable action respectively, and *in* and *de* denote an inertial and defined fluent respectively[21]. All entity variables in the head of a sort declaration must be in a type constraint in its body to define its the domain and codomain, such that:

$$body(a)^l = \{inst\_of(l, \kappa_r, R), inst\_of(l, \kappa_1, T_1), ... \; inst\_of(l, \kappa_n, T_n)\} \tag{3.18}$$

$$body(f)^l = \{inst\_of(l, \kappa_1, T_1), ... \; inst\_of(l, \kappa_n, T_n), inst\_of(l, \kappa_v, \Upsilon)\} \tag{3.19}$$

$$body(c)^l = \{inst\_of(l, \kappa_1, T_1), ... \; inst\_of(l, \kappa_n, T_n)\} \tag{3.20}$$

Where $\kappa_i \in \bar{\kappa_n}$ and $T_i \in \bar{T_n}$. The predicates in the head of the sort declarations are sort constraints, used in domain laws to define the sorts used in them. The non-ground action and state variable declarations will expand during program grounding to a set of variable-free atoms of these predicates, defining all available actions that can be planned and possible state literals that can hold on some time step at a given abstraction level[22]. These are obtained automatically by replacing all variables R, T_i, and Υ, with entity constants of the same class type. The argument set for such variables is all entities of the parameter type, and the domain of a function is the set of all n-tuples combining those arguments.

For example, given the entities; $\{talos\}$ of type *robot* and $\{kitchen, bathroom\}$ of type *location*, then the action declaration in 3.12 would become $action(l, talos, move(kitchen))$ and $action(l, talos, move(bathroom))$ defining all locomotive actions Talos can plan ($talos \Rightarrow move^l(kitchen)$ and $talos \Rightarrow move^l(bathroom)$), and the fluent declaration in 3.13 would become $fluent(l, ine, in(talos), kitchen)$ and $fluent(l, ine, in(talos), bathroom)$ defining all of Talos' possible locations ($in^l(talos) = kitchen$ and $in^l(talos) = bathroom$)).

These ASP sort constraints are overly verbose to write in-text. Therefore, the following short-hand notation is used to define the type of variables occurring in a sort:

[21]The type of actions and fluents will not be explicitly denoted in the short-hand to reduce clutter.

[22]Note that these are all the *possibilities* for actions and state literals, not those that are actually planned or hold in a state, therefore they are grounded only once and don't include a step parameter.

$$R\$\kappa_r \Rightarrow action\_name^l(T_1\$\kappa_1, ... T_n\$\kappa_n) \tag{3.21}$$

$$fluent\_name^l(T_1\$\kappa_1, ... T_n\$\kappa_n) = \Upsilon\$\kappa_\Upsilon \tag{3.22}$$

$$static\_name^l(T_1\$\kappa_1, ... T_n\$\kappa_n) \tag{3.23}$$

Where the capital letters are (as usual) variables for non-ground rules, and the type name to the right of the dollar sign \$ is the type constraint imposed on that variable.

Equations 3.24 and 3.27 are typed declarations for the move location action $R\$robot \Rightarrow in^l(L\$location)$ and the in location fluent $in^l(O\$object) = L\$location$ respectively. The type of the single parameter of the *move* action and the value of the *in* fluent are *location*. Since *location* is an override class defined by relation $o(room, cell, location)$ (described in Figure 3.5b), these are called a *condenser action and fluent*. During program grounding, the *location* typed variable L, is replaced by entity constants of the sub-class which inherits from *location* in current model; that is *room* in the condensed model and *cell* in the original model. This automatically obtains a correctly type constrained version of this state variable for the two models from one declaration. These are shown in Equations 3.25 and 3.26 for the *move* action, and Equations 3.28 and 3.29 for the *in* fluent. Where the class to the right of the @ notation is the override class, and to the left the ancestor or descendent class that takes the inheritance from the override class in the current model. If the relevant override class is a sub-class of the type constraint declared on the parameter or value variable of the sort then the $\leftarrow$ notation is used to denote this. These semantics provide a clear and precise class based mapping between the actions and state variables in the different models.

The action and state abstraction provided by condensed models is powerful in reducing the problem description size. Consider the $in^l(O\$object) = L\$location$ fluent. It condenses the state representation, combining multiple cells from the ground model into fewer rooms in the condensed. In the BWP, this reduces the number of state literals for the possible object locations from 9^7 in the ground model to only 4^7 in the condensed, a 99.7% reduction.

$$R\$robot \Rightarrow move^{pl}(L\$location) \tag{3.24}$$

$$R\$robot \Rightarrow move^{2}(L\$room@location) \tag{3.25}$$

$$R\$robot \Rightarrow move^{1}(L\$cell@location) \tag{3.26}$$

$$in^{sl}(O\$object) = L\$location \tag{3.27}$$

$$in^{2}(O\$object) = L\$room@location \tag{3.28}$$

$$in^{1}(O\$object) = L\$cell@location \tag{3.29}$$

The following shows further condenser actions and fluents describing how the composition of Talos' manipulator arms affect the grasp and component configuration actions in the condensed and ground models (based on the override relations from Figure 3.5a):

$$R\$armed\_robot \Rightarrow grasp^{pl}(G\$grasper, O\$object) \tag{3.30}$$

$$R\$armed\_robot \Rightarrow grasp^{2}(M\$arm@grasper, O\$object) \tag{3.31}$$

$$R\$armed\_robot \Rightarrow grasp^{1}(E\$hand@grasper, O\$object) \tag{3.32}$$

$$R\$armed\_robot \Rightarrow configure^{pl}(C\$component, S\$state) \tag{3.33}$$

$$R\$armed\_robot \Rightarrow configure^{2}(M\$arm@extensible \leftarrow component, S\$state) \tag{3.34}$$

$$R\$armed\_robot \Rightarrow configure^{1}(L\$limb@extensible \leftarrow component, S\$state) \tag{3.35}$$

Any or all of the terms and the value for a sort can be constrained by an override type. If there is more than one override then not all combinations of the ancestor and descendent types are necessary. For example, the action $\$robot \Rightarrow put^{pl}(\$grasper, \$graspable, \$surface)$ where both *grasper* and *surface* are declared override types, only the fully condensed version $\$robot \Rightarrow put^{2}(\$arm@grasper, \$graspable, \$table@surface)$ and the completely original version $\$robot \Rightarrow put^{1}(\$hand@grasper, \$graspable, \$side@surface)$ are represented. All the other versions $\$robot \Rightarrow put^{l}(\$arm@grasper, \$graspable, \$side@surface)$ and $\$robot \Rightarrow put^{l}(\$hand@grasper, \$graspable, \$table@surface)$ are not needed and are excluded[23].

[23]Note that for state variables, the excluded versions can be inferred by taking the Cartesian product of the arguments of the pair of state literals for the condensed and original version of the state variable.

3.5.6 Representation of Planned Actions and States in ASP

In order to generate plans, and represent and reason about actions and states, predicates must be added that parameterise actions and state literals by abstraction level and time.

- Let $f_i^l(\bar{t_n}) = v$ and $f_i^l(\bar{t_n}) \neq v$ be positive and negative fluent state literals, and $c^l(\bar{t_n})$ and $\neg c^l(\bar{t_n})$ be positive and negative static state literals, which hold at abstraction level l and discrete time step i, and whose arguments are $(t_i, ... t_n)$ and values are v.
  - In ASP fluent literals are encoded by the predicate $holds(L, F, V, I)$:

$$\epsilon(f_i^l(\bar{t_n}) = v) = holds(l, f(\bar{t_n}), v, i)$$

$$\epsilon(f_i^l(\bar{t_n}) \neq v) = not\ holds(l, f(\bar{t_n}), v, i)$$

  - In ASP static literals are encoded by the predicate $is(L, C)$:

$$\epsilon(c^l(\bar{t_n})) = is(l, c(\bar{t_n}))$$

$$\epsilon(\neg c^l(\bar{t_n})) = not\ is(l, c(\bar{t_n}))$$

- Let $\sigma_i^l = \{f_i^l(\bar{t_o}) = v, ...\ c^l(\bar{t_o}), ...\}$ be a complete state, a finite set of positive fluent and static state literals at the same abstraction level l and discrete time step i.
  - The ASP encoding is a finite set of atoms:

$$\epsilon(\sigma_i^l) = \{\epsilon(\nu) \mid \nu \in \sigma_i^l\} : \nu \in \{f_i^l(\bar{t_n}) = v, c^l(\bar{t_n})\}$$

- Let $\varsigma_i^l = \{f_i^l(\bar{t_o}) = v, ...\}$ be a partial fluent state, a finite subset of fluent state literals from a complete state, which is consistent such that it contains no contradictions[24].
  - The ASP encoding is a finite set of atoms:

$$\epsilon(\sigma_i^l) = \{\epsilon(\nu_i^l(\bar{t_o}) = v) \mid (\nu_i^l(\bar{t_o}) = v) \in \sigma_i^l\}$$

- Let $\delta_i^l = \langle \sigma_i^l, \sigma_i^{l+1} \rangle$ be a state pair, a tuple of states at abstraction levels l and $l + 1$.
  - The ASP encoding is a finite set of atoms:

$$\epsilon(\delta_i^l) = \epsilon(\sigma_i^l) \cup \epsilon(\sigma_i^{l+1})$$

[24] A contradiction is a pair of fluent state literals with the same argument tuples but different values.

- Let $r \Rightarrow a_i^l(\bar{t_n})$ be an action planned by the robot r to occur or be executed at abstraction level l and discrete time step i whose arguments are $(t_i, ... t_n)$.
  - In ASP planned actions are encoded by the predicate $occurs(L, R, A, I)$:

$$\epsilon(r \Rightarrow a_i^l(\bar{t_n})) = occurs(l, r, a(\bar{t_n}), i)$$

- Let $\alpha_i^l = \{r \Rightarrow a_i^l(\bar{t}), ...\}$ be a concurrently planned action set, a finite set of planned actions at the same abstraction level l and discrete time step i.
  - The ASP encoding is a finite set of atoms:

$$\epsilon(\alpha_i^l) = \{\epsilon(r \Rightarrow a_i^l(\bar{t_n})) \mid (r \Rightarrow a_i^l(\bar{t_n})) \in \alpha_i^l\}$$

- Let $\tau_i^l = \langle \delta_i^l, \alpha_{i+1}^l, \delta_{i+1}^l, \varsigma_{i+1}^l \rangle$ be the state transition occurring from the execution of action set α_i^l in state $\sigma_i^l[\delta_i^l]$ where ς_{i+1}^l is the direct effects of the action set on $\sigma_{i+1}^l[\delta_{i+1}^l]$.
  - The ASP encoding is a finite set of atoms (note that ς_{i+1}^l is a subset of $\sigma_{i+1}^l[\delta_{i+1}^l]$):

$$\epsilon(\tau_i^l) = \epsilon(\delta_i^l) \cup \epsilon(\alpha_{i+1}^l) \cup \epsilon(\delta_{i+1}^l)$$

- Let $\pi_{j,k}^l = \{\tau_i^l \mid i \in [j, k]\}$ be a monolevel plan, a finite contiguous sequence of state transitions at the same abstraction level l and over the discrete time steps $j \leq i \leq k$.
  - The ASP encoding is a finite set of atoms:

$$\epsilon(\pi_{j,k}^l) = \bigcup_{i=j}^{k} \epsilon(\tau_i^l)$$

- Let $\lambda_\varrho^{l+1} = \{f_\varrho^l(\bar{t}) = \upsilon, ...\}$ be a sub-goal stage at sequence index[25] ϱ, a finite set of positive fluent literals at abstraction level $l + 1$, equivalent to the partial-state ς_ϱ^l.
  - In ASP a sub-goal is encoded by the predicate $sub\_goal(L, F, V, S)$:

$$\epsilon(\lambda_\varrho^{l+1}) = \{sub\_goal(l, f(\bar{t}), \upsilon, \varrho) \mid (f_\varrho^l(\bar{t}) = \upsilon) \in \lambda_\varrho^l\}$$

- Let $\Lambda_{\varphi,\vartheta}^{l+1} = \{\lambda_\varrho^{l+1} \mid \varrho \in [\varphi, \vartheta]\}$ be a conformance constraint, a finite sequence of sub-goal stages at abstraction level $l + 1$ and between sequence indices $\varphi \leq \varrho < \vartheta$.
  - The ASP encoding is a finite set of atoms:

$$\epsilon(\Lambda_{\varphi,\vartheta}^{l+1}) = \bigcup_{\varrho=\varphi}^{\vartheta} \epsilon(\lambda_\varrho^{l+1}) \mid \lambda_\varrho^{l+1} \in \Lambda_{\varphi,\vartheta}^{l+1}$$

[25]Note that the sequence index of a sub-goal is the time step it was planned at in its abstract plan.

3.5.7 Domain Laws

The domain laws contain the core descriptive knowledge base of the robot(s). It is a set of logical axioms that encode the fundamental physical laws of the dynamic domain in which the robot(s) operate. A robot can use these to intelligently find plans and in a general way, by simulating how the effects of its actions can change the state towards any given goal.

The type of axiomatic knowledge the system laws can contain is intuitive and easily provided by humans. This can include (but is not limited to[26]) the following concepts:

1. *Causal Relations*: These relate to cause and effect, including the direct effects of actions and casual connections between physical properties of the domain (where changing one thing causes another thing to change as well). For example; "moving causes a location change" and "moving whilst holding an object also moves the object".

2. *Non-Causal Relations*: These are non-casual connections between physical properties of the domain, such as a relation that a set of things must hold simultaneously (but one thing doesn't cause the others), and constraints on what constitutes a valid state of the world. For example; "an object that is on a table is above the table" and "it is not possible for two objects to occupy the same space simultaneously".

3. *Enabling Constraints*: These are constraints on the execution of actions, including conditions that must hold prior to an action being executed (called preconditions) or constraints that make it impossible for the effects of an action to result in a valid state. The classic example is of course; "you can not put a square peg in a round hole".

The ability to induce these types of axioms and generalise from them to solve arbitrarily complex and unseen problems, is core to human intelligence. The intuition is then, that a robot equipped with these axioms, and some mechanism for reasoning with them, captures an essence of the high-level reasoning and problem solving skills that humans possess.

[26] ASP can represent any axiomatic knowledge that can be expressed in formal logic, few restrictions exist.

To keep the syntax and semantics of domain laws clear and concise, a sub-set of the action language BC (a high-level notation for answer set programs) with a small notational extension is employed to represent them. The general forms of the laws are as follows:

- **Action Effects:**

$$\langle action : pl, i \rangle \textbf{ causes } \langle effect : pl, i \rangle \textbf{ if } \langle conditions : pl, i-1 \rangle$$

- **Action Preconditions:**

$$\langle action : pl, i \rangle \textbf{ requires } \langle precondition : pl, i-1 \rangle \textbf{ if } \langle conditions : pl, i-1 \rangle$$

- **State Variable Relations:**

$$\langle result : sl, i \rangle \textbf{ if } \langle conditions : sl, i \rangle$$

- **State Variable Constraints:**

$$\textbf{impossible } \langle conditions : sl, i \rangle$$

Where the notation l, i after the colon, define are the abstraction level l and time step i, of the planned actions and state literals allowed in the given clause, *action* is any planned action, $effect$ is a positive inertial fluent literal, *precondition* is any positive or negative fluent literal, *result* is any state literal, and *condition* is a conjunction of state literals. Effects and preconditions are included only for the planning level pl. Whereas state relations and constraints included for both state representation levels sl, to ensure that an invalid state relative to the abstract level is not reached during refinement planning. The body of an effect and precondition, and the **requires** clause of a precondition, is conditional of the previous state at step $i-1$. The **causes** clause of an action effect is applied, and state relations and constraints are conditional on, the current state at step i. Action effects add the fluent literal in the **causes** clause to a state in which the action in the head is planned if the previous state satisfies its body. Preconditions prohibit the action in the head from being planned from a previous state which satisfies its body but does not satisfy the fluent literal in the **requires** clause. State relations force the state literal in the head to hold in a

state which satisfies its body. State constraints prohibit any state that satisfies its body.

The **requires** clause of action preconditions is not standard in BC and is added since it has a closer correspondence to the ASP encoding. It is simply an alias for the BC law:

$$\langle precondition : pl, i-1\rangle \textbf{ before } \langle action : pl, i\rangle \textbf{ if } \langle conditions : pl, i-1\rangle$$

The semantics of these types of law can be described more intuitively as follows:

1. Action effects are causal relations, defining state transitions, by linking actions to their direct deterministic Markovian and instantaneous effects[27] on the current state when executed. These effects may change dynamically if conditioned on the previous state.

2. Action preconditions are enabling constraints describing direct conditions on actions that must hold in the previous state before they can be planned in the current state.

3. State variable relations[28] can define both causal and non-causal relations, these can be the indirect effects of actions (which can avoid the need to define complex conditional effects that typically have poor elaboration tolerance) and dependencies between the values of different state variables (including recursive and transitive dependencies).

4. State variable constraints can define non-casual relations and indirect enabling constraints. These are always constraints between the values of different state variables, which specify conditions that cannot physically hold, to prevent invalid states. These can define enabling constraints, as they prevent an action from being planned if its effects lead to an invalid state[29], this can be a much more compact and elaboration tolerant method of defining enabling conditions than action preconditions.

[27]Deterministic Markovian and instantaneous effects of an action a depend only on the previous state s, and are applied only in the resulting state s', such that there is one possible s' for the transition $\langle s, a, s'\rangle$.

[28]It is possible to specify a planning domain without state relations and constraints, but this can lead to errors and poor elaboration tolerance, because action effects and preconditions also need to specify their indirect effects and enabling conditions, which must be carefully engineered to ensure an invalid state is never reached. This is one of the main benefits of ASP based planning over classical heuristic planning.

[29]It is important to note that because fluents can only take one value in each state, non-boolean fluents do not need state variable constraints to prevent them from taking more than one value at a time.

Definition 3.5.5 (Domain Laws) Formally, the domain laws are defined by the tuple:

$$R^{tl} = \langle E^{tl}, P^{tl}, V^{tl}, U^{tl} \rangle \tag{3.36}$$

Where each element is a finite set, E^{tl} is the set of *action effects*, P^{tl} the set of *action preconditions*, V^{tl} the set of *state variable relations*, and U^{tl} the set of *state variable constraints*. These are formally declared by the above laws of action language BC, using the short-hand notation for type constrained sorts defined in 3.21, 3.22, and 3.23, as follows:

Effect[30]:	$R\$\kappa_r \Rightarrow a_i^{pl}(\bar{T}\$\bar{\kappa}_n^a)$ **causes** $f_i^{pl}(\bar{T}\$\bar{\kappa}_n^f) = \Upsilon\$\kappa_\upsilon$ **if** $cons_{i-1}^{pl}$
Positive Precondition:	$R\$\kappa_r \Rightarrow a_i^{pl}(\bar{T}\$\bar{\kappa}_n^a)$ **requires** $f_{i-1}^{pl}(\bar{T}\$\bar{\kappa}_n^f) = \Upsilon\$\kappa_\upsilon$ **if** $cons_{i-1}^{pl}$
Negative Precondition[31]:	$R\$\kappa_r \Rightarrow a_i^{pl}(\bar{T}\$\bar{\kappa}_n^a)$ **requires** $f_{i-1}^{pl}(\bar{T}\$\bar{\kappa}_n^f) \neq \Upsilon\$\kappa_\upsilon$ **if** $cons_{i-1}^{pl}$
Fluent Relation:	$f_i^{sl}(\bar{T}\$\bar{\kappa}_n^f) = \Upsilon\$\kappa_\upsilon$ **if** $cons_i^{sl}$
Static Relation:	$c_i^{sl}(\bar{T}\$\bar{\kappa}_n^c)$ **if** $cons_i^{sl}$
Constraint:	**impossible** $cons_i^{sl}$

Where the notation $\bar{T}\$\bar{\kappa}_n^s$ is a vector of short-hand type constrained entity variables of the form $(T_0\$\kappa_0^s, ..., T_n\$\kappa_n^s)$, the superscript $s \in \{a, f, c\}$ is the sort which that vector is related to, and the conditions $cons_i^l$ contain: a) the set of conditions of the law; b) a conjunction of state literals at the abstraction level denoted by the super-script l and c) the time step denoted by the sub-script i. The type constraint on a given entity variable does not have to be the same for every place the variable occurs in a domain law, but for some entity to be substituted for that variable, the entity must satisfy every type constraint on that variable.

[30]The fluent in the head of an effect must be a positive inertial fluent. They cannot be negative because inertial fluents can be non-boolean (whereby they can have more than two possible values), and therefore a negative effect (that deletes a positive literal from the state) would have a non-deterministic effect (which is undefined and not permitted). Explicit delete lists for actions are not needed (as they are in many classical planning languages, such as PDDL), since the continuity constraint and closed world assumption require that each state literal take exactly one value at any given time and all other values are implicitly false.

[31]For a boolean fluent, a literal $f_i^l(\bar{t}) = false$ is a positive literal and cannot be considered a negative literal, since it explicitly sets the value of the variable to false, but for a non-boolean fluent, a literal $f_i^l(\bar{t}) \neq \upsilon$ is a negative literal as it does not explicitly set the value but instead prevents υ for being the value.

The ASP encoding of the action language BC used by ASH is different to the standard encoding. To ensure the semantics and soundness of the language are preserved, a bidirectional translation between the standard encoding and the encoding used by ASH is therefore defined in Appendix A. Encoding the laws in ASH is simple and defined as follows:

- For each Action Effect;
  - Replace: $R\$\kappa_r \Rightarrow a_i^{pl}(\bar{T}\$\bar{\kappa}_n^a)$ **causes** $f_i^{pl}(\bar{T}\$\bar{\kappa}_m^f) = \Upsilon\$\kappa_v$ **if** $cons_{i-1}^{pl}$
  - With: $effect(pl, R, a(\bar{T}_n^a), i, f(\bar{T}_m^f), \Upsilon, i) \leftarrow \epsilon(cons_{i-1}^{pl}), body^{pl}(a, f)$
- For each Positive Action Precondition;
  - Replace: $R\$\kappa_r \Rightarrow a_i^{pl}(\bar{T}\$\bar{\kappa}_n^a)$ **requires** $f_{i-1}^{pl}(\bar{T}\$\bar{\kappa}_m^f) = \Upsilon\$\kappa_v$ **if** $cons_{i-1}^{pl}$
  - With: $precond(pl, R, a(\bar{T}_n^a), i, f(\bar{T}_m^f), \Upsilon, true, i) \leftarrow \epsilon(cons_{i-1}^{pl}), body^{pl}(a, f)$
- For each Negative Action Precondition;
  - Replace: $R\$\kappa_r \Rightarrow a_i^{pl}(\bar{T}\$\bar{\kappa}_n^a)$ **requires** $f_{i-1}^{pl}(\bar{T}\$\bar{\kappa}_m^f) \neq \Upsilon\$\kappa_v$ **if** $cons_{i-1}^{pl}$
  - With: $precond(pl, R, a(\bar{T}_n^a), i, f(\bar{T}_m^f), \Upsilon, false, i) \leftarrow \epsilon(cons_{i-1}^{pl}), body^{pl}(a, f)$
- For each Fluent State Variable Relation;
  - Replace: $f_i^{sl}(\bar{T}\$\bar{\kappa}_n^f) = \Upsilon\$\kappa_v$ **if** $cons_i^{sl}$
  - With: $holds(sl, f(\bar{T}_n^f), \Upsilon, i) \leftarrow \epsilon(cons_i^{sl}), body^{sl}(f)$
- For each Static State Variable Relation;
  - Replace: $c_i^{sl}(\bar{T}\$\bar{\kappa}_n^c)$ **if** $cons_i^{sl}$
  - With: $is(sl, c(\bar{T}_n^c)) \leftarrow \epsilon(cons^{sl}), body^{sl}(c)$
- For each State Variable Constraint;
  - Replace: **impossible** $cons_i^{sl}$
  - With: $\epsilon(cons_i^{sl})$

Where the conditions $cons_i^l$ are translated by replacing all state variables with their ASP encoding (as defined in Section 3.5.1), such that $\epsilon(cons_i^l) = \{\epsilon(d_i^l = \upsilon) \mid (d_i^l = \upsilon) \in cons_i^l\}$, and the $body^l(...)$ of each ASP rule contains a sort constraint for each sort occurring in any clause of the domain law[32]. Notice that action effects and preconditions have a specialised encoding, where the action and state literal in the head are enclosed in predicates $effect$ and *precond*. These allow the domain-independent rules of the operational modules of ASH to reason with and apply effects and enforce preconditions in a general way (see Section 4.4.3).

The following is a sub-set of the core domains laws for the BWP. The full list is given in Appendix B and the complete ASP encoding can be found at Kamperis, 2023.

- Action Effects:
  - When a robot moves its location changes accordingly;

$$R\$robot \Rightarrow\ move^{pl}(L\$location)\ \textbf{causes}\ in^{pl}(R\$object) = L\$location$$

  - When a robot grasps an object with a grasper the object becomes grasped;

$$R\$robot \Rightarrow grasp^{pl}(G\$grasper, O\$graspable)$$
$$\textbf{causes}\ grasping^{pl}(R\$robot, G\$grasper) = O\$graspable$$

  - A robot can change the configuration of its components;

$$R\$robot \Rightarrow configure^{pl}(C\$component, S\$state)$$
$$\textbf{causes}\ configuration^{pl}(R\$robot, C\$component) = S\$state$$

- Action Preconditions:
  - A robot can only move between connected locations;

$$R\$robot \Rightarrow move^{pl}(L\$location)$$
$$\textbf{requires}\ connected^{pl}(L_1\$location, L_2\$location) = true$$
$$\textbf{if}\ in^{pl}(R\$robot) = L_2\$location, pl < 3$$

[32]To constrain the type of a parameter or value variable to a sub-class of the type constraint used to declare the sort, an extra type constraint must also be added to $body^{pl}$ (using the $isa^l(T, \kappa)$ predicate).

– A robot can only move if all of its manipulator arms are retracted;

$$R\$robot \Rightarrow move^{pl}(L\$location)$$

$$\textbf{requires } configuration^{pl}(R\$robot, C\$component) = retracted$$

$$\textbf{if } part^{pl}(R\$robot, C\$component), isa^{pl}(C, extensible)$$

– A robot can only grasp an object which shares its location;

$$R\$robot \Rightarrow grasp^{pl}(G\$grasper, O\$graspable)$$

$$\textbf{requires } in^{pl}(R\$object) = in^{pl}(O\$object)$$

– A robot can only grasp an object with a grasper if the grasper is extensible and it is extended or if it is attached to another extensible component[33] that is extended;

$$R\$robot \Rightarrow grasp^{pl}(G\$grasper, O\$graspable)$$

$$\textbf{requires } configuration^{pl}(R\$robot, G\$component) = extended$$

$$\textbf{if } part^{pl}(R\$robot, G\$component), isa^{pl}(G, extensible)$$

$$R\$robot \Rightarrow grasp^{pl}(G\$grasper, O\$graspable)$$

$$\textbf{requires } configuration^{pl}(R\$robot, E\$component) = extended$$

$$\textbf{if } part^{pl}(R\$robot, G\$component), part^{pl}(R\$robot, E\$component),$$

$$sib^{pl}(G, E), isa^{pl}(E, extensible)$$

- State Variable Relations:

  – An object that is grasped by a robot shares the location of the robot;

  $$in^{pl}(R\$object) = in^{pl}(O\$object) \textbf{ if } grasping^{pl}(R\$robot, G\$grasper) = O\$graspable$$

  – A block that is on a table or the side of a table is the base of a tower;

  $$tower\_base^{sl}(B\$block, T\$tower) = true \textbf{ if } on^{sl}(B\$object) = T\$surface$$

  – A block that is the base of a tower is in that tower;

  $$in\_tower^{sl}(B\$block, B\$block) = true \textbf{ if } tower\_base^{sl}(B\$block) = true$$

[33] The siblings relation between classes *limb* and *hand* to the ancestor class *arm* on the override classes *extensible* and *grasper* respectively represents the fact that a *hand* is attached to a *limb*.

– A block that is on top of another block is in the same tower as the block it is on;

$$in\_tower^{sl}(B_1\$block, B_2\$block) = true$$

$$\textbf{if } on^{sl}(B_1\$object) = B_2\$surface, in\_tower^{sl}(B_2\$block, B_3\$block) = true$$

- State Variable Constraints:
  - A block cannot have more than one block on top of it;

    $$\textbf{impossible } on^{sl}(B_1\$block) = B_2\$block, on^{sl}(B_1\$object) = B_3\$surface, B_2 \neq B_3$$

  - A robot cannot be grasping a block that has another block on top of it;

    $$\textbf{impossible} grasping^{sl}(R\$robot, G\$grasper) = B_1\$block, on^{sl}(B_1\$object) = B_2\$surface$$

Static definitions and relations that encode the static structure and map of the domain are written simply as state relations, sometimes with no conditions. For example, the statement $connected^{l}(store\_room, hallway)$ unconditionally defines the connection between the store room and hallway and the relation $connected^{1}((R, X_1)\$cell, (R, X_2)\$cell)$ **if** $X_2 = X_1+1$ defines the connections between adjacent cells of a room in the ground model.

As with the domain sorts, during program grounding, any laws containing override classes in their type constraints are expanded automatically, to obtain versions related to the condensed and original models automatically. This avoids the need to manually re-write the domain laws for the condensed abstract model (as was necessary in past work that used similar abstractions), and obtains a far more compact encoding, over the more cumbersome entity constant based relation used in Sridharan, Gelfond, et al., 2019. The following shows how the action effect for the locomotive *move* action expands to a version for moving between rooms in the condensed model and for moving between cells in the original:

$$R\$robot \Rightarrow move^{pl}(L\$location) \textbf{ causes } in^{pl}(O\$object) = L\$location \quad (3.37)$$

$$R\$robot \Rightarrow move^{2}(L\$room@location) \textbf{ causes } in^{2}(O\$object) = L\$room@location \quad (3.38)$$

$$R\$robot \Rightarrow move^{1}(L\$cell@location) \textbf{ causes } in^{1}(O\$object) = L\$cell@location \quad (3.39)$$

3.5.8 State Transition Systems

A state transition system described by the domain laws can be viewed as a directed graph, whose nodes are states and arcs are transitions (labelled by actions). Speaking conceptually, action effects "create" arcs of the graph, action preconditions "delete" a sub-set of those arcs, and state variable constraints "delete" states and all arcs connected to those states. Planning can then be seen as searching for a path within this graph[34] (see Section 2.1 for details).

Each planning domain model has its own state transition system graph (as described in Section 3.5.2). Given the specification of an exhaustive deductive state abstraction mapping, each node of an abstract model's graph is then mapped to from at least one node of its original model's graph, and each node of the original maps to exactly one abstract of the abstract.

The example graph in Figure 3.6 describes a state transition system, in which a robot r is able to move between two rooms a and b, and is able to close or open a door d between those rooms. The solid arrows indicate transitions that can occur from the execution of a given action in any the four possible states of this system. The dashed arrows (on the lower states) indicate a prohibited transition, that moving between a and b is not possible when d is closed. Notice for example, that moving has no effect on the state of the door, since it is not one of the effects of the move action, and therefore stays the same by inertia.

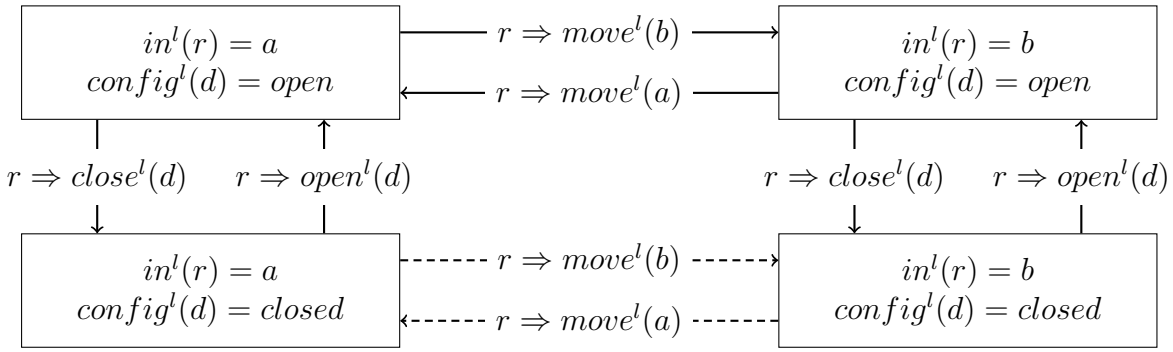


Figure 3.6: An Example State Transition System Graph.

[34] In practice, the transition graph is not actually constructed, as this would be intractable even for simple problems. The graph is simply representative of the possible states and transitions of a planning domain.

3.5.9 State Abstraction Mappings

The state abstraction mappings propagate the state representation in an upwards manner, from the current planning level pl to the previous adjacent level $pl + 1$. This links together the state transition systems described by each domain model in the hierarchy, and maintains a consistent conforming state pair over the state representation levels sl. Where a state or state literal of an original model, conforms with one of its abstract model, if the original maps to the abstract deductively, whereby the original cannot map to another abstract. More formally, conformance between abstract and original state literals is given by:

$$\exists f_i^{pl}(\bar{t}) = \nu_1 \in \sigma_i^{pl}, f_i^{pl+1}(\bar{t}) = \nu_2 \in \sigma_i^{pl+1}.f_i^{pl}(\bar{t}) = \nu_1 \mapsto f_i^{pl+1}(\bar{t}) = \nu_2. \tag{3.40}$$

There are two types of state abstraction mapping, fluent and static. Fluent mappings are used primarily to enforce plan conformance. They allow reasoning about the achievement of sub-goal literals, planned in and relative to the state representation of an abstract model, when refinement planning in the original. Static mappings ensure that static relations remain invariant over the abstraction hierarchy. The general form of the mapping is similar to a state relation, except: a) the result is replaced by an abstract state variable (at abstraction level $pl + 1$); and b) the conditions are replaced by an original state variable (at abstraction level pl) and a set of ancestry constraints if the abstract variable is a condenser, as follows:

$$\langle abstract\ variable : pl + 1, i \rangle \textbf{ if } \langle original\ variable : pl, i \rangle \textbf{ and } \langle ancestry\ constraints : pl \rangle$$

Definition 3.5.6 (State Abstraction Mappings) Formally, the state abstraction mappings are defined by the tuple:

$$M^{tl-1} = \langle FM^{tl-1}, CM^{tl-1} \rangle \tag{3.41}$$

Where $FM^{pl,pl+1}$ are *fluent mappings* and $CM^{pl,pl+1}$ are *static mappings*, defining the relation between the dynamic and static state respectively. These are declared by the above law of action language BC, using the short-hand notation of type constraints, as follows:

Fluent Mapping: $fa_i^{pl+1}(\bar{T}\$\bar{\kappa}_n^{fa}) = \Upsilon\$\kappa_v$ **if** $fo_i^{pl}(\bar{T}\$\bar{\kappa}_n^{fo}) = \Upsilon\$\kappa_v, cons^{pl}$

Static Mapping: $ca_i^{pl+1}(\bar{T}\$\bar{\kappa}_n^{ca})$ **if** $co_i^{pl}(\bar{T}\$\bar{\kappa}_n^{co}), cons^{pl}$

Where fa and fo, and ca and co, are used to denote the abstract and original versions of the variables[35]. The notation is similar to the domain laws, except $cons^{pl}$ contains (a possibly empty) set of ancestry constraints related to parameter or value variables of condenser state variables, such that $cons^{pl} \subseteq \{chi^{pl}(t^a, t^o) \vee des^{pl}(t^a, t^o) \mid t^a \in \bar{T}^{da}, t^o \in \bar{T}^{do}\}$ where $d \in \{f, c\}$. For an exhaustive specification, every state variable must have an abstraction mapping.

The ASP encoding of a state abstraction mapping is again similar to a state relation:

- For each fluent state abstraction mapping:
  - Replace: $f_i^{pl+1}(\bar{T}\$\bar{\kappa}_n^{f}) = \Upsilon\$\kappa_v$ **if** $f_i^{pl}(\bar{T}\$\bar{\kappa}_n^{f}) = \Upsilon\$\kappa_v, desc^{pl}$
  - With: $holds(pl+1, f(\bar{T}_n^f), \Upsilon, i) \leftarrow holds(pl, f(\bar{T}_n^f), \Upsilon, i), \epsilon(cons^{pl}), body^{sl}(fa, fo)$
- For each static state abstraction mapping:
  - Replace: $c_i^{pl+1}(\bar{T}\$\bar{\kappa}_n^{c})$ **if** $c_i^{pl}(\bar{T}\$\bar{\kappa}_n^{c}), body^{sl}$
  - With: $is(pl+1, c(\bar{T}_n^{ca})) \leftarrow is(pl, c(\bar{T}_n^{co})), \epsilon(cons^{pl}), body^{sl}(ca, co)$

Where ancestry constraints are translated by replacing all constraints with their ASP encoding as defined in Section 3.5.4, such that $\epsilon(cons^{pl}) = \{child\_of(l, \phi_1, \phi_2) \mid chi^l(\phi_1, \phi_2) \in cons^{pl}\} \cup \{desce\_of(l, \phi_1, \phi_2) \mid des^l(\phi_1, \phi_2) \in cons^{pl}\}$, and the $body^{sl}(...)$ contains a single sort constraint for the given state variable in the abstraction mapping.

The nature of these mappings are specific to the type of abstract model being created. Recalling that if the abstract state representation relative to the original is unchanged, as in relaxed models, the mapping is one-to-one, such that all state variables map to themselves, making the mapping trivial to define. Whereas, if a state abstraction is used, as in condensed

[35]The abstract and original versions of the variables have the same name in both models. This is a point of novelty over REBA (Sridharan, Gelfond, et al., 2019) which required renaming of abstract state variables.

models, the mappings are instead many-to-one. For condensed models, such mappings are very simple and general to define. All condenser variables simply map from their descendants to their ancestors, and all other variables map to themselves. More formally:

- For all condenser state variables, and for all variables $T_k\$\kappa_k$ in their parameter vector $\bar{T_n}\$\bar{\kappa_n}$ constrained by an override class add the state abstraction mapping:

$$f_i^{pl+1}(\bar{T_n}) = \upsilon \ \textbf{if} \ f_i^{pl}(\bar{T_n}) = \upsilon, des^{pl}(T_k, T_k)$$

- For all fluent condensers whose value variable $\Upsilon\$\kappa_\upsilon$ is defined by an override add:

$$f_k^{pl+1}(\bar{t_n}) = \Upsilon \ \textbf{if} \ f_k^{pl}(\bar{t_n}) = \Upsilon, des^{pl}(\Upsilon^{pl+1}, \Upsilon^{pl})$$

An abstract condenser state literal (containing ancestor entities) therefore holds, if any of the original state literals (containing descendants) that map to the abstract holds.

The following encode the state abstraction mappings from the BWP domain, defining a disjunctive definition where an object is in a room if it is in any cell of that room. If a robot moves to a room in an abstract plan, this can then be achieved by moving to any cell of that room in the refinement. This is admissible, since if the room is accessible, then it will always be possible to move into the room by reaching some cell (no problem constraint can block this, as condensed models can only generalise planning constraints, not remove them).

- If an object is in a location that is a descendant of some ancestor location, then that object is also located in the ancestor location;

$$in^{pl+1}(O\$object) = AL\$location \ \textbf{if} \ in^{pl}(O\$Object) = DL\$location, des^{pl}(AL, DL)$$

- If descendant locations are connected then so are their ancestors;

$$connected^{pl+1}(AL_1\$location, AL_2\$location) = true$$
$$\textbf{if} \ connected^{pl}(DL_1\$location, DL_2\$location) = true,$$
$$desc^{pl}(AL_1, DL_1), desc^{pl}(AL_2, DL_2)$$

3.6 HCR Planning Problems

This section provides formal definitions for hierarchical and monolevel planning problems.

3.6.1 Hierarchical Planning Problems

A HCR hierarchical planning problem fundamentally requires generating and successively refining plans over multiple levels of an abstraction hierarchy defined by a hierarchical planning domain. This is done by solving the most abstract monolevel problem, and then iteratively solving progressively more concrete monolevel planning problems, downwards over the hierarchy in descending order to the ground-level. This movement over the levels in the abstraction hierarchy during hierarchical planning is called the vertical axis of HCR planning.

Only at the top-level, in the most abstract and least constrained domain model, must a classical monolevel problem be pre-defined and solved directly. Classical planning problems must always be complete, whereby they start in the initial-state and must achieve the final-goal. At all lower levels, conformance refinement monolevel problems are dynamically created and solved to refine (at least part of) an abstract plan obtained from the previous adjacent abstraction level. Conformance refinement monolevel planning problems can be either complete or partial. A partial-problem refines any contiguous sub-sequence of sub-goal stages, and a complete-problem refines the entire sub-goal sequence from the previous abstraction level. A partial-problem with multiple sub-goal stages or a complete-problem is a combined-problem. A sequence of partial-problems is a divided problem, since they represent divisions of the respective combined problem which would include all the sub-goal stages in that sequence of partial-problems. Solving a complete (classical or conformance refinement) problem always obtains a complete plan that finalises the given abstraction level. Solving a sequence of partial-problems requires successively extending the partial monolevel plan into a complete plan, by incrementally solving the partial problems across the goal sequence

in ascending order from left-to-right. This progression across the goal and partial-problem sequence during online planning is called the horizontal axis of HCR planning.

A partial refinement planning problem does not necessarily have to start in the initial-state or achieve the final-goal. It must only achieve a contiguous sub-sequence of the sub-goal stages obtained from the abstract plan at the previous abstraction level. A partial-problem that starts in the initial-state is an initial partial-problem, otherwise it is non-initial. The start state of a non-initial partial-problem is the end state of the monolevel partial-plan that solved the previous partial-problem that the same level. A partial-problem that must achieve the final-goal (and the last sub-goal stage) is a final-problem, otherwise it is non-final. By definition, a partial-problem cannot be both initial and final, since it would then be complete. A sequence of partial refinement problems which starts with an initial problem and ends with a final problem, defines a division over the respective complete refinement problem. Solving such a sequence of partial-problems and concatenating their partial-plans therefore constitutes the solution to the complete problem. This is because it results in a monolevel plan that is initial, final, achieves the complete sequence of sub-goal stages obtained from the abstract plan at the previous abstraction level, and is therefore complete.

There are two modes in which a HCR planner can solve a hierarchical planning problem: offline and online planning modes. In offline planning, all refinement problems must be complete, and solved in a single descending ordered sequence, whereby the vertical progression is always downwards. All problems are solved prior to plan execution, because the planner can only yield actions to the robot when a complete ground-level plan is found. In online planning, refinement problems are permitted to be partial; the vertical progression can move up and down the hierarchy, whilst the horizontal progression extends plans along the goal and problem sequence. Only a partial ground-level plan is found prior to execution, and is extended by a left-to-right incrementation over the sequence of partial-problems during execution. The constraints underlying this process are described in the following Chapter 4 where problem division strategies and online planning methods are proposed.

3.6.2 Specifying a Hierarchical Planning Problem

The problem specific planner inputs are needed to specify a hierarchical planning problem.

- *Initial state*: The initial state is the state from which the robot must begin execution. It must include the initial location and configuration of all entities in the domain. It is given as a complete set of positive fluent state literals at the ground-level on step zero.
- *Final-goal*: The final-goal specifies the objective the robot must reach at the end of plan execution. It includes the desired location and configuration of a sub-set of the entities in the domain. It is given as a set of positive and negative fluent state literals at the ground-level with no time step. These are called final-goal literals (defined later).

To obtain a hierarchical planning problem, the initial state and final-goal must be expanded, to create conforming initial states and final-goals over every abstraction level in the hierarchy. This ensures that for all complete monolevel plans in a hierarchical plan: the initial state and the sub-set of the end states related to the final-goal fluents, will both map to each other exhaustively and deductively through the state abstraction mappings.

Definition 3.6.1 (Hierarchical Planning Problem) A hierarchical planning problem in the hierarchical planning domain DD^{tl} is denoted by the tuple:

$$HP^{tl} = \langle DD^{tl}, \Sigma^{tl}, \Omega^{tl} \rangle \tag{3.42}$$

$$\Sigma^{tl} = \{\delta_0^1, ... \delta_0^{tl-1}, \sigma_0^{tl}\} \tag{3.43}$$

$$\Omega^{tl} = \{\omega^1, ... \omega^{tl}\} \tag{3.44}$$

Where Σ^{tl} are the initial states and Ω^{tl} the final-goals, both of which are defined over all models in the abstraction hierarchy at levels $[1, tl]$. At the top-level tl, only a single initial state σ_0^{tl} is needed to solve the top-level classical planning problem, whilst at all lower levels $l \in [1, tl)$ an initial state pair δ_0^l is needed to solve the conformance refinement problems.

3.6.3 Representation of Final-Goals

The final-goal is a logical conjunction of final-goal literals to be achieved at the end of a monolevel plan. A final-goal literal is essentially a fluent state literal with no time step. A fluent state literal can only be assigned as a final-goal literal, if its state variable is declared as a goal-fluent state variable. This is necessary to reason about generating a conforming final-goal over the abstraction hierarchy. When planning, the final-goal is asserted to hold on the terminal state of a plan at some fixed time step k. If there exists a plan that can reach such a state, then the problem is solvable with a plan within time step k.

- Let $gf(f^l(\bar{t}))$ denote the declaration of the fluent $f^l(\bar{t})$ as a final-goal fluent;
  - The ASP encoding is an atom:

$$\epsilon(gf(f^l(\bar{t}))) \ = goal\_fluent(l, g(\bar{t}))$$
$$\epsilon(gf(f^l(\bar{t}))) \ = goal\_fluent(l, g(\bar{t}))$$

- Let $g^l(\bar{t}) = v$ and $g^l(\bar{t}) \neq v$ be positive and negative final-goal literals respectively;
  - The ASP encoding is an atom:

$$\epsilon(g^l(\bar{t}) = v) \ = final\_goal(l, g(\bar{t}), v, true)$$
$$\epsilon(g^l(\bar{t}) \neq v) \ = final\_goal(l, g(\bar{t}), v, false)$$

- Let $\omega^l = \{g^l(\bar{t}) = v, ..., \ g^l(\bar{t}) \neq v, ...\}$ be a final-goal set, a finite set of goal literals;
  - The ASP encoding is a set of atoms:

$$\epsilon(\omega^l) = \{\epsilon(g^l(\bar{t}) = v) \mid (g^l(\bar{t}) = v) \in \omega^l\}$$
$$\cup \{\epsilon(g^l(\bar{t}) \neq v) \mid (g^l(\bar{t}) \neq v) \in \omega^l\}$$

The following rules are a partial encoding of the initial state and final-goals of BWP problems. The rules give the initial and desired goal position of the blocks as depicted in Figures 3.1b and 3.1b. Block 5 and 6 start in cell 0 of the store room, whilst the other four blocks start stacked on the table. The goal is to stack the blocks in descending order towers of unique colour, with block 6 on the right and block 3 on the left. The abstract state and

final-goal literals are the only possible interpretations of those at the ground-level.

$$on_0^1(block_1) = (table, left) \qquad on_0^1(block_2) = (table, right) \tag{3.45}$$

$$on_0^{sl>1}(block_1) = table \qquad on_0^{sl>1}(block_2) = table \tag{3.46}$$

$$on_0^{sl}(block_3) = block_1 \qquad on_0^{sl}(block_4) = block_2 \tag{3.47}$$

$$in_0^1(block_5) = (store\_room, 0) \qquad in_0^1(block_6) = (store\_room, 0) \tag{3.48}$$

$$in_0^{sl>1}(block_5) = store\_room \qquad in_0^{sl>1}(block_6) = store\_room \tag{3.49}$$

$$gf^{pl}(on(B\$block)) \tag{3.50}$$

$$g^1(on(block_3) = (table, left)) \qquad g^1(on(block_6) = (table, right)) \tag{3.51}$$

$$g^{pl>1}(on(block_3) = table) \qquad g^{pl>1}(on(block_6) = table) \tag{3.52}$$

$$gf^{pl}(complete\_tower(C\$colour)) \tag{3.53}$$

$$g^{pl}(complete\_tower(C\$colour) = true) \tag{3.54}$$

3.6.4 Monolevel Planning Problems

A hierarchical planning problem is solved as a sequence of monolevel problems. There is at least one descending sequence of monolevel problems over the vertical axis of planning. If online planning with problem division is enabled, there may also be an ascending sequence of partial-problems over the horizontal axis of planning at each abstraction level.

Definition 3.6.2 (Monolevel Planning Problem) A monolevel planning problem of a hierarchical planning problem HP^{tl} is defined by a four tuple containing: a single domain model for classical problems, or a pair of domain models for refinement problems; the start state; possibly a final-goal; and possibly a conformance constraint. The domain models, initial state, and final goals are obtained from the hierarchical problem definition, whereas the conformance constraint can only be obtained from an abstract plan.

A monolevel problem is denoted by $MP^{pl,\varkappa}_{j,\varphi,\vartheta}$ where $\varkappa$ is the problem's sequence number, j its start step, and φ and ϑ are the first and last indices of its goal-sequence, such that the problem size is $1 + (\vartheta - \varphi)$. A problem is conformance refinement if it includes sub-goal stages, otherwise it is classical. Note that a monolevel problem includes the entire domain definition within it. However, when the problem is solved, the ASP program is grounded to automatically represent only the models relevant to the problem it defines.

If a problem is at the top-level $pl = tl$, then it must be a complete classical planning problem. A complete classical monolevel planning problem is defined by the tuple:

$$MP^{pl,1}_{0,1,1} = CP^{pl} = \ \langle DD^{tl}, \sigma^{pl}_0, \omega^{pl}, \emptyset \rangle \tag{3.55}$$

Where CP^{pl} is a simplified notation for classical problems. The classical problem is initial ($j = 0$), and is the first and only problem at the top-level ($\varkappa = 1$). It contains only the final-goal and no sub-goals, and has a size of 1 to represent inclusion of the final-goal.

If the problem is below the top-level $pl < tl$ then it is a conformance refinement planning problem. It additionally includes a sub-sequence of the conformance constraining sub-goal stages. These sub-goal stages are produced from some abstract plan obtained at the previous level, to be achieved by refinement planning at the current. Conformance refinement problems can have any non-zero size. A complete or partial conformance refinement monolevel planning problem at planning level $pl \in [1, tl)$ is defined by the tuple:

$$MP^{pl,\varkappa}_{j,\varphi,\vartheta} = \begin{cases} \langle DD^{tl}, \delta^{pl}_0, \omega^{pl}, \lambda^{pl+1}_{1,\varpi} \rangle & pl < tl, \varkappa = 1, j = 0, \varphi = 1, \vartheta = clen(pl+1) \quad (3.56a) \\ \langle DD^{tl}, \delta^{pl}_0, \emptyset, \lambda^{pl+1}_{\varphi,\vartheta} \rangle & pl < tl, \varkappa = 1, j = 0, \varphi = 1, \vartheta < clen(pl+1) \quad (3.56b) \\ \langle DD^{tl}, \delta^{pl}_j, \omega^{pl}, \lambda^{pl+1}_{\varphi,\vartheta} \rangle & pl < tl, \varkappa > 1, j \geq 0, \varphi \geq 1, \vartheta = clen(pl+1) \quad (3.56c) \\ \langle DD^{tl}, \delta^{pl}_j, \emptyset, \lambda^{pl+1}_{\varphi,\vartheta} \rangle & pl < tl, \varkappa > 1, j \geq 0, \varphi \geq 1, \vartheta < clen(pl+1) \quad (3.56d) \end{cases}$$

Where $clen$ is a function of the form $len : \mathbb{N}_{>0} \rightarrow \mathbb{N}_{>0}$ such that $l \mapsto k$ defines the complete plan length at level l. The term $\varpi = clen(pl{+}1)$ denotes the length of the complete monolevel

plan from the previous level. This also defines the number of sub-goal stages that have to be refined to find a complete refinement plan at the current level. Therefore, $\lambda_{\varphi,\vartheta}^{pl+1} \subseteq \lambda_{1,\varpi}^{pl+1}$ is a contiguous sub-sequence of sub-goal stages refined by a partial-problem.

A problem is *initial* if it starts in the initial state at step $j = 0$, otherwise it is *non-initial* and starts in some intermediate state at some step $j > 0$. An initial problem is the first in sequence problem at the given level such that $\varkappa = 1$. All non-initial problems are later in the sequence such that $\varkappa > 1$. A problem is *final* if it includes the final-goal, otherwise it is *non-final*. If a problem is both initial and final then it is *complete*, otherwise it is *partial*. A partial problem that is both non-initial and non-final is called an *intermediate* partial-problem. A classical problem must always be complete, whereas a conformance refinement problem can be partial. Therefore, given the above definitions, case 3.56a is a complete refinement problem, case 3.56b is an initial partial refinement problem, case 3.56c is a final partial refinement problem, and case 3.56d is an intermediate partial problem.

If a conformance refinement problem is initial ($j = 0$) then $\varphi = 1$, and it must include the first sub-goal stage produced from the previous level (3.56a and 3.56b). If the problem is non-initial ($j > 0$) then $\varphi \in (1, clen(pl + 1)]$, and is a partial-problem which extends the existing solution to previous partial-problems at the given level (3.56c and 3.56d). If the problem is final then it is the last in sequence partial problem at that level. It must include both the last sub-goal stage produced from the previous level and the final-goal. Where the last sub-goal stage is produced from the final-goal achieving abstract action set.

Most elements of a monolevel problem are obtained from the hierarchical problem specification: the domain model being planned in is $DD^{tl}[HP^{tl}]$; if the problem is initial the start state is $\{\sigma_0^{pl}, \delta_0^{pl}\} \in \Sigma^{tl}[HP^{tl}]$; and if the problem is final the final-goal is $\omega^{pl} \in \Omega^{tl}[HP^{tl}]$. If the problem is non-initial, it starts in the terminal state of the partial plan which solved the previous partial-problem at $\varkappa - 1$. The sub-goal stages $\lambda_{\varphi,\vartheta}^{pl+1}$ are then obtained from the abstract plan that was generated at the previous adjacent abstraction level $pl + 1$.

3.6.5 Templating Partial-Problems

Non-initial partial-problems can only be fully defined once their start state is obtained from the end state of the partial-plan that solves the previous partial-problem at the same level. To remedy this, a sequence of partial-problems can be templated, whereby the refinement problem of a contiguous sub-sequence of sub-goal stages is pre-allocated to that problem. This makes it possible to make multiple divisions over a combined refinement problem of an abstract plan, despite that only the start state of the first partial-problem is known.

Templating of partial-problems is handled by the Δ function. This maps a partial-problem's abstraction level and sequence number to the range of sub-goal stages it refines:

$$\Delta : \mathbb{N}, \mathbb{N}, \rightarrow (\mathbb{N}, \mathbb{N}) \tag{3.57}$$
$$\text{such that } pl, \varkappa \mapsto (\varphi, \vartheta)$$

The function is partial of the number of partial-problems, since no fixed limit on the number of problems is imposed prior to planning. Intuitively, the only constraint on this function is that the sub-goal stage range of partial-problems must follow contiguously such that:

$$\forall pl, \varkappa.\ \Delta(pl, \varkappa) = (\varphi, \varrho), \Delta(pl, \varkappa + 1) = (\varrho, \vartheta) \tag{3.58}$$

The function is constructed by a problem division strategy during online planning.

The ability of HCR planning to solve combined refinement problems can overcome dependencies between sub-goal stages. The desire is to maximise the number of problem divisions, but to divide only between independent partial-problems (grouping dependent sub-goals together), since this will minimise the complexity of the overall problem and achieve the best possible quality plan (under the given conformance constraint). A sequence of partial-problems are independent if the concatenated partial-plans are equal in length to a complete plan generated from solving the respective complete refinement problem. Unfortunately, to know this for certain requires solving both the complete and divided problems.

3.7 Planning Problem Solutions

This section provides formal definitions for the solutions to hierarchical and monolevel planning problems. The operational models of ASH, presented in Section 4.4, contain the encoding of these definitions as ASP rules, which allows ASP to perform HCR planning.

3.7.1 Classical Problem Solutions

Classical planning finds the solution to a monolevel planning problem by directly solving the complete original version of the problem: namely the problem containing the original domain model, and the initial state and final-goal. The solution to a classical monolevel planning problem is called a *classical complete monolevel plan*. A classical complete monolevel plan is simply one that starts in the initial state and achieves the final-goal of the problem.

Definition 3.7.1 (Final-Goal Satisfaction and Achievement) A state σ_k^{pl} *satisfies* the final-goal set ω^{pl} *iff* all final-goal literals in the set are in that state, such that:

$$fgsat(\omega^{pl}, \sigma_k^{pl}) \Leftrightarrow (\forall(g^{pl}(\bar{t}) = \upsilon) \in \omega^{pl}.\ (f_k^{pl}(\bar{t}) = \upsilon) \in \sigma_k^{pl}) \wedge \\ (\forall(g^{pl}(\bar{t}) \neq \upsilon) \in \omega^{pl}.\ (f_k^{pl}(\bar{t}) = \upsilon) \notin \sigma_k^{pl}) \tag{3.59}$$

Whereby, all positive final-goal literals are in the state, and all negative literals are not. A monolevel plan $\pi_{j,k}^{pl}$ *achieves* the final goal ω^{pl} *iff* its end state satisfies the final-goal:

$$fgach(\omega^{pl}, \pi_{j,k}^{pl}) \Leftrightarrow fgsat(\omega^{pl}, \sigma_k^{pl}[\delta_k^{pl}[\tau_{k-1}^{pl}]]), \tau_{k-1}^{pl} \in \pi_{j,k}^{pl} \tag{3.60}$$

An initial monolevel plan $\pi_{0,k}^{pl}$ (one that starts at $j = 0$) is *final* on monolevel problem $MP_{0,1,\vartheta}^{pl,1}$ *iff* the plan achieves the final-goal of the problem such that $fgach(\omega^{pl}[MP_{0,1,\vartheta}^{pl,1}], \pi_{0,k}^{pl})$.

Definition 3.7.2 (Classical Complete Monolevel Plan) An initial monolevel plan $\pi_{0,k}^{pl}$ is *classical complete* on a complete classical monolevel planning problem CP^{tl} *iff* the plan

both start's in the problem's initial state and includes the final-goal. This is such that:

$$clc(MP^{pl,1}_{0,1,1}, \pi^{pl}_{0,k}) \Leftrightarrow fgach(\omega^{pl}[CP^{tl}], \pi^{pl}_{0,k}) \wedge \sigma^{pl}_0[\delta^{pl}_0[\tau^{pl}_0]] = \sigma^{pl}_0[CP^{tl}], \tau^{pl}_0 \in \pi^{pl}_{0,k} \tag{3.61}$$

Similarly, an initial monolevel plan $\pi^{pl}_{0,k}$ is *classical complete* on a complete conformance refinement monolevel planning problem $MP^{pl,1}_{0,1,\vartheta}$ *iff* the plan starts in the problem's initial state pair and is final on the problem. This is also true if for a sequence of partial refinement problems which divide the complete refinement problem, in which case the monolevel plan is the concatenation of a sequence of partial-plans. This is such that:

$$clc(MP^{pl,1}_{0,1,1}, \pi^{pl}_{0,k}) \Leftrightarrow fgach(\omega^{pl}[MP^{pl,1}_{0,1,\vartheta}], \pi^{pl}_{0,k}) \wedge \delta^{pl}_0[\tau^{pl}_0] = \delta^{pl}_0[MP^{pl,1}_{0,1,\vartheta}], \tau^{pl}_0 \in \pi^{pl}_{0,k} \tag{3.62}$$

Recalling that, for classical problems it is necessary that the initial state be only a single state σ^{pl}_0, because the state is represented at only a single level pl during classical planning. Whereas for conformance refinement problems it is necessary that the initial state be an state pair δ^{pl}_0, because the state is represented over an adjacent pair of abstraction levels $sl \in \{pl, pl+1\}$ during conformance refinement planning (to allow abstract reasoning).

3.7.2 Conformance Refinement Problem Solutions

Conformance refinement planning finds the solution to a monolevel planning problem by refining the solution to an abstract model of a problem into a solution to the original. Such a problem contains both the abstract and original domain model, the conformance constraint obtained from an abstract plan, and the initial state and final-goal. The solution to a complete conformance refinement monolevel planning problem is a *conformance complete monolevel plan*. Such a plan is classical complete, is a solution to the original problem, satisfies the conformance constraint, and is thus also a refinement of the abstract plan.

The conformance constraint used in conformance refinement planning acts as a specification for the structure of a refined plan. When a refined plan is generated, a conformance mapping is built, which serves to represent how the refined plan's structure satisfies the constraint, therefore defining how the plan constitutes the refinement of the abstract plan. It is desirable and necessary that this ordering over achievement of sub-goal stages be flexible. Whereby, the number of steps and actions between the achievement of any given sub-goal stage and the next should not be fixed prior to planning. It is necessary, because it is rarely possible or computationally cheap enough to accurately predict the length of a given sub-plan before refinement. It is desirable, because it enables the interleaving property which exists because the planner is able to dynamically delay the achievement of sub-goal stages. However, recall that refined plans are not guaranteed to be minimal in length. This is because the requirement to conform with the abstract plan (achieve the same effects by achieving its sub-goal stages in sequence), might require taking a non-optimal path to the final-goal.

Each sub-goal stage involved in a refinement planning problem is achieved by a distinct sub-plan. A contiguous sequence of sub-plans then forms a partial or complete plan. A contiguous sequence of partial-plans, containing an initial partial-plan and a final partial-plan, can be concatenated to a complete-plan. The start state of each non-initial sub- or partial-plan is known only to satisfy the previous sub-goal stage, and is otherwise unknown until it has been achieved by the previous sub- or partial-plan. The original level action set that causes the terminal state transition of a refined sub-plan is its matching-child, of the parent abstract action set that produced the sub-goal stage refined by that sub-plan.

A core requirement for a refined plan to satisfy a conformance constraint is the concept of unique sub-goal stage achievement. To uniquely achieve a sub-goal stage, there must not be an interpretation of the sub-plan where the matching-child would be anywhere other than the end of the sub-plan. Unique achievement does not require minimal "greedy" achievement, as this would prevent the interleaving property from functioning. Sub-plans are allowed to have actions within them that are not directly related to the achievement, or enabling the

achievement, of its sub-goal stage. A sub-set of actions in a sub-plan might prepare for something needed to achieve a later sub-goal stage that is included in the same combined refinement problem. Therefore, it might be possible to make a specific sub-plan shorter, and allow its sub-goal stage to be achieved earlier in the short-term. However, this might make it harder to achieve the later sub-goal stages, and therefore increase the overall plan length.

There are two different ways the achievement of sub-goal stages in a conformance constraint can be enforced during refinement planning. These are as follows:

1. *Simultaneous Sub-Goal Literal Achievement*: All sub-goal literals in a stage must be uniquely achieved by the matching-child action set of the refined sub-plan of the parent abstract action set. This is such that, all sub-goal literals must be simultaneously satisfied (hold true at the same time) in the terminal state of the refined sub-plan, and to ensure unique achievement, no other state of the sub-plan can also do so.

2. *Sequential Sub-Goal Literal Achievement*: Individual sub-goal literals in a stage can be achieved by any action set of the refined sub-plan of the parent abstract action set. Only a non-empty sub-set of the sub-goal literals must be uniquely achieved by the matching-child. Consequently, each sub-goal literal must be satisfied in any intermediate state of the refined sub-plan. To ensure unique achievement, the matching-child must satisfy at least one of the sub-goal literals that was not satisfied in an intermediate state. Importantly, it is permitted for a sub-goal literal that is satisfied in an intermediate state of a sub-plan, to then be unsatisfied in a later state of the sub-plan.

The conjecture is that simultaneous this is likely to provide the strongest conformance constraint, by applying the greatest restriction on the search space. However, it requires that all abstract effects of the parent action set being refined to be satisfiable in the same original level state, when the original is mapped back to the abstract state. It may not always be possible to do this. This is likely to not be possible during concurrent action planning, in which a large number of abstract actions might be planned on the same step, whose

effects cannot be simultaneously achieved under the stricter constraints of the original model. Sequential achievement is a more relaxed constraint intended to overcome this limitation. Sequential achievement instead allows each individual sub-goal literal to be satisfied in any of the states of a sub-plan. Therefore, if all sub-goal literals in a stage are achievable in some state of the original model, then an abstract action set will then always be refinable.

Definition 3.7.3 (Sub-Goal Stage Satisfaction and Achievement) A sub-goal stage λ_{ϱ}^{pl+1} is satisfied by a state pair δ_k^{pl} *iff* all sub-goal literals are in the original level state:

$$sgsat(\lambda_{\varrho}^{pl+1}, \delta_k^{pl}) \Leftrightarrow \lambda_{\varrho}^{pl+1} \subseteq \sigma_k^{pl+1}[\delta_k^{pl}] \tag{3.63}$$

A sub-plan $\pi_{j,k}^{pl}$ *simultaneously achieves* a sub-goal stage λ_{ϱ}^{pl+1} *iff* it ends in a state that satisfies all sub-goal literals in the stage:

$$sim\_sgach(\lambda_{\varrho}^{pl+1}, \pi_{j,k}^{pl}) \Leftrightarrow sgsat(\lambda_{\varrho}^{pl+1}, \delta_{k+1}^{pl}[\tau_k^{pl}]), \tau_k^{pl} \in \pi_{j,k}^{pl} \tag{3.64}$$

Sub-plan $\pi_{j,k}^{pl}$ *sequentially achieves* sub-goal stage λ_i^{pl+1} *iff* it ends in a state that satisfies at least one sub-goal literal, and all other literals are satisfied in intermediate states:

$$\begin{aligned} seq\_sgach(\lambda_{\varrho}^{pl+1}, \pi_{j,k}^{pl}) \Leftrightarrow sgsat(B, \delta_{k+1}^{pl}[\tau_k^{pl}]), B \subseteq \lambda_{\varrho}^{pl+1}, \tau_k^{pl} \in \pi_{j,k}^{pl}, \\ (\forall (f_{\varrho}^{pl+1}(\bar{t}) = \upsilon) \in (\lambda_{\varrho}^{pl+1} - B). \\ \exists\, \tau_i^{pl} \in \pi_{j,k}^{pl}.\ sgsat(\{f_{\varrho}^{pl+1}(\bar{t}) = \upsilon\}, \delta_{i+1}^{pl}[\tau_i^{pl}])) \end{aligned} \tag{3.65}$$

Sub-plan $\pi_{j,k}^{pl}$ *achieves* sub-goal stage λ_{ϱ}^{pl+1} *iff* it simultaneously or sequentially achieves it:

$$sgach(\lambda_{\varrho}^{pl+1}, \pi_{j,k}^{pl}) \Leftrightarrow sim\_sgach(\lambda_{\varrho}^{pl+1}, \pi_{j,k}^{pl}) \vee seq\_sgach(\lambda_{\varrho}^{pl+1}, \pi_{j,k}^{pl}) \tag{3.66}$$

The sub-plan $\pi_{j,k}^{pl}$ *uniquely achieves* subgoal stage λ_{ϱ}^{pl+1} *iff* it does not contain within it another shorter sub-plan that would *intermediately achieve* the stage in less steps:

$$uni\_sgach(\lambda_{\varrho}^{pl+1}, \pi_{j,k}^{pl}) \Leftrightarrow sgach(\lambda_{\varrho}^{pl+1}, \pi_{j,k}^{pl}) \ \wedge\ (\nexists\, \pi_{j,a}^{pl} \in \pi_{j,k}^{pl}.\ sgach(\lambda_{\varrho}^{pl+1}, \pi_{j,a}^{pl})) \tag{3.67}$$

Definition 3.7.4 (Conformance Constraint Satisfaction) A monolevel plan $\pi^{pl}_{j,k}$ satisfies a conformance constraint containing the sub-goal stages $\lambda^{pl+1}_{\varphi,\vartheta}$ *iff* it uniquely achieves all stages in ascending order, such that:

$$\begin{aligned} con(\lambda^{pl+1}_{\varphi,\vartheta}, \pi^{pl}_{j,k}) \Leftrightarrow (\forall \lambda^{pl+1}_{\varrho} \in \lambda^{pl+1}_{\varphi,\vartheta-1}.\ \exists!\ (\pi^{pl}_{a,b}, \pi^{pl}_{b,c}) \subseteq \pi^{pl}_{j,k}. \\ (uach(\lambda^{pl+1}_{\varrho}, \pi^{pl}_{a,b}) \wedge uach(\lambda^{pl+1}_{\varrho+1}, \pi^{pl}_{b,c}))) \\ \wedge uach(\lambda^{pl+1}_{\varphi}, \pi^{pl}_{j,x}) \wedge uach(\lambda^{pl+1}_{\vartheta}, \pi^{pl}_{y,k}) \end{aligned} \tag{3.68}$$

Whereby, for all sub-goal stages at indices $\varrho \in [\varphi, \vartheta - 1)$ there is exactly one contiguous pair of sub-plans $(\pi^{pl}_{a,b}, \pi^{pl}_{b,c}) \subseteq \pi^{pl}_{j,k}$, where the prior sub-plan uniquely achieves the sub-goal stage λ^{pl+1}_{φ} before the latter uniquely achieves the next stage $\lambda^{pl+1}_{\varrho+1}$. This is such that the end step of the first is the start step of the second. The first sub-plan must commence in the start step of the plan $\pi^{pl}_{j,x}$ and the last sub-plan terminates in the end step of the plan $\pi^{pl}_{y,k}$.

A sub-goal stage is satisfied in an intermediate state if: a) the state has conformance with the stage, and b) the stage is current in the state. That is, the previous stage has been satisfied in a earlier state, and the next stage in a later state, and the current stage must not be satisfied in any state between the earlier and current state (unique achievement).

Definition 3.7.5 (Conformance Complete Monolevel Plan) A monolevel plan $\pi^{pl}_{0,k}$ is conformance complete on planning problem $MP^{pl,1}_{0,1,\vartheta}$ *iff* it is classical complete and satisfies the conformance constraint containing its subgoal stages:

$$\begin{aligned} crc(MP^{pl,1}_{0,1,\varpi}, \pi^{pl}_{0,k}) \Leftrightarrow clc(MP^{pl,1}_{0,1,\varpi}, \pi^{pl}_{0,k}) \wedge \\ con(\lambda^{pl+1}_{1,\varpi}[MP^{pl,1}_{0,1,\varpi}], \pi^{pl}_{0,k}) \end{aligned} \tag{3.69}$$

This also holds if $\pi^{pl}_{0,k}$ is the concatenation of a sequence of partial plans $\{\pi^{pl}_{0,i}, \pi^{pl}_{i,j}, ...\pi^{pl}_{j+x,k}\}$ which solves a partial-problem sequence of length n, such that:

$$\forall \varkappa \in [1, n].\ con(\lambda^{pl+1}_{\varphi,\vartheta}, \pi^{pl}_{i,j} \in \pi^{pl}_{0,k}), \lambda^{pl+1}_{\varphi,\vartheta} \in \lambda^{pl+1}_{0,\varpi}[MP^{pl,1}_{0,1,\varpi}], \Delta(\varkappa) = (\varphi, \vartheta) \tag{3.70}$$

If the partial plan $\pi^{pl}_{i,j} \in \pi^{pl}_{0,k}$ that achieves the last sub-goal stage does not achieve the final-goal such that $uach(\lambda^{pl+1}_{\varpi}, \pi^{pl}_{i,j}) \wedge \neg ach(\omega^{pl}, \pi^{pl}_{i,j})$, then the sub-plan $\pi^{pl}_{j,k} \in \pi^{pl}_{0,k}$ where $ach(\omega^{pl}, \pi^{pl}_{j,k})$ is called the *trailing sub-plan* of $\pi^{l}_{0,k}$. This is because $\pi^{pl}_{j,k}$ does not achieve any sub-goal stage from the previous level and therefore does not refine part of the abstract plan.

Trailing plans can occur when refining from a condensed model. This is because the original state is more detailed than the condensed. For a fluent condenser state variable, many original state literals can map to one given condensed literal. Thus, it is possible for many original states to satisfy the last sub-goal stage, but not the final-goal. Achieving the last sub-goal stage, might not achieve the final-goal, and the trailing plan must do so.

The diagram in Figure 3.7 depicts the conformance refinement planning process. The blue squares are states, red circles actions, and green diamonds sub-goal stages. The upper-level is an abstract plan and the lower-level is its conformance refinement. The downwards vertical dashed arrows indicate the production of sub-goal stages from the direct effects of abstract action sets. The upwards vertical dotted arrows indicate conformance between the plans: the initial states have total conformance (all state original literals map to the abstract), the intermediate states have sub-goal stage conformance (original level states satisfy the sub-goals from the abstract actions and original level actions achieve the same effects), and the end states have final-goal conformance (all final-goal literals map to the abstract).

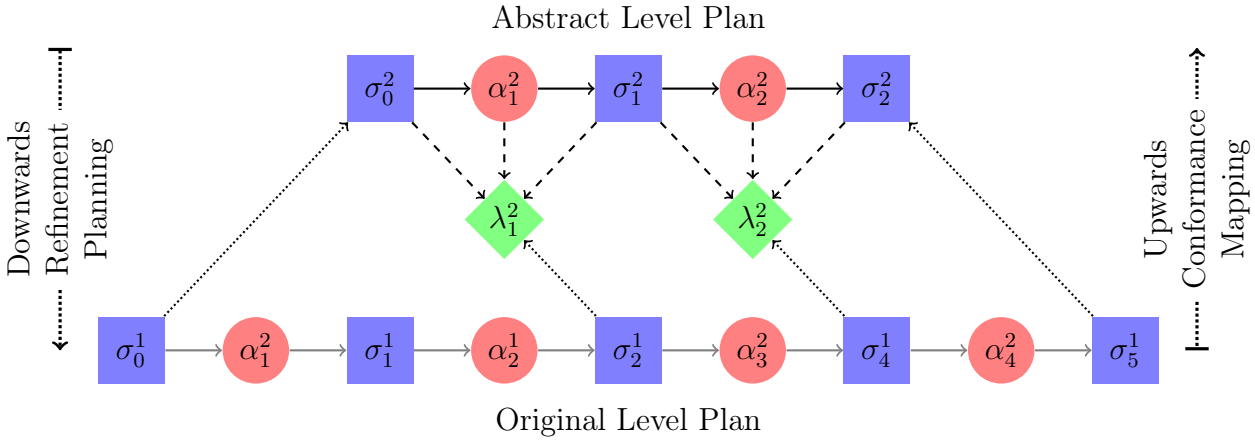


Figure 3.7: Example Depicting the Conformance Refinement of some Abstract plan.

3.7.3 Solving Problems by Concurrent Action Planning

Concurrent action planning is a technique intended to improve planning speed and generated plan quality. When a set of actions are planned concurrently, it represents that in the model those actions where planned, no problem constraint exists that require them to be planned sequentially. Therefore, these planned actions are arbitrarily ordered, and can be executed by the robot if they are ground, or achieved in a refinement if they are abstract, in either any sequence or all simultaneously. That is, any sequential permutation of the action set would be a valid sequential action plan. If the actions are abstract, the sub-goals produced from the effects of the concurrently planned actions can then be combined into a single stage. The individual sub-goal literals don't have to be achieved in any particular order, as they would if they were in separate sub-goal stages, nor should they have to be achieved simultaneously.

The potential benefits of concurrent action planning are two-fold;

- *Compression of Plan Lengths*: Concurrent action plans should be shorter than sequential plans, because they compress sub-sequences of arbitrarily ordered actions into a single set. This is beneficial because planning time grows exponentially with plan length (Jiang et al., 2019). However, there may be an overhead introduced by the increased complexity of the necessary concurrency constraints presented below.

- *Path of Least Commitment*: Concurrent action planning takes a path of least commitment in refinements, because it doesn't order actions, and therefore individual sub-goal literals into separate stages, until this is absolutely necessary to obtain a valid plan within a given model. This is important, because abstract models remove problem constraints by design. In contrast, in sequential action planning, the planner always has to choose one sequential action plan to return. If there are no problem constraints that require a strict order over planning a set of actions in an abstract model, the planner must therefore non-deterministically choose some random ordering over those

actions. This happens despite the order can have a impact on the quality of the plan's refinement. This is because stricter constraints may exist in the original model that can mean that one specific order produces the best quality plan. This is particularly important, since HCR planning cannot currently backtrack. The downside is that the conformance constraint will potentially not be as effective at restricting the search space. This is because it is the more flexible and therefore likely less restrictive.

Determining whether actions are arbitrarily ordered and can be planned concurrently is not a trivial reasoning problem. This is why so few existing planners support it. Consider a blocks world example where a robot plans to move a block A from the top of a tower to the table, and concurrently plans to move some other block B on top of A. This results in a valid state, where A is on the table and B is atop A. However, the order of executing these actions is not arbitrary. If B is moved on top of A first, it is not possible to move A at all. The planner must be able to understand that a precondition for moving block A is that no other block is on its top. Therefore, the planner cannot concurrently plan an action that has the effect of putting a block on top of A, as this disables the action of moving A.

To enable concurrent action planning for HCR, this thesis therefore contributes a set of general concurrency constraints that allow the planner to determine if the preconditions and effects of a sequence of actions makes them arbitrarily ordered:

1. Multiple concurrently planned actions cannot lead to an invalid state,
2. Multiple concurrently planned actions cannot affect the same fluent,
3. A set of concurrently planned actions must be sequentially plannable in any permutation. This requires than the effects of an action cannot unachieve the precondition of any other action planned concurrently on the same step. More specifically the effect of an action must not cause a fluent to take a value that violates the precondition of another action (the effect cannot falsify a positive or "truify" a negative precondition).

3.8 Hierarchical Plans

The following formally defines the structure of HCR hierarchical plans.

Definition 3.8.1 (Hierarchical Plan) A hierarchical plan is a function, that maps planning level l and problem number $\varkappa$, to a monolevel plan. The monolevel plan mapped to progresses the horizontal axis of online planning along the sub-goal sequence as attained by solving problem number $\varkappa$, at level l on the vertical axis. This represents how the plan evolves during the process of hierarchical planning. A hierarchical plan is formally denoted as follows:

$$\eta^{tl} : l \in [1, tl], \varkappa \in \mathbb{N}_{>0} \rightarrow \langle \pi^{l}_{j,k}, \gamma^{l}_{\varphi,\vartheta}, \rho \rangle \tag{3.71}$$

The function maps abstraction level l and problem number $\varkappa$ to: a monolevel plan $\pi^{l}_{j,k}$ which solved problem $\varkappa$; a conformance mapping $\gamma^{l}_{\varphi,\vartheta}$ linking the monolevel plan to the sub-goal stage range of the abstract (partial) plan which it refines; and a boolean $\rho \in \{\top, \bot\}$ defining whether the monolevel plan is final. The partial-problem template function $\Delta(l, \varkappa) = (\varphi, \vartheta)$ is intuitively linked to the sub-goal stage range of the conformance mapping $\gamma^{l}_{\varphi,\vartheta}$.

The function Δ is total on the abstraction level range (for all levels is the function defined), since the size of the abstraction hierarchy is pre-defined prior to planning. However, it is partial on the problem number range (only for a contiguous sub-sequence of the index set $\mathbb{N}_{>0}$ is the function defined), since the number of partial-problems is not fixed.

In the planning algorithms, the function $complete : \mathbb{N} \rightarrow \{\top, \bot\}$ such that $l \mapsto \rho$ is used as an alias for $\rho[\eta^{tl}(l, problems(l))]$, and function $problems : \mathbb{N} \rightarrow \mathbb{N}$ such that $l \mapsto \varkappa$ records the current number of partial-problems that have been solved at the given level.

The levels of a hierarchical plan must be completed in descending order, i.e. in the sequence $(l)^{1}_{l=tl}$, as by definition an abstract plan must be complete, before its complete refinement can be generated, as the entire sub-goal stage sequence is needed to do so.

The conformance mapping defines the links between abstract plans and their conformance refinements. It is a downwards one-to-one mapping, from sequence index $\varrho \in [1, \vartheta]$ of a sub-goal stage produced from an action set at abstract level $l+1$, to the time step $i \in [1, k]$ of the matching-child action set that achieved it at original level l:

$$\begin{aligned} \text{A partial function:} \quad & \gamma^{l}_{\vartheta,k} : [1, \vartheta] \subseteq \mathbb{N}_{>0} \rightarrow [1, k] \subseteq \mathbb{N}_{>0} : k \geq \vartheta \\ \text{Such that:} \quad & \varrho \in [1, \vartheta] \mapsto i \in [\varrho, k] \\ & \varrho \notin [1, \vartheta] \mapsto null \end{aligned} \tag{3.72}$$

The function literal $\gamma^{l}_{\vartheta,k}(\varrho) = i$ can therefore be read as; the plan time step i at level l, upon which the sub-goal stage at sequence index ϱ was achieved. Where intuitively, for any $\varrho \mapsto i$ then $i \geq \varrho$, since at least one action is required to achieve each sub-goal stage.

Hierarchical plans are built dynamically during the process of hierarchical planning. New levels are added to a hierarchical plan over the vertical axis of planning, whilst in online planning, partial-plans are extended across the horizontal axis of planning. This requires a hierarchical plan to be an extensible structure that can be constructed incrementally. The following defines how a hierarchical plan is extended when a monolevel problem is solved.

Definition 3.8.2 (Construction of Hierarchical Plan) If $\zeta^{pl}_{j,k}$ is the answer set that solves a conformance refinement planning problem $MP^{pl,\asymp}_{j,\varphi,\vartheta}$ and η^{tl} is a hierarchical plan, then the plan $\pi^{pl}_{j,k}$ which solves the monolevel planning problem is extracted as follows:

$$\pi^{pl}_{j,k} = \{\tau^{pl}_{i} \mid i \in [j, k]\} : \epsilon(\tau^{pl}_{i}) \in \zeta^{pl}_{j,k} \tag{3.73}$$

The conformance mapping is extracted as follows:

$$\forall \varrho \in [\varphi, \vartheta].\ \gamma^{l}_{\vartheta,k}(\varrho) = \begin{cases} i \in [j, k] : sgls\_ach\_at(l, \varrho, i) \in \zeta^{pl}_{j,k} & \text{if } \varrho \in [\varphi, \vartheta] \\ \phi^{l}_{\varphi-1,j-1}(\varrho) & \text{if } \varrho \in [1, \vartheta - 1] \end{cases} \tag{3.74}$$

And the hierarchical plan is extended as follows:

$$\pi^{pl}_{0,k}[\eta^{tl}(pl)] = \begin{cases} \pi^{pl}_{0,j}[\eta^{tl}(pl, \varkappa - 1)] \cup \pi^{pl}_{j,k} & \textbf{if } \varkappa \neq 1, j > 0 \\ \pi^{pl}_{j,k} & \textbf{if } \varkappa = 1, j = 0 \end{cases} \tag{3.75}$$

$$\rho^{pl}[\eta^{tl}(pl)] = \rho^{pl+1}[\eta^{tl}(pl+1)] = \top \wedge \gamma(pl, \varkappa) = len(pl+1) \tag{3.76}$$

If the plan at the previous level is complete, and the plan achieves the last sub-goal stage from the previous level, then the plan is final and extends that level to be complete.

Plan Expansion Factor and Expansion Balance

Hierarchical plans have two properties for measuring the complexity of HCR planning and quality of refined plans. These are the plan and problem expansion factor and balance.

The monolevel plan expansion factor θ is the factor by which a complete (concatenated) refined monolevel plan increases in length over the complete (concatenated) abstract plan it refines. It is a unit-less ratio of lengths, with a lower-bound of 1.0 (no expansion has happened), and no upper bound. Expansion factors are important, because they tell us how much progress is made by refinement in a particular domain model. If the expansion at a given level is too low, then this may indicate a model is not abstract enough, and insufficient progress is made during plan expansion for the model to be effective. If the expansion factor is too high, this may indicate that a model is too abstract, and there may be too few sub-goal stages to restrict the search space or promote problem divisions at the next level.

The overall hierarchical plan expansion factor Θ_h of a hierarchical plan is the ratio of the top-level classical plan's length to the ground-level plan's length. It is given by:

$$\Theta_h = \frac{clen(1)}{clen(tl)} \tag{3.77}$$

The smoothed monolevel plan expansion factor Θ_s of a hierarchical plan is the most balanced monolevel plan expansion factor. Whereby, it is the target value of θ such that achieving $\theta = \Theta_s$ would spread the expansion of plans evenly over the hierarchy. It is given by:

$$\Theta_s = \sqrt[tl-1]{\Theta_h} \tag{3.78}$$

This equation holds because the length of monolevel plans can increase exponentially in the depth of the hierarchy, and therefore require only a linear increase in hierarchy depth to scale to problems with exponentially longer plan lengths, such that:

$$mlen(l) = clen(tl) \times \Theta_s^{tl-l} \tag{3.79}$$

Where $mlen$ is a function of the form $mlen : \mathbb{N}_{>0} \rightarrow \mathbb{N}_{>0}$ such that $l \mapsto k$ defining the maximum possible plan length achievable at level l given Θ_s.

The conjecture is that, because planning complexity is exponential in the plan length, spreading the expansion of plans evenly over the hierarchy is desirable. This is because it implies that the complexity of refinement planning is also evenly spread, and that the constraints of a problem are therefore evenly broken down over the hierarchy. This spread is measured by the hierarchical plan expansion balance, in two ways:

- The balance deviation B_d is the coefficient of deviation (ratio of the standard deviation to the mean) of the monolevel plan expansion factor over the entire hierarchy,
- The balance error B_e is the normalised mean absolute error between the monolevel plan expansion factor and the smoothed plan expansion factor over the entire hierarchy.

The two balances B_d and B_e are unit-less, with lower-bounds of 0.0 indicating perfect balance (all plans expand equally), and no upper-bound. Both are normalised measures and can be used to compare between different planning domains and problems. The balance deviation assumes that plan lengths are normally distributed whereas the balance error does not.

Representing Hierarchical Plans

Hierarchical plans generated by HCR planning are graphical structures, and as such can be represented diagrammatically. However, due to the multiple monolevel plans involved and complex conformance mappings between them, it can be difficult to do so compactly.

To keep the representation minimal, the states are left implicit, and the abstraction mappings are not included. Only the following are displayed: actions; their effects if they either form a sub-goal to be passed to the next level, or achieve a sub-goal from the previous level; the links between actions and their effects; and the conformance mappings. Further, although a hierarchical plan is a vertical structure, where more abstract plans exist at the higher levels, and more concrete plans at the lower, when they are depicted diagrammatically, they are drawn horizontally, where more abstract plans are written to the left, and more concrete plans to the right, "nested" amoungst the abstract actions, like a tiered list.

The diagram in Figure 3.8 is a minimal example of a hierarchical plan, depicting the independent partial-refinement of a single abstract locomotive plan over three abstraction levels[36]. On the left-hand side of the diagram, the cyan nodes are actions, the solid blue downwards arrows indicate the sequence of actions in the same monolevel plan, whereas the blue upwards dashed arrows link actions of a refined sub-plan to the abstract action from the previous level which they refine. On the right-hand side, the magenta nodes are action effects, and the dashed red arrows from left-to-right link actions to their effects. If these effects are abstract then they are sub-goals to be achieved by the refined plans. The solid red upwards arrows indicate the conformance mapping between abstract and refined plans, by linking low-level action effects to higher-level action effects which they achieve when mapped to the state abstraction. On the far right, a link over all levels always exists, mapping up from the effects of the last ground-level action. This is because the last action of a refined plan must achieve the same effects as the last action from the abstract plan.

[36]Note that although this hierarchical plan could be represented as a refinement tree, hierarchical plans are strictly more general that refinement trees, as they can have an entire monolevel plan at the top-level.

The nature of the plan itself is as follows. A robot r starts in the starting room rt_r in cell 1 $(rt\_r, 1)$ and its goal is to be in the store room st_r. At the top-level of 3, the robot can plan just one abstract action which immediately moves it to the store room, as the relaxed problem ignores the preconditions requiring movement between only connected locations. At the next level of 2, this precondition is reintroduced, and therefore to reach the store room (and achieve the same effects as the top-level abstract action) it must first move through the hallway hw, therefore expanding the plan to two actions. At the ground-level, the robot must consider which cells of the rooms to move through; it minimally uniquely achieves the effect of the first abstract action from the previous level by moving to cell 1 of the hallway, and then achieves the effect of the second action by moving to cell 1 of the store room. This expands the plan to four actions. Unfortunately, this hierarchical plan, which contains only three levels and a ground-plan length of four actions, is already almost too large to present on a page, indicating the difficulty of graphically presenting these structures.

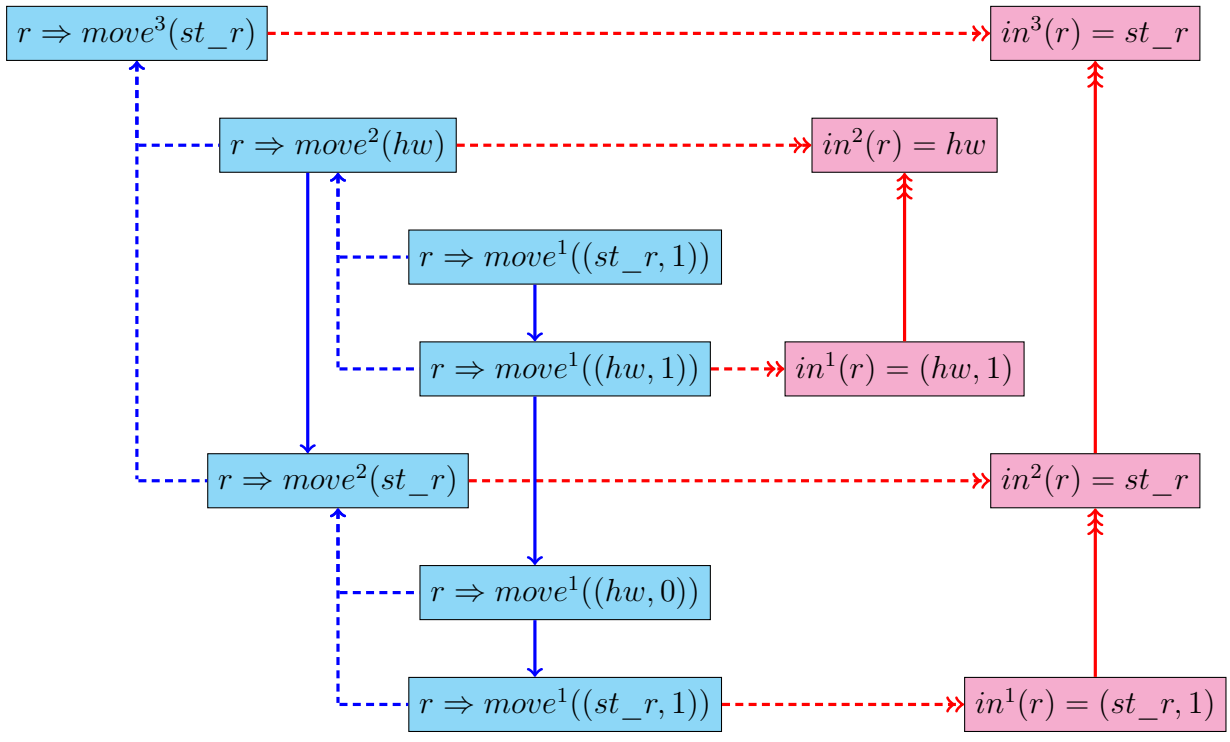


Figure 3.8: An Example Hierarchical Plan. Depicted is a three-level hierarchical locomotive plan (discrete path plan) in the BWP domain (see Figure 3.1 for the complete structure of the BWP domain and 3.2 for the hierarchy structure used in BWP problems).

Chapter Four

Algorithms and Decision Making Systems for HCR Planning

This chapter presents the online planning systems, HCR planning algorithms, and the operational modules of ASH that define the domain-independent rules for HCR planning.

4.1 Online Planning Systems

Problem division strategies and online planning methods are decision making systems used in online planning. They together decide how the algorithm moves along the vertical and horizontal axes of planning. In doing so, they construct and traverse the problem division diagram, a structure that represents how a hierarchical problem was divided and solved. Division strategies decide the number and size of partial-problems that divide a combined refinement problem, and online methods decide when to divide and solve them.

4.1.1 Problem Division Diagrams

A problem division diagram is a branching structure, containing within it a tree, whose nodes contain monolevel planning problems and the monolevel plans that solve them, and whose arcs link abstract planning problems and plans to their refinement planning problems and refined plans. Traversal links then denote the order in which problems are solved and plans

generated. This allows one to trace and observe the progression of online planning across the many monolevel planning problems that may be involved in a hierarchical problem. When a node is visited, its planning problem is solved, when it is expanded, the refinement problem of its abstract plan is divided. Multiple abstract nodes can be joined before expanding them, which constitutes the concatenation of their abstract plans, before problem division occurs over their combined refinement problem. When a traversal link descends downwards from a node, it must first be expanded to create at least one node to visit at the next level.

There are two important properties used to measure the balance of problem division diagrams: the partial-problem size balance and partial-plan expansion balance. The partial-problem size balance can be measured in two ways: the problem balance deviation μ_d is the coefficient of deviation over the size of all partial-problems that divide the complete combined refinement planning problem; and the problem balance error μ_e is the normalised mean absolute error between each partial-problem's actual size and their "perfectly" balanced size normalised by the complete problem's size. Where the perfectly balanced size is the complete combined refinement problem's size divided by the number of partial-problems. The partial-plan expansion balance can also be measured in two ways: the plan balance deviation β_d is the coefficient of deviation over the expansion factor of all partial-plans that where concatenated to form a complete monolevel plan; and the plan balance error β_e is the normalised mean absolute error between each partial-plan's length and their "perfectly" balanced length normalised by the complete plan's length. Where the perfectly balanced length is the complete concatenated plan's length divided by the number of partial-plans.

4.1.2 Problem Division Strategies

A problem division strategy commits divisions over a combined refinement problem prior to refinement planning. They are called by the hierarchical planning algorithm immediately after an abstract plan is refined (if requested by the online method), and once committed,

any divisions made over its refinement problem cannot be modified or removed. Divisions create templates for partial refinement problems through the Δ function. This is because they define only the sub-goal stage range of the partial-problems, but not their start states. See Section 3.6.4 for the definition of partial-problems and the Δ templating function.

There are three division strategies presented in this thesis. The Basic strategy makes templates for a desired number of partial-problems for each combined refinement problem that is divided, regardless of their size. The Hasty and Steady strategies dynamically decide how many partial-problems to make templates for based on a desired problem size bound.

Assumptions of Homogeneous Strategies

All three strategies in this thesis are homogeneous. Homogeneous division strategies are uninformed and decide upon problem divisions based only on the length of an abstract plan. They do not consider the actions of a plan, the constraints under which they were planned in the abstract model, or the effects of the more elaborate constraints of the original model on the abstract plan's refinement. The following assumptions must therefore be applied:

1. *Balanced Refinements Assumption:* The minimum length plan that solves a refinement planning problem increases linearly with its size, whereby all abstract action sets refine to equal length sub-plans (all refinement trees are balanced). A sequence of homogeneous size partial refinement problems are therefore all solved by homogeneous length partial-plans. If planning time increases exponentially with plan length, then planning time also increases exponentially with problem size and plan expansion factor[1].

2. *Uniform Probability of Dependency Assumption:* The quantity of divisions made over a combined refinement problem is directly proportional to the probability that at least one pair of mutually dependent problems will exist in the resulting sequence. In other

[1]The requirement for balanced trees would restrict this to integer expansion factors (which are unlikely).

words, placing a division between any given pair of sub-goal stages is equally as likely to reveal a dependency between them, prevent interleaving and reducing plan quality. Under this assumption, the position of divisions has no impact on the probability of causing a dependency, only the number of divisions and sizes of the problems.

Because constraints removed by an abstract model will either not to affect all actions, or not to affect all actions equally, neither of these assumptions is likely to hold in practice. These assumptions are challenged in the experimental studies in Chapter 5 and plans for future work on informed strategies that relax these assumptions are given in Section 6.2.

Making Homogeneous Partial-Problems

Since division of integers does not always produce an integer, it is not always possible to make a sequence equal sized partial problems over an abstract plan. As a result, some flexibility with the definition of a homogeneous sized sequence of partial problems is necessary. Therefore, a sequence of partial problems is considered homogeneous if their sizes differ by no more than 1 sub-goal stage. Homogeneous divisions can be made via the following method:

1. To attempt to divide an abstract plan of length k into (at most) n partial problems,
2. Template $m = min(k, n)$ partial problems from $d = m - 1$ division points such that;
   (a) $m_{small} = m - (k \mod m)$ "small" problems of size $s_{small} = \lfloor k/m \rfloor$,
   (b) $m_{large} = (k \mod m)$ "large" problems of size $s_{large} = s_{small} + 1 = \lceil k/m \rceil$,

   This is guaranteed to produce an m length sequence of homogeneous partial problems of size $s \in \{x, x+1\} : x \in \mathbb{N}_{>0}$,
3. Bias the order of the problem sequence such that the small problems occur first and the large follow (in an attempt to reduce execution latency and early wait times).

The Basic Strategy

The Basic strategy takes a combined refinement problem of an abstract plan of length k, and divides it into an m length sequence of partial problems, each of size as close to $s \approx k/m$ as possible. The input parameters to the Basic strategy are given as a vector of integers. Each integer defines the number of divisions to make over combined refinement problems of abstract plans, for each non-ground abstraction level in the hierarchy, in descending order.

An important point is that the basic strategy is not guaranteed to produce a homogeneous sequence of partial problems across the entire sequence at any particular abstraction level. It is only guaranteed to produce a homogeneous sequence over the combined refinement problem of a particular abstract plan. If that abstract plan is not a complete plan, then the strategy is making sub-divisions of a problem. Resultantly, it cannot guarantee partial-problems of that abstract plan will also be homogeneous to other partial-problems of different abstract plans from the previous adjacent abstraction level. A trade-off is likely needed: more divisions may in theory reduce overall problem complexity, but may also reduce concatenated plan quality. Reduced plan quality is due to a proliferation of errors, caused by dependencies induced by the increased number of non-interleaving partial problems.

The Basic strategy always produces balanced problem division diagrams, in the sense that each parent node has the same number of children (given that the given number of problems to template is smaller than the length of the abstract plans). However, because there is no guarantee that refinement trees will be balanced, there is no guarantee that there will be balance over the length of partial-plans that solve the partial-problems in the nodes at any given level. When those nodes are expanded independently, there is therefore also no guarantee that there will be balance over the size of the children of different parent nodes. This means that a sequence of homogeneous sized partial-problems, rarely refines to a sequence of homogeneous length partial-plans. Due to the way the Basic strategy and online planning methods work, if the partial-plans are abstract, the divisions over their combined

refinement problems will be made independently, all dividing into a equal number of partial-problems despite the variance in their size (a small combined problem is divided as many times as a large problem). Resultantly, an accumulating effect can occur, which causes the variance in the size of partial-problems and plans to increase with the depth of the problem division diagram. Below the second-to-top-level, all the partial problems do not share a common parent node. If the refinements of the highest level of partial problems are not balanced partial-plans, then the next level of partial-problems will not be balanced either.

Size Bound Strategies: Hasty and Steady

The size bound based strategies Hasty and Steady make dynamic length sequences of homogeneous partial-problems based on a desired problem size bound. They take a combined refinement problem of an abstract plan of length k, and divides it into an arbitrary number of partial-problems each of size as close to the desired bound b as possible. Whereby, larger combined refinement problems are divided into more partial-problems, and vice versa for smaller problems. The input parameters are given as a vector of integers. Each integer defines the desired size bound, as the quantity of sub-goal stages to be allocated to all partial-problems, for each non-ground abstraction level in the hierarchy, in descending order.

This more flexible but simple approach is intended to reduce the variance in the size of partial-problems across the problem sequence at a given level in a problem division diagram. This is because the size of the combined refinement problem being divided does not matter to the size-bound based strategies, given that the size of the problem is bigger than the bound. However, note that Hasty and Steady still assume that the size of a refinement problem is an accurate measure of its complexity. The conjecture is however, that by keeping partial-problems homogeneous across the entire problem sequence at a given level, may avoid the issue with the Basic strategy, whereby the effect of unbalanced refinements on the variance of the length of partial-plans is less likely to accumulate down the hierarchy.

Hasty and Steady only add one extra initial step to the decision making process over Basic, in which they decide dynamically on the number of divisions to make over a combined refinement problem based on a size bound. The difference between Hasty and Steady, is how they handle the case where size of the combined refinement problem is not divisible by the size bound. For Hasty, the bound is a limit on the maximum size of partial-problems, since problems cannot be larger than the bound. For Steady, this is a limit on the minimum size, since problems cannot be smaller than the bound. The input for both is simply a vector of positive non-zero integers defining the size bound to use for each abstraction level. Note that the bound value at the lower levels of the hierarchy must never be larger than the bound value at a higher level[2]). The strategies therefore function as follows:

- Attempt to divide the combined refinement problem of an abstract plan of length k into a quantity m of homogeneous partial-problems proportional to n, such that:
  - $n = max(1, k/b^l)$ where b^l is the size bound at level l.
- Since n is not always an integer, it is of course first necessary to round to obtain a valid number of problems (this is the only functional difference between Hasty and Steady):
  - Hasty: $m = \lceil n \rceil$ such that $m \in \{n, n+1\}$,
  - Steady: $m = \lfloor n \rfloor$ such that $m \in \{n-1, n\}$.
- Invoke the Basic strategy to template the m homogeneous partial-problems.

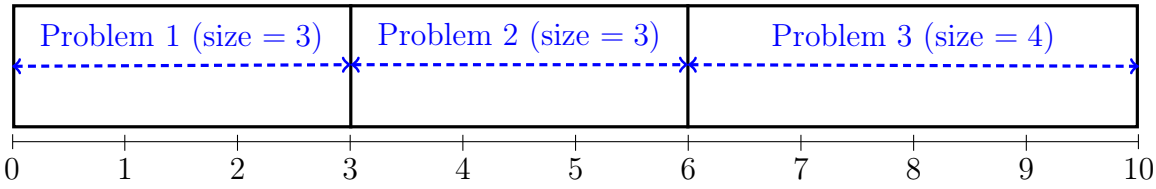
The intuition behind the design of these strategies is that: a) Hasty favours planning speed by biasing problems to be smaller and therefore faster to solve; b) Steady favours plan quality by biasing problems to be larger and therefore avoiding dependencies (by allowing more capacity for interleaving). Whilst this is a notable functional difference, the most important factor in the performance of the strategies is likely to be the selection of the bound value itself. This will be therefore tested in the experimental studies.

[2]Since expansion factors are always > 1.0, the strategies cannot make a problem smaller than the bound, unless the top-level plan is shorter than the bound, or lower-level bounds are larger than higher-level bounds.

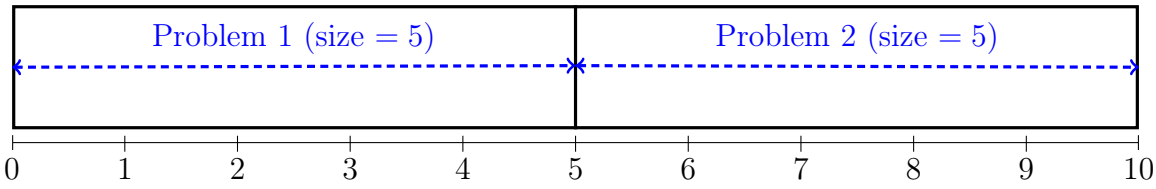
Problem Division Scenarios

Problem division scenarios are a simple structure that represent how the combined refinement problem of a single abstract plan is divided into a sequence of partial refinement problems. They are essentially a detailed view on a single node in a problem division diagram.

Figure 4.1 contains example division scenarios that could be produced from Hasty and Steady. The time steps and resulting sub-goal stage indices are shown on the scenarios' horizontal axes. The index of the left division point is exclusive, whilst the right division point is inclusive, of the sub-goal stage index range of the templated partial problems[3]. Both scenarios show some abstract plan of length 10, being divided based on a size bound of 4. Since $10/4 = 2.5$, Hasty makes 3 heterogeneous partial-problems, and Steady makes 2 homogeneous partial-problems. Whilst the functional difference between these strategies is small, they do produce significant differences in how they produce partial-problems.



(a) Scenario for Hasty: Partial-problems are no bigger than the bound, causing a larger quantity of smaller sized problems.



(b) Scenario for Steady: Partial-problems are at least as big as the bound, causing a smaller quantity of larger sized problems.

Figure 4.1: Homogeneous Division Scenarios for Hasty and Steady over Abstract Plans of Length 10 and Size Bound of 4.

[3]This is because, the divided abstract plan starts from the initial state at step 0, but the first sub-goal stage index is 1, since this is the first step upon which an abstract action can be planned. All non-initial partial problems start from the state in which the last sub-goal stage from the previous problem was achieved.

4.1.3 Online Planning Methods

Online planning methods decide how the problem division diagram is traversed and therefore when problems are solved and combined refinement problems divided. They control how often ground-level partial-plans are yielded to the robot, resultantly impacting execution latency and yield times of partial-plans. These are general decision making systems and few constraints exist on which order problems are solved or abstract plans divided. The only fundamental constraint, is that problems must be templated, solved, and divided in descending order over the vertical axis and in ascending order over the horizontal axis.

The most obvious online planning methods preform a depth-first or breadth-first traversal of a problem division diagram. Both cases are well defined and intuitive, as there is only one possible path that can be taken through any given problem division diagram. The following describes both these methods and one further method that combines them.

Ground-First

The ground-first method performs a depth-first traversal of the problem division diagram. It solves only the first available partial-problem at a given abstraction level before moving to the next level. The method therefore tells the planner to solve and immediately expand only the lowest and left-most problem node in the diagram. It then only returns to higher levels when all currently available lower refinement problem nodes have been solved.

The method first solves only the initial partial-problem at each level, to propagate directly down to the ground-level and yielding the first executable ground-level partial-plan as quickly as possible, to minimise execution latency. It then subsequently plans downwards from the lowest level at which an unsolved refinement problem exists, to solve and yield the next ground-level partial-problem and plan as quickly as possible, to minimise non-initial wait and minimum execution times. See Section 5.1.2 for details on these timing criteria.

In ground-first, as soon as a problem is solved and an abstract plan is generated, its combined refinement problem is divided and at least part of the plan is refined immediately. Combined refinement problems of abstract plans are therefore divided independently from all other abstract plans at the same level. This may exaggerate the problems of accumulating partial-problem imbalance of the Basic problem division strategy discussed previously.

Complete-First

The complete-first method perform a breath-first traversal of the problem division diagram, solving all partial-problems at a given level before moving to the next. The planner always solves the highest unsolved problem in the diagram. Every problem at a given level is solved before concatenating their partial-plans and then dividing the concatenated plan.

This successively completes each level in descending order, until the ground is reached and completed. The only planning increments that yield executable ground actions to the robot(s) are those that start at the ground-level, this only happens after all abstract levels have been completed. The execution latency time is likely to be higher than ground-first because all abstract levels have to be completed before planning at the ground-level. However, once the ground-level is reached, all planning increments solve just one ground-level partial-problem, and yield an executable partial-plan, therefore likely obtaining lower non-initial wait and minimum execution times than ground-first after the initial latency.

Unlike the ground-first method, by always concatenating the entire abstract plan at any given level before making problem divisions, any difference in abstract plan lengths is not propagated to the next level, and therefore cannot accumulate. Since the expectation is that the abstract levels will be fast to solve, and account for a relatively small proportion of the total planning time compared to the ground-level planning time, the conjecture is that this benefit may come at the cost of only minorly increased execution latency time.

Hybrid

The hybrid online planning method uses a combination of ground-first and complete-first. It first solves only the initial partial-problems to propagate directly down to the ground-level as fast as possible, and then successively completes each level in descending order.

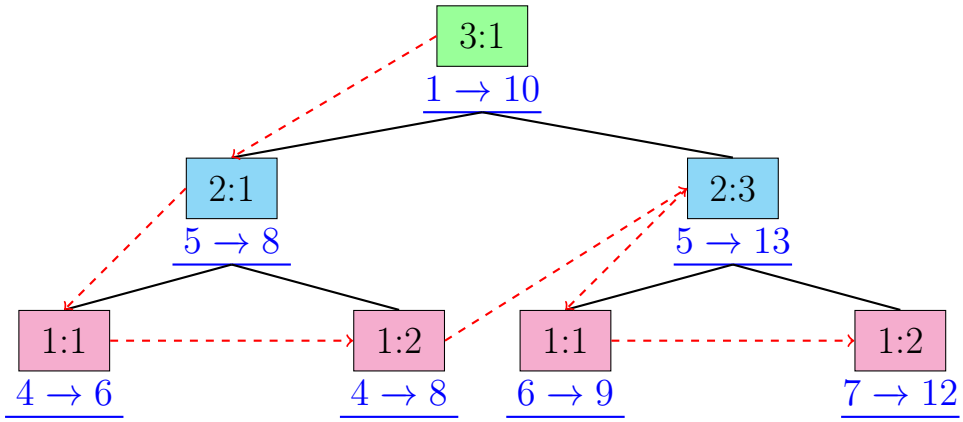
The idea is to obtain both the low execution latency of ground-first and the more regular problem division diagram structure of complete-first. Intuitively, if the execution time of the initial ground-level partial-plan is greater than the wait time for the second partial-plan, then the robot will not experience any downtime after execution, therefore gaining the benefit of complete-first without losing any of the speed of ground-first.

Online Method Examples

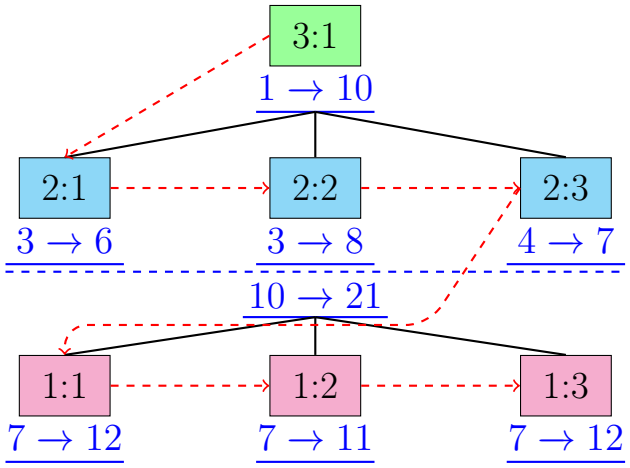
The following Figures 4.2 and 4.3 contain example problem division diagrams[4] for the Basic and Steady problem division strategies. The partial problem and plan size balance score is one minus the mean over all non-top-levels of the coefficient of deviation of the problem size or plan length, and is calculated over all problems/plans at a given level. A score of 1.0 indicates perfect balance, and values lower than 1.0 indicate unbalance. The diagrams for the Basic strategy show that the complete-first method can give better partial-problem and plan balancing that ground-first[5] and hybrid, because the whole abstract plan at level 2 is divided simultaneously. In contrast, the diagrams for the Steady strategy show that good balance can be achieved by all methods, due to the requirement to divide based on the problem size bound. This also causes the Steady strategy to produce an irregular number of partial-problems when it expands a node, as compared to the rigid structure of Basic. For both strategies, there may be small trade-offs in terms of execution latency and average wait times of partial-plans, based on how the method traverses the tree and expands nodes.

[4]These diagrams are examples and not indicative of actual experimental results for a real problem.

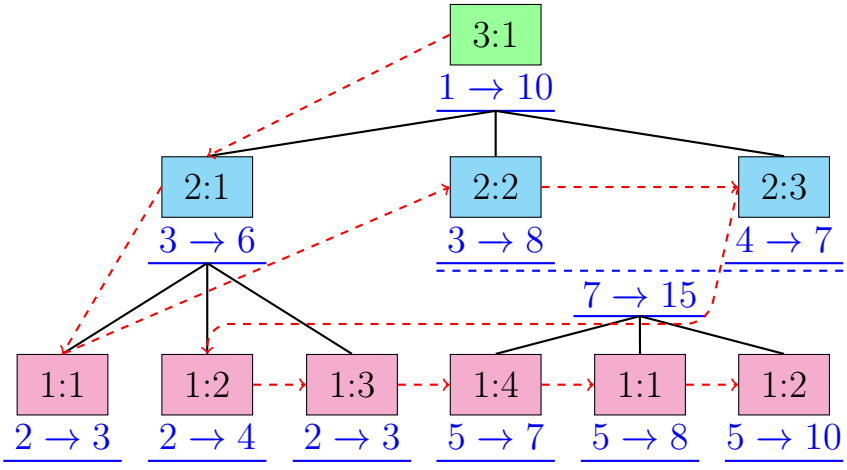
[5]A lower problem-bound of 2 is used for ground-first for Basic to keep the size of the diagram manageable.



(a) Ground-first Problem Division Diagram for Basic Strategy with Problem-bound of 2. The partial-problem balance score is ≈ 0.86 and the partial-plan balance score is ≈ 0.69.

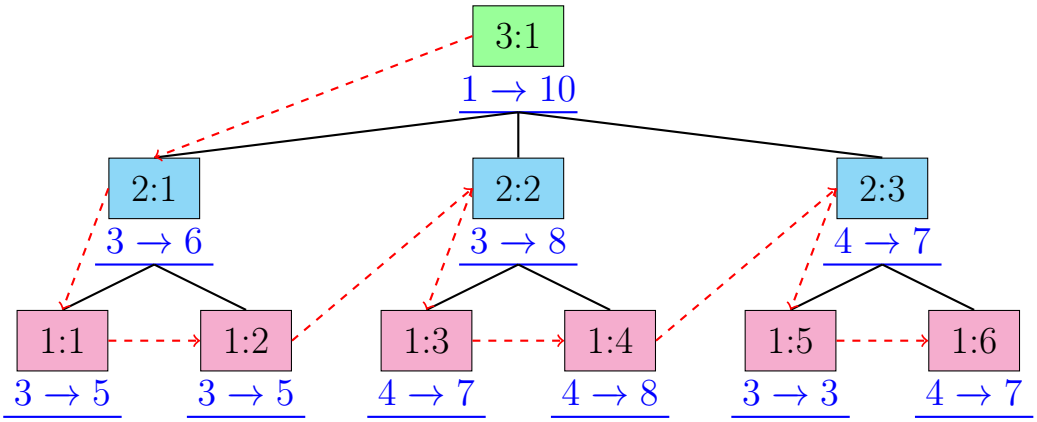


(b) Complete-first Problem Division Diagram for Basic Strategy with Problem-bound of 3. The partial-problem balance score is ≈ 0.91 and the partial-plan balance score is ≈ 0.90.

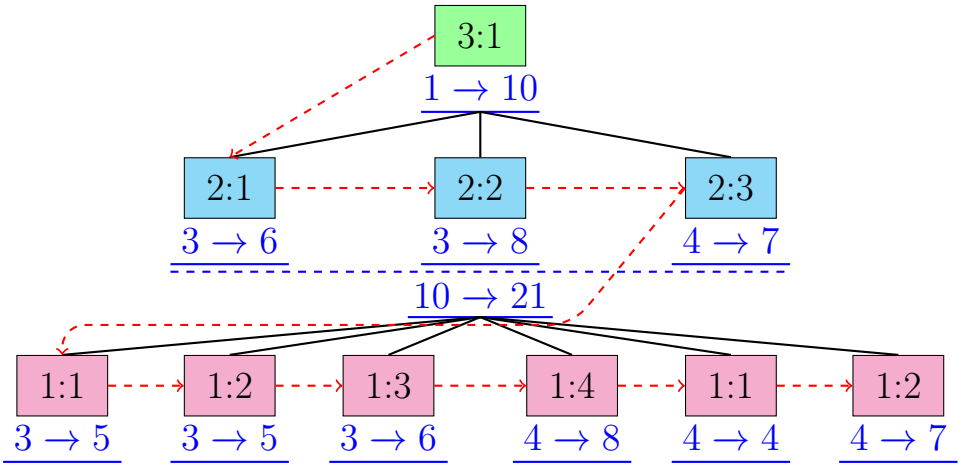


(c) Hybrid Problem Division Diagram for Basic Strategy with Problem-bound of 3. The partial-problem balance score is ≈ 0.68 and the partial-plan balance score is ≈ 0.68.

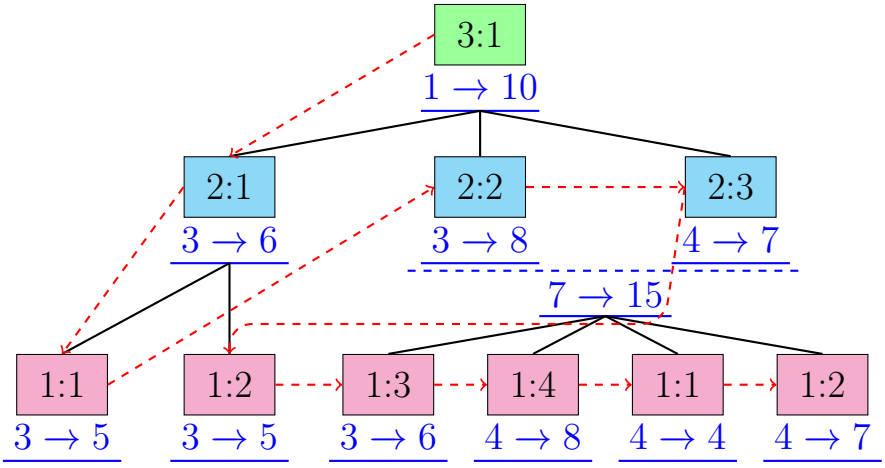
Figure 4.2: Problem Division Diagrams using the Basic Problem Division Strategy.



(a) Ground-first Problem Division Diagram for Steady Strategy with Size-bound of 3. The partial-problem balance score is ≈ 0.84 and the partial-plan balance score is ≈ 0.77.



(b) Complete-first Problem Division Diagram for Steady Strategy with Size-bound of 3. The partial-problem balance score is ≈ 0.84 and the partial-plan balance score is ≈ 0.80.



(c) Hybrid Problem Division Diagram for Steady Strategy with Size-bound of 3. The partial-problem balance score is ≈ 0.84 and the partial-plan balance score is ≈ 0.80.

Figure 4.3: Problem Division Diagram using the Steady Problem Division Strategy.

4.2 Additional Techniques for Online Planning

This section briefly describes three additional techniques proposed for potentially improving the performance and generated plan quality of online planning.

4.2.1 Final-Goal Preemptive Achievement

Final-Goal preemptive achievement is a heuristic optimisation method, that biases the planner towards attempting to preemptively achieving final-goal literals during non-final partial-problems. This is beneficial when refining from a model that uses a state abstraction, such as a condensed model. This is because when sub-goal stages are generated from abstract actions that are planned in a state abstraction and which achieve abstract final-goal literals, those sub-goal stages do not necessarily require the correct achievement of the corresponding original level version(s) of the same final-goal literals during refinement planning. The effect is very similar to the cause of trailing plans described in Section 3.7.2, which occurs from the fact that there are many original level states that achieve the same effects of abstract actions encoded in a sub-goal stage, but many of those states will not achieve the more specific original level final-goal. For example, placing a block on the table in the condensed model of the BWP domain creates the effect $on_i^2(block_n) = table$. This effect can be achieved by the $talos \Rightarrow put_i^2(block, table\_left)$ or $talos \Rightarrow put_i^2(block, table\_right)$ actions, since both map to the abstract effect. Since the side of the table upon which the blocks must be stacked does matter towards achieving the final-goal in the original model, in any non-final partial problem, the planner needs some method to determine which original level action is preferable to achieve the final-goal. Without this, early partial-problems that do not have context of the final-goal, can have very poor quality solutions relative to the complete problem.

Final-goal pre-emptive achievement defines a preference relation over original level action selections towards achieving final-goal literals, which is reduced to an optimisation

problem. The optimisation forces the ASP solver to find a plan containing the maximum possible number of actions whose effects pre-emptively achieve positive final-goal literals. Informally, the preference requests the planner to select actions whose effects achieve positive final-goal literals in non-final partial-problems, if doing so does not reduce the partial-plan's quality. It cannot force the planner to generate a longer than minimal plan in order the pre-emptively achieve a final-goal literal. For example, if there are two actions that can be used in a partial-plan that still achieve all sub-goal stages, but one achieves a final-goal literal and the others do not, then the choice between the actions can be considered arbitrary in the local context of the partial-problem itself. The one that achieves the final-goal literal is however considered preferable in the global context, because it is more likely to leave the resulting partial-plan in a state that makes it easier to achieve other final-goal literals.

4.2.2 Saved Program Groundings

In all existing literature on ASP based hierarchical planning, a new unique logic program was always created to solve each individual monolevel planning problem. In HCR planning, it is possible to avoid this, because later partial-problems at any given abstraction level simply extend earlier problems along the goal sequence. Resultantly, the grounding of an existing logic program used to solve an early partial-problem can be saved, that is held in a suspended state, and re-loaded and extended to solve the next in sequence partial-problem at the same level. This can be done by simply adding the sub-goal stage sequence related to the next problem to the program. To enforce the division between the partial-problems, the existing partial-plan at the given level is re-inserted into the program to "fix" it to the grounding, therefore preventing it from being changed. This constitutes the same effect as making a problem division by discarding the old program and creating a new one.

Saved groundings are potentially beneficial because they avoid the need to re-ground non-incremental ASP program parts, such as the class hierarchy, domain sorts, and en-

tity instance constraints. Online planning requires many monolevel problems to be solved throughout the hierarchical planning process. Each of which usually requires their own logic program to solve. Saved groundings may therefore eliminate a large amount of computational overhead. The downside is that the grounding may grow in size as it is extended along the sub-goal and partial-problem sequence. This may adversely affect search efficiency.

4.2.3 Partial-Problem Blending

Partial-problem blending is a simple technique that partly merges adjacent partial-problems, in an attempt to alleviate dependencies between sub-goal stages even in online planning. It involves including in the current partial-problem an early sub-sequence of the sub-goal stages from the next partial-problem. This allows the sub-goal stages in the current partial-problem to be achieved non-greedily with respect to the blended sub-goal stages from the next.

Partial-problem blending is enabled by passing a "blend quantity" (a natural number) to the hierarchical planning algorithm. This then captures the necessary blended sub-goal stages for each partial-problem[6]. When a partial-problem includes blended sub-goal stages, the part of the partial-plan which achieves the blended sub-goal stages is discarded. This leaves the end state of the partial-plan better prepared for the next partial-problem. The blended sub-goals are then refined again when the next partial-problem is solved, therefore being achieved in non-greedily with respect to all the sub-goal stages in that partial-problem. If there is a dependency between any of the sub-goal stages in the current partial-problem and the blended sub-goal stages, this simple method enables interleaving to eliminate those dependencies. The downside of this method is that blending can only eliminate dependencies between adjacent partial-problems. Further, blending is likely to increase computational costs due to the need to refine some sub-goal stages twice. Consequently, any improvement in plan quality is likely to come with the trade-off of increased planning time.

[6]The full details of the implementation are not described here, since the concept is simple, but the code for handling blends can be complex. The reader is advised to consult the code repository for details.

4.3 Logic Programs and Answer Sets

The following defines bespoke notation for constructing and representing ASP programs and their answer sets over an abstraction hierarchy as is necessary for HCR planning.

Definition 4.3.1 (ASP Program) A Clingo ASP program (Gebser, Kaminski, Kaufmann, Lindauer, et al., 2022b) conforming to the ASP-Core-2 language standard (Calimeri, W. Faber, et al., 2012) and represented over an abstraction hierarchy is denoted by:

$$\Pi^{pl,tl}_{j,k}(R_0, ..., R_m) \rightarrow Z^{pl}_{j,k} = \{\zeta^{pl}_{j,k,0}, ..., \zeta^{pl}_{j,k,n}\} : (m, n) \in \mathbb{N} \tag{4.1}$$

Where $(R_0, ..., R_m)$ is a vector of sets of ASP logic rules (the program rules) and the finite set $Z^{pl}_{j,k} = \{\zeta^{pl}_{j,k,0}, ..., \zeta^{pl}_{j,k,n}\}$ are the program's answer sets. The program contains the constants;

- $pl \in \mathbb{N} \cap [1, tl]$ the current planning level,
- $sl = \{pl, pl + 1\} \cap [1, tl]$ the possible state representation levels,
- $tl \in \mathbb{N}_{\geq 2}$ the abstraction hierarchy's top-level,
- $j \in \mathbb{N} \cap [0, k)$ the time step to start planning from,
- $k \in \mathbb{N}_{>j}$ the current search length (the current maximum time step).

Rules of a program and atoms of an answer set with an abstraction level parameter (always the first) will have arguments in the range $l \in [pl, tl]$, and those with a time step parameter (always the last) will have arguments in the range $i \in [j, k]$. Almost all rules/atoms have an abstraction level parameter, representing the level(s) in the hierarchy they exist at, whereas only those that relate to actions, state, and sub-goals have a step parameter.

Such a program has two parts; a base part containing static knowledge of a fixed size, and a cumulative part containing dynamic knowledge dependent on, increasing in size, and possibly changing, with search length k. When incremental solving, the base part is grounded only once, but the cumulative part is grounded and extended on every step $i \in [j, k]$.

4.4 Operational Modules of ASH

The operational modules of ASH define the domain-independent logic rules used for HCR planning and problem generation. Each module is a set of rules with a distinct functional role. The modules are then built into a single general declarative knowledge base, and by selectively grounding them, an ASP program can be set to various "operational modes" for the needs of different HCR planning algorithms. There are two types of module; static modules (part of the base program) and incremental modules (part of the cumulative program).

A module header declaration is denoted in the form:

$$\mathcal{N}(param_1 : dom_1, param_i : dom_i, ..., param_n : dom_n) \tag{4.2}$$

Where $\mathcal{N}$ is the module name, the vector $(param_1 : dom_1, param_i : dom_i, ..., param_n : dom_n)$ are the module's parameters, $param_i$ is a parameter name (a string over some fixed alphabet), and dom_i is the domain of the parameter. The module's rules may contain any parameter name as an atom or term (as they are replaced by their arguments during grounding).

A module used as part of a selective program grounding is denoted in the form:

$$\Pi^{pl}_{j,k}(\mathcal{N}(arg_1, arg_i, \dots, arg_n), ...) \tag{4.3}$$

An argument for each parameter is given in the vector $(arg_1, arg_i, \dots, arg_n)$ at the time of building an ASP program, except for time steps parameters of incremental modules (always j or k). Time steps parameters are dynamically assigned their arguments as the program is extended for increased search length k. This is because there is no fixed limit on the search length or size of the program grounding. Since most modules include parameters for abstraction levels and time step bounds, all of which can be inferred from the program, to reduce verbosity, these parameters will be omitted when denoting the use of such a module in a program grounding. For example, the program $\Pi^{pl}_{j,k}(\mathcal{A}(pl, j, k))$ will be reduced to $\Pi^{pl}_{j,k}(\mathcal{A})$.

4.4.1 Instance Relations Module

The instance relations module generates the ground type and ancestry constraints (described in Section 3.5.4) used to define the domain and codomain of actions and state variables declared in the domain sorts (described in Section 3.5.5). The module header is as follows:

$$\mathcal{IRM}(tl : \mathbb{N}_{>0}) \tag{4.4}$$

This is the only static module used in HCR planning. It is grounded only once as a separate initial step, generating the type and ancestry constraints, and by extension the domain sorts. All rules in the $\mathcal{IRM}$ are then sub-sequentially simplified away and reduced to facts.

The rules of the instance relations module function as follows:

- Rules 4.5-4.7: All entities are instances of; their base class type at the declared level of that class, their super-class types at the same level as their most base class, and all the same class types at the next level which are not overridden by a descendant.
- Rules 4.8-4.9: An entity that is an instance of an ancestor class loses its inheritance to an override class at a level at which an instance of any of its descendant classes exist.
- Rules 4.10: An entity is a child of another at a given level if there is a ancestry relation between them, and they both exist at that level.
- Rules 4.11-4.12: An entity is a descendant of another if either: the entity is the child of the other entity; or, by transitivity, if the entity is a descendant of some different entity which is itself a descendant of the other entity.

$$insta\_of(l, \kappa, \psi) \leftarrow entity(\kappa, \psi),\ class(l, \kappa) \tag{4.5}$$

$$insta\_of(l_3, \kappa_1, \psi) \leftarrow insta\_of(l_2, \kappa_2, \psi),\ super\_class(\kappa_1, \kappa_2),\ \kappa_1 \neq \kappa_2$$
$$entity(\kappa_3, \psi),\ class(l_3, \kappa_3),\ class(l_1, \kappa_1),\ l_1 \geq l_2 \tag{4.6}$$

$$insta\_of(l-1,\kappa,\psi) \leftarrow insta\_of(l,\kappa,\psi),\ not\ ovr\_at(l-1,\kappa,\psi) \tag{4.7}$$

$$ovr\_at(l-1,\kappa,\psi_1) \leftarrow ovr\_by(l-1,\kappa,\psi_1,\psi_2),\ insta\_of(l,\kappa,\psi_1),\ insta\_of(l-1,\kappa,\psi_2) \tag{4.8}$$

$$ovr\_by(l-1,\kappa_3,\psi_1,\psi_2) \leftarrow desce\_of(l-1,\psi_1,\psi_2),\ ovr(\kappa_1,\kappa_2,\kappa_3),\ insta\_of(l,\kappa_1,\psi_1),\ insta\_of(l-1,\kappa_2,\psi_2),\ insta\_of(l,\kappa_3,\psi_1),\ insta\_of(l-1,\kappa_3,\psi_2) \tag{4.9}$$

$$child\_of(l,\psi_1,\psi_2) \leftarrow ances\_rel(\psi_1,\psi_2),\ insta\_of(l,\_,\psi_1) \tag{4.10}$$

$$desce\_of(l,\psi_1,\psi_2) \leftarrow child\_of(l,\psi_1,\psi_2) \tag{4.11}$$

$$desce\_of(l,\psi_1,\psi_3) \leftarrow desce\_of(l,\psi_1,\psi_2),\ desce\_of(l,\psi_2,\psi_3) \tag{4.12}$$

4.4.2 State Representation Module

The state representation module ensures correctness of the state representation. Informally, it ensures that the initial state is complete, and then ensures that all future states continue to remain complete throughout time, as planned actions transition them sequentially. It is the smallest module, but is crucial as it is always included when generating and solving all problems, except generation of final-goals. The module header is as follows:

$$\mathcal{SRM}(j:\mathbb{N}_{>0}, i:\mathbb{N}_{>j}, re:\{cl,cr,hr\}) \tag{4.13}$$

Where j and i are the start and current search steps respectively (as they are in problem definitions), and re is the state representation mode; *cl* stands for *classical*, *cr* for *refinement*, and *hr* for *hierarchical*. The semantics of the state representation modes are as follows:

- *Classical*: The classical mode represents the state at the current planning level only

$sl = \{pl\}$. This mode is only used for the single top-level classical planning problem.

- *Refinement*: The refinement mode simultaneously represents the state over an adjacent pair of levels; the current planning level and the previous $sl = \{pl, pl+1\}$. This mode is required for conformance refinement problems, to allow reasoning about conformance of a refined plan with the abstract plan it refines (see Section 3.5.1 for details).

- *Hierarchical*: The hierarchical mode represents the state over all levels in the hierarchy simultaneously $sl = \{1, ..., tl\}$. This is used to generate conforming initial states over all levels when defining a hierarchical planning problem (see Section 3.6.2 for details).

The state representation module contains only four rules (introduced in Section 3.5.1):

- Rule 4.14: *Law of awareness*; A generator choice rule, which obtains a complete initial state, by allowing the generation of any fluent state literal (assuming the argument tuple of the state variable was explicitly assigned a value in the problem definition).

- Rule 4.15: *Law of continuity*; An integrity constraint, which requires exactly one fluent state literal in the state for each unique argument tuple of all fluent state variables.

- Rule 4.16: *Law of inertia*; propagate the value of fluents through time if there is reason to believe that their value has been changed (by the actions of the robot).

- Rule 4.17: *Closed world assumption for defined fluents*; If a defined fluent is not known to be true, then it is assumed to be false.

$$\{\, holds(l, f(\bar{t}), \nu, 0) \,\} \leftarrow fluent(l, \_, f(\bar{t}), \nu) \tag{4.14}$$

$$\leftarrow not\ \{\, \nu : holds(l, f(\bar{t}), \nu, k) \,\} = 1, fluent(l, \_, f(\bar{t}), \_) \tag{4.15}$$

$$holds(l, f(\bar{t}), v, k) \leftarrow holds(l, f(\bar{t}), \nu, k-1),\ not\ not\ holds(l, f(\bar{t}), \nu, k),\ fluent(l, ine, f(\bar{t}), \nu) \tag{4.16}$$

$$holds(l, f(\bar{t}), false, k) \leftarrow not\ holds(l, f(\bar{t}), true, k),\ fluent(l, def, f(\bar{t}), \_) \tag{4.17}$$

The law of inertia is of particular interest, as it encodes a defeasible principle, which ensures that if a fluent has not been assigned a new value by the effects of an action on the current time step, then it takes its existing value from the previous time step, therefore ensuring that the system state does not change unless perturbed. This works through the double *naf* literal, which allows the conclusion $(f_i^l(\bar{t}) = \nu) \in \sigma_i^l$ to be derived from the premise $(f_{i-1}^l(\bar{t}) = \nu) \in \sigma_{i-1}^l$ without deductive support, if the conclusion is consistent in the state σ_i^l[7]. If the robot plans an action whose effects change the value of the fluent, either directly (through an action effect) or indirectly (through a state relation), then the law of continuity defeats the law of inertia, and the old value of the fluent cannot not be propagated to the next time step (because each fluent may only take one value at a time).

It is now possible to formally define a valid state and a valid state pair:

Definition 4.4.1 (Valid State) A valid state is a state that satisfies the rules of the state representation module and the laws of a (pair of) domain model(s). This requires that the state be complete according to the law of continuity and consistent with the domain laws.

Specifically, a state σ_j^{pl} is *classical valid* in DM^{pl} of DD^{tl}, *iff* the ASP program containing the domain model, the state, the state representation module in classical state representation mode, and the instance relations module has exactly one answer set:

$$\Pi_{j,j}^{pl,tl}(DD^{tl}, \epsilon(\sigma_j^{pl}), \mathcal{IRM}, \mathcal{SRM}(cl)) \rightarrow Z_{j,j}^{pl} = \{\zeta_{j,j,0}^{pl}\} \tag{4.18}$$

If $pl < tl$ then the planning domain model can be used for conformance refinement planning, and therefore state pairs must be represented to allow reasoning about conformance of refined plans with abstract plans from the model DM^{pl+1}. The definition is similar, a state pair $\delta_j^{pl} = \sigma_j^{pl} \cup \sigma_j^{pl+1}$ is called *conformance valid* if the program has exactly one answer set:

$$\Pi_{j,j}^{pl,tl}(DD^{tl}, \epsilon(\delta_j^{pl}), \mathcal{IRM}, \mathcal{SRM}(cr)) \rightarrow Z_{j,j}^{pl} = \{\zeta_{j,j,0}^{pl}\} \tag{4.19}$$

[7]In other words, the truth of $f_{i-1}^l(\bar{t}) = \nu$ in the previous state σ_{i-1}^l is sufficient support for its truth in the immediately next state σ_i^l, even though this conclusion may be defeated by additional knowledge.

It is important to understand that if a state σ_j^{pl+1} is classical valid in DM^{pl+1}, then it is not guaranteed that there will exist some state pair $\delta_j^{pl} = \sigma_j^{pl} \cup \sigma_j^{pl+1}$ containing σ_j^{pl+1} that is conformance valid in DM^{pl}. Intuitively, this is because an abstract domain model could remove enough constraints to allow a state that is valid at the abstract model, but which cannot be mapped to from any valid states of the more constrained original model[8].

It is also now possible to define generation of initial states over the hierarchy:

Definition 4.4.2 (Conforming Initial State Generation) A ground-level initial state is valid if the ASP program containing the entire hierarchical planning domain, the initial state, the state representation module in hierarchical state representation mode, and the instance relations module has at least one answer set:

$$\Pi_{0,0}^{1,tl}(DD^{tl}, \epsilon(\sigma_0^1), \mathcal{IRM}, \mathcal{SRM}(hr)) \rightarrow Z_{0,0}^1 \neq \emptyset \tag{4.20}$$

If there is exactly one answer set $Z_{0,0}^1 = \{\zeta_{0,0,0}^1\}$ then the initial state is consistent, such that there is exactly one interpretation of the state at all abstract levels of the hierarchy.

The initial states of some hierarchical problem HP^{tl} then become:

$$\Sigma^{tl} = \{\epsilon(\sigma_0^{tl}) \in \zeta_{0,0,0}^1\} \cup \{\delta_0^l \mid \epsilon(\delta_0^l) \in \zeta_{0,0,0}^1\} : l \in [1, tl) \tag{4.21}$$

If there is more than one answer set, then the initial state is inconsistent, because multiple interpretations of the abstract initial state exist, and therefore no single deterministic definition of the hierarchical problem can be obtained. This can only happen if the state abstraction mapping is not exhaustive and deterministic[9]. If there are no answer sets, then the initial state is invalid with the domain laws according to Definition 4.4.1.

[8]Since state relations and constraints cannot be removed in relaxed or condensed models used in this thesis (only converted to more general versions in condensed models), there is no reason to believe that a valid state of an abstract model will not have a valid state of the original that maps to it. Noting that, even in condensed models, a generalised version of a domain law containing ancestor entities only exists because of its original version of the law containing descendent entities which maps to it on their class override relations.

[9]There is some original model state literal which maps to multiple state literal of its abstract model.

4.4.3 Minimal Planning Module

The minimal planning module ensures that planned state transitions are legal, and generated plans are minimal in length and classical complete (according to Definition 3.7.2). It applies action effects, enforces action preconditions, handles final-goal achievement, enforces conditions for concurrent action planning, and applies the solution checking constraint (ensuring programs are satisfiable only when a solution exists). The module header is as follows:

$$\mathcal{MPM}(j : \mathbb{N}_{>0}, i : \mathbb{N}_{>j}, mb : \mathbb{N}_{\geq j}, af : \{\top, \bot\}, yi : \{\top, \bot\}, co : \{\top, \bot\}) \tag{4.22}$$

Where j and i are the start and current search steps respectively, mb is the minimum search length bound, af is a boolean which indicates whether the problem being solved is final, yi is a boolean which enables or disables sequential yield planning mode, and co is a boolean which enables or disables concurrent action planning. The minimum search length bound is disabled if the given value is 0, otherwise it is enabled. As aforementioned, the minimum search length bound is equal to the size of a conformance refinement planning problem. Note that if sequential yield planning mode is enabled, then the minimum search length bound must be disabled. It is also an error for the minimum search length bound or sequential yield planning mode to be enabled when solving a classical planning problem.

The rules of the minimal planning module function as follows:

- Rule 4.23: Generator choice rule for planning actions at the current planning level.
- Rules 4.24-4.25: Activity constraint; plan exactly one action on each step in sequential action planning, plan at least one action on each step in concurrent action planning.
- Rule 4.26: Apply the direct effects of an action on the time step it was planned at.
- Rules 4.27-4.28: The action precondition integrity constraints, ensuring that an action cannot be planned in a state that does not satisfy (violates) its preconditions.

- Rules 4.29-4.30: Goal satisfaction rules; a positive final-goal literal is satisfied if its fluent literal holds, a negative final-goal literal is satisfied if its fluent literal does not.
- Rule 4.31: Standard final-goal achievement; always enforce achievement of the final-goal if the problem is final and sequential yield planning is disabled.
- Rules 4.32-4.33: Sequential yield final-goal achievement; if sequential yield planning is enabled, then create an external atom, which when set to true (by the sequential yield planning algorithm) indicates that the end of the sub-goal stage sequence has been reached. This enables the volatile rule that enforces achievement of the final-goal.
- Rule 4.34: A monolevel plan is incomplete whilst any final-goal literal is unsatisfied.
- Rule 4.35: For each planned action, generate a sub-goal literal for each of its direct effects, whose sequence index is the time step i which it was planned at.

Since action effects and preconditions can be state dependent or state independent, the time step parameter of the effect or preconditions atoms can be omitted. There is resultantly a version of each of these rules with such substitutions made, excluded here for conciseness.

$$\{occurs(l, r, a(\bar{t}), i)\} \leftarrow \tag{4.23}$$

$$\leftarrow not\ occurs(l, r, a(\bar{t}), i) = 1, co = \bot \tag{4.24}$$

$$\leftarrow not\ occurs(l, r, a(\bar{t}), i) \geq 1, co = \top \tag{4.25}$$

$$holds(l, f(\bar{t}), v, i) \leftarrow effect(l, r, a(\bar{t}), f(\bar{t}), v, i),\ occurs(l, r, a(\bar{t}), i) \tag{4.26}$$

$$\leftarrow precond(l, r, a(\bar{t}), f(\bar{t}), v, true, i), \tag{4.27}$$
$$not\ holds(l, f(\bar{t}), v, i-1),\ occurs(l, r, a(\bar{t}), i)$$

$$\leftarrow precond(l, r, a(\bar{t}), f(\bar{t}), v, false, i), \tag{4.28}$$
$$holds(l, f(\bar{t}), v, i-1),\ occurs(l, r, a(\bar{t}), i)$$

$$goal\_sat(l, f(\bar{t}), v, true, i) \leftarrow goal(l, f(\bar{t}), v, true),\ holds(l, f(\bar{t}), v, i) \tag{4.29}$$

$$goal\_sat(l, f(\bar{t}), v, false, i) \leftarrow goal(l, f(\bar{t}), v, false),\ not\ holds(l, f(\bar{t}), v, i) \tag{4.30}$$

$$enforce\_achieve\_fgoals(i) \leftarrow af = \top, yi = \bot \tag{4.31}$$

$$\#external\ seq\_achieve\_fgoals(i) \leftarrow af = \top, yi = \top \tag{4.32}$$

$$enforce\_achieve\_fgoals(i) \leftarrow seq\_achieve\_fgoals(j), i \geq j \tag{4.33}$$

$$\begin{aligned} no\_plan\_exists(i) \leftarrow\ & goal(l, f(\bar{t}), v, B), \\ & not\ goal\_sat(l, f(\bar{t}), v, B, i), \\ & enforce\_achieve\_fgoals(i) : B \in \{true, false\} \end{aligned} \tag{4.34}$$

$$sub\_goal(l, r, a(\bar{t}), f(\bar{t}), v, i) \leftarrow occurs(l, r, a(\bar{t}), i), effect(l, r, a(\bar{t}), f(\bar{t}), v, i) \tag{4.35}$$

The remaining core part of the planning module is Rule 4.37, the solution checking constraint, a volatile rule specific for each search step $i > j$, which check whether a plan exists of length $i-j$. Essentially, the external atom $query(i)$ has its value set to $true$ on each search step i, checking for the existence of a plan by activating the integrity constraint. The atom $no\_plan\_exists(i)$ is the $false$ only *iff* all final-goals and sub-goal stages have been achieved. If they have not, then the integrity constraint evaluates to $true$ and the program is unsatisfiable. The program will therefore only have an answer set when a valid plan exists. Because the program is solved incrementally in the time step sequence $(i)_{i=mb}^{k}$, when such an answer set is found, then it is guaranteed to contain a minimal plan by definition.

$$\#external\ query(i) \leftarrow \tag{4.36}$$

$$\leftarrow no\_plan\_exists(i),\ query(i),\ i > (mb - 1) \tag{4.37}$$

The minimal planning module also handles concurrent action planning by enforcing the concurrency constraints proposed in Section 3.7.3. Recall that the purpose of concurrent action planning is to compress the length of plans and to provide more information about

the ordering constraints underlying actions. This is achieved by allowing sub-sequences of actions that are arbitrarily ordered to be grouped into one set. The constraints are restated:

1. Multiple concurrently planned actions cannot lead to an invalid state,
2. Multiple concurrently planned actions cannot affect the same fluent,
3. A set of concurrently planned actions must also be sequentially plannable in any permutation, this requires than the effects of an action cannot not unachieve the precondition of any other action planned concurrently on the same step. Specifically, the effect an action must not cause a fluent to take a value that violates the precondition of another action (the effect cannot falsify a positive or "truify" a negative precondition).

The first constraint does not need to be handled explicitly, since no answer set can contain an invalid state. Additional rules are needed to handle the latter constraints. Rule 4.38 handles the second and Rules 4.39 and 4.40 handle the third.

$$\leftarrow occurs(l, r_1, a(\bar{t})_1, i),\ occurs(l, r_2, a(\bar{t})_2, i), a(\bar{t})_1 \neq a(\bar{t})_2,\ r_1 \neq r_2,$$
$$effect(l, r_1, a_1, f(\bar{t}), v_1, i), effect(l, r_2, a_2, f(\bar{t}), v_2, i) \tag{4.38}$$

$$\leftarrow occurs(l, r_1, a(\bar{t})_1, i),\ occurs(l, r_2, a(\bar{t})_2, i), a(\bar{t})_1 \neq a(\bar{t})_2,\ v_1 \neq v_2,$$
$$effect(l, r_1, a(\bar{t})_1, f(\bar{t}), v_1, i),\ precond(l, r_2, a(\bar{t})_2, f(\bar{t}), v_2, true, i) \tag{4.39}$$

$$\leftarrow occurs(l, r_1, a(\bar{t})_1, i),\ occurs(l, r_2, a(\bar{t})_2, i), a(\bar{t})_1 \neq a(\bar{t})_2,$$
$$effect(l, r_1, a(\bar{t})_1, f(\bar{t}), v, i),\ precond(l, r_2, a(\bar{t})_2, f(\bar{t}), v, false, i) \tag{4.40}$$

It is now possible to define a legal state transition and monolevel plan:

Definition 4.4.3 (Legal State Transition) A legal state transition is a state transition that satisfies the rules of the state representation module, the minimal planning module, and the laws of (a pair of) domain model(s). This requires the start and end states of the

transition be valid, and the change between states be consistent with the domain laws.

Specifically, a state transition τ_j^{pl} is *classical legal* in DM^{pl} of DD^{tl}, *iff* the ASP program containing the domain model, the state transition, the rules of the state representation module in classical state representation mode, the instance relations module, and the minimal planning module with final-goal achievement disabled, has exactly one answer set:

$$\Pi_{j-1,j}^{pl,tl}(DD^{tl}, \epsilon(\tau_j^{pl}), \mathcal{IRM}, \mathcal{SRM}(cl), \mathcal{MPM}(-1, \bot, \bot, co)) \rightarrow Z_{j-1,j}^{pl} = \{\zeta_{j-1,j}^{pl}\} \quad (4.41)$$

The state transition is *conformance legal* if the same program with the state representation module in conformance refinement state representation mode has exactly one answer set:

$$\Pi_{j-1,j}^{pl,tl}(DD^{tl}, \epsilon(\tau_j^{pl}), \mathcal{IRM}, \mathcal{SRM}(cr), \mathcal{MPM}(-1, \bot, \bot, co)) \rightarrow Z_{j-1,j}^{pl} = \{\zeta_{j-1,j}^{pl}\} \quad (4.42)$$

It is not guaranteed that a classical legal state transition τ_j^{pl+1} will have a single conformance legal state transition τ_k^{pl} where both $\sigma_{j-1}^{pl+1} \subset \delta_{j-1}^{pl}$ and $\sigma_j^{pl+1} \subset \delta_j^{pl}$, such that the start and end states of the refinement state transition map to the start and end states of the classical state transition. This is intuitive, since abstract models remove or generalise constraints with the desire to reduce abstract plan lengths, by allowing state transitions that would not be possible in the original. Therefore, typically a sequence of state transitions are needed to form a path between the same start and end states in the original model.

A monolevel plan is legal if either program ($re \in \{cl, cr\}$) has exactly one answer set:

$$\Pi_{j,k}^{pl,tl}(DD^{tl}, \epsilon(\pi_{j,k}^{pl}), \mathcal{IRM}, \mathcal{SRM}(re), \mathcal{MPM}(-1, \bot, \bot, co)) \rightarrow Z_{j,k}^{pl} = \{\zeta_{j,k}^{pl}\} \quad (4.43)$$

Such a plan is final if it has an answer set with the final-goal enforced:

$$\Pi_{j,k}^{pl,tl}(DD^{tl}, \epsilon(\pi_{j,k}^{pl}), \mathcal{IRM}, \mathcal{SRM}(re), \mathcal{MPM}(-1, \top, \bot, co)) \rightarrow Z_{j,k}^{pl} = \{\zeta_{j,k}^{pl}\} \quad (4.44)$$

Intuitively, if the start state of the plan is the initial state of the hierarchical planning problem at the given level according to Definition 3.7.2 then the plan is classical complete.

4.4.4 Conformance Refinement Module

The conformance refinement module enforces the conformance constraint in refinement planning problems. It ensures that all sub-goal stages included in the problem, are achieved by the refined plan, in the same order as the actions from the abstract plan which produced them. This constitutes the refined plan achieving the same effects as the abstract plan. Therefore, the refined plan attains conformance with the abstract plan, as it expands and specialises to satisfy the stricter planning constraints of the original problem. This is the definition of conformance refinement planning. The module header is as follows:

$$\mathcal{CRM}(j : \mathbb{N}_{>0}, k : \mathbb{N}_{>j}, \varphi : \mathbb{N}_{>0}, \vartheta : \mathbb{N}_{\geq\varphi}, at : \{seqa, sima\}, yi : \{\top, \bot\}) \tag{4.45}$$

Where j and k are the start and current search steps respectively, and φ and ϑ are the first and last sub-goal stage indices (again, as they are in problem definitions), at is the sub-goal stage achievement type; *seqa* stands for *sequential achievement*, and *sima* for *simultaneous achievement*, and *yi* is a boolean which enables or disables sequential yield planning mode.

The rules of the conformance refinement module function as follows:

1. Rule 4.46: *Standard sub-goal stage inclusion*; for each time step i greater than the starting step j, include all the sub-goal stages at sequence indices ϱ between the first and last indices of the monolevel problem φ and ϑ inclusive.

2. Rule 4.47: *Set sequential yield sub-goal stage index*; defines the external atom for setting the current last in sequence sub-goal stage index for sequential yield mode.

3. Rule 4.48: *Sequential yield sub-goal stage inclusion*; for each time step i greater than the step y at which the current last sub-goal stage at index x was added, include all sub-goal stages at sequence indices ϱ between the first index of the monolevel problem φ and the current last x. I.e. include the current and all previous sub-goal stages.

Where y is the step after the previous sub-goal stage at index $x-1$ was achieved.

4. Rule 4.49: the current sub-goal stage to be achieved from the starting step j is initially the first sub-goal stage at index φ.

5. Sequential sub-goal stage achievement;

   (a) Rule 4.50: *Sub-goal literal satisfaction*; a current sub-goal literal is satisfied on a step in which it holds in the state.

   (b) Rule 4.51: *Current sub-goal goal literal sequential propagation*; a current sub-goal is propagated to the next time step if it has not yet been satisfied. Therefore, if the sub-goal literal is satisfied, it is not propagated. This individual evaluation of propagation allows sub-goal literals to be achieved sequentially.

6. Simultaneous sub-goal stage achievement;

   (a) Rule 4.52: the current sub-goal stage is not simultaneously achieved whilst any of its sub-goal literals with its index are not satisfied in the current state,

   (b) Rule 4.53: the sub-goal literals from the previous time step $i-1$ are propagated to the current time step i, if the subgoal index that was current at time step $i-1$ has not yet been simultaneously achieved.

7. Rule 4.54: if the sequence index that was current at the previous time step is no longer current at i, then the next in sequence subgoal stage index $\varrho+1$ becomes current at i.

8. Rule 4.55: a sub-goal stage is achieved on the last step upon its index is current (that is, it is current up until and inclusive of the time step at which it is satisfied in a state).

9. Rule 4.56: the current sub-goal stage sequence index ϱ to be achieved at k is equal to the index value of the current sub-goal literals.

10. Rule 4.57: the plan is incomplete whilst there is at least one current subgoal at step i i.e. all subgoal sequence indices must be exhausted to complete the plan.

$$include\_sub\_goal\_index(\varrho, i) \leftarrow yi = \bot,\ \varrho \in [\varphi, \vartheta],\ i > j \tag{4.46}$$

$$\#external\ current\_last\_sgoals(\varrho, i) \leftarrow yi = \top,\ \varrho \in [\varphi, x],\ i > j \tag{4.47}$$

$$include\_sub\_goal\_index(\varrho, i) \leftarrow yi = \top,\ \varrho \in [\varphi, x],\ i \geq y,\ cur\_last\_sub\_goal\_index(x, y) \tag{4.48}$$

$$cur\_sub\_goal(pl + 1, f(\bar{t}), v, \varphi, i) \leftarrow sub\_goal(pl + 1, f(\bar{t}), v, \varphi) \tag{4.49}$$

$$sub\_goal\_sat(l + 1, f(\bar{t}), v, \varrho, i) \leftarrow cur\_sub\_goal(l + 1, f(\bar{t}), v, \varrho, i - 1),\ holds(l + 1, f(\bar{t}), v, i) \tag{4.50}$$

$$cur\_sub\_goal(l + 1, f(\bar{t}), v, \varrho, i) \leftarrow at = seqa,\ cur\_sub\_goal(l + 1, f(\bar{t}), v, \varrho, i - 1),\ uach\_sub\_goals(l + 1, \varrho, i - 1) \tag{4.51}$$

$$unach\_sub\_goals(l + 1, \varrho, i) \leftarrow cur\_sub\_goal\_index(l + 1, \varrho, i),\ cur\_sub\_goal(l + 1, f(\bar{t}), v, \varrho, i),\ not\ sub\_goal\_sat(l + 1, f(\bar{t}), v, \varrho, i) \tag{4.52}$$

$$cur\_sub\_goal(l + 1, f(\bar{t}), v, \varrho, i) \leftarrow at = sima,\ cur\_sub\_goal(l + 1, f(\bar{t}), v, \varrho, i - 1),\ unach\_sub\_goals(l + 1, \varrho, i - 1) \tag{4.53}$$

$$cur\_sub\_goal(l + 1, f(\bar{t}), v, \varrho + 1, i) \leftarrow sub\_goal(l + 1, f(\bar{t}), v, \varrho + 1),\ include\_sub\_goal\_index(\varrho, i),\ cur\_sub\_goal\_index(l + 1, \varrho, i - 1),\ not\ cur\_sub\_goal\_index(l + 1, \varrho, i) \tag{4.54}$$

$$sgls_ach\_at(l + 1, \varrho, i) \leftarrow cur\_sub\_goal\_index(l + 1, \varrho, i),\ unach\_sub\_goals(l + 1, \varrho, i) \tag{4.55}$$

$$cur\_sub\_goal\_index(l + 1, \varrho, i) \leftarrow current\_sub\_goal(l + 1, f(\bar{t}), v, \varrho, i) \tag{4.56}$$

$$no\_plan\_exists(i) \leftarrow include\_sub\_goal\_index(\varrho, i),\ current\_sub\_goal(l + 1, f(\bar{t}), v, \varrho, i) \tag{4.57}$$

4.4.5 Plan Optimisation Module

The plan optimisation module handles optimisation of plan according to final-goal preemptive achievement and concurrent action planning. The module header is as follows:

$$\mathcal{POM}(j : \mathbb{N}_{>0}, k : \mathbb{N}_{>j}, \omega^{pl}, co : \{\top, \bot\}, pa : \{\top, \bot\}) \tag{4.58}$$

Where j and k are the start and current search steps respectively, ω^{pl} are the final-goal literals at level pl, co is a boolean which enables or disables concurrent action planning, and pa is a boolean which enables or disables final-goal pre-emptive achievement.

Optimisation over answer sets is using weak constraints. These are similar to integrity constraints, except when they are satisfied, instead of making the program unsatisfiable, they add their weight to the cost of the answer set. The ASP solver is then guaranteed to find an answer set with minimum total cost (sum of weights of all satisfied weak constraints). The minimisation of actions in concurrent action planning is implemented according to a weak constraint defined by Equation 4.59[10], which applies unit cost of one for each planned action. A satisfiable plan is then guaranteed to contain a minimal number of actions. Final-goal pre-emptive achievement is implemented according to a weak constraint defined by Equation 4.60. This applies a negative unit cost (a reward) whenever an action's effect achieves a final-goal literal. A satisfiable plan is then guaranteed to contain a maximal number of actions which pre-emptively achieve final-goal literals. Note that 4.59 is applicable only *iff* $co = \top$ and 4.60 *iff* $pa = \top$. To handle multiple objectives when both are enabled, the solver can simply minimise 4.59 and then 4.60, separately, since they are non-interacting.

$$\sum_{i=j}^{k} 1 \leftarrow r \Rightarrow a^{pl}_{i+1}(\bar{t}) \in \alpha^{pl}_{i+1}[\tau^{pl}_i] : \tau^{pl}_i \in \pi^{pl}_{j,k} \tag{4.59}$$

$$\sum_{i=j}^{k} -1 \leftarrow r \Rightarrow a^{pl}_{i+1}(\bar{t}) \in \alpha^{l}_{i+1}[\tau^{l}_i], f^l(\bar{t}) = v \in \varsigma^{l}_{i+1}[\tau^{l}_i], fg(\bar{t}) = v \in \omega^{pl} : \tau^{l}_i \in \pi^{l}_{j,k} \tag{4.60}$$

[10]Equations representing the weak constraints are presented instead of rules for clarity and conciseness.

4.4.6 Goal Abstraction Module

The goal abstraction module serves only to generate a conforming final-goal over all levels in an abstraction hierarchy. The module header, takes no parameters, as follows:

$$\mathcal{GAM}() \tag{4.61}$$

The goal abstraction module converts all final-goal literals into standard fluent state literals. A partial state containing only fluents declared as final-goal fluents are then represented over all levels in the abstraction hierarchy simultaneously (as with initial state generation). The initial state at all abstraction levels is then converted back to final-goal literals, resulting in the conforming set of final-goals for a hierarchical planning problem.

1. Rules 4.64-4.63: Ground-level final-goals are applied as state literals in the initial state.
2. Rules 4.64-4.65: Convert the initial state back to final-goal literals.
3. Rules 4.66-4.67: Defined final-goals must be satisfied in the initial state; if the defined final-goal must hold true, then it cannot hold false, and if the defined final-goal must hold false, then it cannot hold true.
4. Rules 4.68-4.70: The goal abstraction module contains three rules from the state representation module; law of awareness, law of continuity, and the closed-world assumption for defined fluents, each of which is modified to only apply to final-goal fluents.

$$holds(l, f(\bar{t}), \nu, 0) \leftarrow final\_goal(l, f(\bar{t}), \nu, true) \tag{4.62}$$

$$not\ holds(l, f(\bar{t}), \nu, 0) \leftarrow final\_goal(l, f(\bar{t}), \nu, false) \tag{4.63}$$

$$final\_goal(l, f(\bar{t}), \nu, true) \leftarrow holds(l, f(\bar{t}), \nu, 0), goal\_fluent(l, f(\bar{t})) \tag{4.64}$$

$$final\_goal(l, f(\bar{t}), \nu, false) \leftarrow not\ holds(l, f(\bar{t}), \nu, 0), goal\_fluent(l, f(\bar{t})) \tag{4.65}$$

$$\leftarrow holds(l, f(\bar{t}), false, 0), final\_goal(l, f(\bar{t}), b, b),$$
$$inst\_of(l, bool, b), fluent(l, defined, f(\bar{t}), \_) \tag{4.66}$$
$$\leftarrow holds(l, f(\bar{t}), true, 0), final\_goal(l, f(\bar{t}), b_1, b_2),$$
$$inst\_of(l, bool, b_1), inst\_of(l, bool, b_2), b_1 \neq b_2,$$
$$fluent(l, defined, f(\bar{t}), \_) \tag{4.67}$$
$$\{\, holds(l, f(\bar{t}), \nu, 0) \,\} \leftarrow fluent(l, \_, f(\bar{t}), \nu), goal\_fluent(l, f(\bar{t})) \tag{4.68}$$
$$\leftarrow not\ \{\, \nu : holds(l, f(\bar{t}), \nu, k) \,\} = 1, fluent(l, \_, f(\bar{t}), \_),$$
$$goal\_fluent(l, f(\bar{t})) \tag{4.69}$$
$$holds(l, f(\bar{t}), false, k) \leftarrow not\ holds(l, f(\bar{t}), true, k),\ fluent(l, def, f(\bar{t}), \_),$$
$$goal\_fluent(l, f(\bar{t})) \tag{4.70}$$

It is also now possible to define generation of the final-goal over the hierarchy:

Definition 4.4.4 (Conforming Final-Goal Generation) A ground-level final-goal is valid if the ASP program containing the entire hierarchical planning domain, the final-goal, the instance relations module, and the goal abstraction module, has at least one answer set:

$$\Pi^{1,tl}_{0,0}(DD^{tl}, \epsilon(\sigma^1_0), \mathcal{IRM}, \mathcal{GAM}) \rightarrow Z^1_{0,0} \neq \emptyset \tag{4.71}$$

If there is exactly one answer set $Z^1_{0,0} = \{\zeta^1_{0,0,0}\}$ then the final-goal is consistent, such that there is exactly one interpretation of the final-goal at all abstract levels of the hierarchy.

The final-goals of some hierarchical problem HP^{tl} are then:

$$\Omega^{tl} = \{\omega^l_0 \mid \epsilon(\omega^l_0) \in \zeta^1_{0,0,0}\} : l \in [1, tl] \tag{4.72}$$

As with initial-state generation in Definition 4.4.2; if there is more than one answer set the final-goal is inconsistent, and if there are no answer sets then the the final-goal is invalid.

4.5 Planning Algorithms and Systems

This section presents the four core planning algorithms used in HCR planning.

1. *Hierarchical Planning Algorithm*: Controls hierarchical and online planning progression along the horizontal and vertical axes of planning, according to decisions made by the online planning method. Generates monolevel planning problems by extracting: the problem's initial state and final-goal from the hierarchical planning domain definition; or its intermediate starting state and conformance constraints for the current hierarchical plan if the problem is a partial refinement planning problem, and; the sub-goal stage range for the given problem as decided by the problem division strategy.

2. *Monolevel Planning Algorithm*: Loads the necessary operational modules of ASH and constructs the ASP program for a given monolevel planning problem from the domain and problem specific components passed from the hierarchical planning algorithm.

3. *Sequential Yield Planning Algorithm*: Inserts and minimally achieves each individual sub-goal stage of a conformance refinement planning problem sequentially. This allows the progression of search towards the final-goal to be observed externally.

4. *ASP Incremental Search System*: Handles the search for plans that solve any given monolevel planning problem, by incrementally grounding and solving the ASP program passed from the monolevel planning algorithm for that problem. Incremental search continues until a minimal solution is found, or a search length or time limit is reached.

Each algorithm and their subordinate functions include at least one **return** statement, and explicit **success** or **failure** declarations. A **return** statement has its usual meaning of returning the result of resolving an algorithm call. A **success** declaration states that the algorithm has found the expected result and a valid solution, whereas a **failure** declaration states that the expected result could not be obtained and returning would be an error.

The following special keywords are used to denote calls to the Clingo ASP system used by ASH. Keyword **build** constructs the program's Abstract Syntax Tree (AST) ready for grounding and solving. Keywords **base ground** $\Pi^{pl}_{j,k}$ ground the base program part of the logic program, and keywords **inc ground** $\Pi^{pl}_{j,k}$ ground the incremental part of the logic program up to the current search length; this extends all program parts with the step parameters j, k as their first parameters (see Section 4.4). Keywords **load grounding** $\Pi^{pl}_{j,k}$ reloads an existing program that has previously been grounded, **store grounding** $\Pi^{pl}_{j,k}$ makes such a grounding available for loading, and **delete grounding** $\Pi^{pl}_{j,k}$ deletes such a grounding. Keywords **extend** Π_1 **with** Π_2 extends the program Π_1 with the grounding of Π_2. Keyword **solve** attempts to solve the logic program, checking if a solution exists whose length is equal to the given search length. Keyword **simplify** cleanups and simplifies the logic program: it is used in incremental solving to prepare the logic program from the next search step. Keywords **assign external** $variable = value$ **on** $\Pi^{pl}_{j,k}$ and **release external** $variable = value$ **on** $\Pi^{pl}_{j,k}$, set and unset the value of externable variables within the logical problem. These variables can be changed externally in the logic program by an imperative algorithm between each step of incremental grounding and solving of a program.

4.5.1 Hierarchical Planning Algorithm

The hierarchical planning algorithm is the primary workhorse and controls the process of HCR planning. It is summarised in pseudo code Algorithm 1. It functions as follows:

1. The hierarchical problem is first initialised. The initial-state and final-goal are given to the HCR planner only at the ground-level (which is the actual problem to solve, see Section 3.6.1). Initialising the hierarchical problem then requires generating conforming initial states and final-goals over the entire abstraction hierarchy using Algorithm 2, such that all monolevel plans in a hierarchical plan start and end in conforming states (see Section 3.6.2 for why this is necessary, and Definitions 4.4.2 and 4.4.4 for details).

- A program is built that represents the state of the hierarchical planning domain over all abstraction levels simultaneously (hierarchical representation hr, see Section 4.4.2), the ground-level initial-state is then included, and the program is solved. This automatically abstracts the initial-state to all abstract levels. This is such that the resulting answer set contains a conforming initial-state over all monolevel problems in the hierarchical planning problem[11]. If there is no answer set, then the initial-state is invalid. If there is more than one answer set, then the initial-state is inconsistent, and the planner cannot find a unique conforming interpretation of the ground-level initial-state at all abstraction levels.
- This happens similarly for the final-goals, except the goal abstraction module is used instead of the state abstraction module (in hr mode) (see Section 4.4.6). It is necessary to use a separate module, because only fluents specified as final-goal fluents are part of the final-goal specification, all others are left unmentioned[12].

2. Hierarchical planning begins by solving the most abstract top-level classical problem.
   - The top-level classical plan is then used to initialise the hierarchical plan,
   - The combined refinement problem of this plan is then immediately divided.
3. Planning then proceeds to enter the online partial-planning loop.
   - This continues for an indefinite number of increments until the problem is solved,
   - The hierarchical problem is solved once all abstraction levels in the hierarchical plan are complete. It is sufficient to check whether the ground-level is complete.
4. Call the online planning method to determine: the highest-level $hl \in \mathbb{N}_{>0}$ and lowest-level $ll \in \mathbb{N}_{[hl,1]}$, of the contiguous level range over the vertical axis of hierarchical planning, involved in the current online planning increment, and; whether to divide $divide \in \{\top, \bot\}$ the refinement problems of abstract plans during this movement.

[11] To have initial-state conformance all fluent state literals must map to each other deterministically.

[12] To have final-goal conformance only the fluent state literals in the partial state defined by the final-goal fluents must map to each other, all other fluents can freely take any other value.

- Ground-first chooses hl as the highest non-final level and ll as the ground-level,
- Complete-first chooses both hl and ll as the highest non-final level,
- Hybrid chooses ground-first initially and complete-first subsequently.

5. Update number of solved problems and find size of the current monolevel problem,
6. Obtain a complete definition of the monolevel problem by extracting the necessary components from the hierarchical plan using Algorithm 3. If the problem is partial and non-initial, find its start state from the hierarchical plan's conformance mapping.
7. Call the monolevel planning algorithm to solve the monolevel problem.
8. Update the hierarchical plan with the monolevel plan that solved the problem.
9. Call the division strategy (if requested) to make divisions over the combined refinement problem of the monolevel plan (if it is abstract), and update the partial-problem templating function Δ (see Section 3.6.5 for details of this function).
   - Recall the following functions introduced previously: the number of monolevel problems that have been solved at a given level $problems(l) = \varkappa$; the number of abstract partial-plans whose refinement problems have been divided over $divided(l) = n$; whether the monolevel plan at a given level is complete $complete(l) = \rho$; and the current concatenated monolevel plan length $mlen(l) = k$.
   - The division strategy can be called every time a monolevel plan is generated if requested by the online method. When the strategy is called, it takes as argument the number of problems that have been divided at the given level so far $divided(l) = n$ and the current problem number $problems(l) = \varkappa$. The strategy must then concatenate and divide the combined refinement problem of all partial-plans that solved the partial-problems in the range $[n, \varkappa]$.
10. End both loops, return to step 3 if level ll has been reached, else return to step 4. If the ground-level is final, break instead the loop and return the ground-level plan.

4.5.2 Monolevel Planning Algorithm

The monolevel planning algorithm handles the creation of ASP logic programs that define monolevel classical or conformance refinement planning problems and makes calls to search for solutions to those problems. It is agnostic to the process of hierarchical planning.

The monolevel planning algorithm is summarised in pseudo code in Algorithm 4, it also relies on auxiliary Algorithm 5 to create the necessary logic programs.

There are three search modes that the monolevel planning algorithm can use;

1. *Standard Search Mode*: The standard incremental search mode, in which the planner incrementally increases the search length in ascending order $(i)_{i=1}^{\infty}$. It starts from the initial step $i = 1$, checks for a solution (by solving the logic program) at every time step, and returns on the first step one is found, guaranteeing a minimal length plan.

2. *Minimum Search Length Bound Search Mode*: Similar to the standard search mode, except the planner does not check for a solution until a minimum bound on the plan length has been reached to reduce search time. The bound is currently used only for refinement problems, and is equal to the number of sub-goal stages in the problem.

3. *Sequential Yield Search Mode*: A novel search mode presented in Sub-Section 4.5.3.

4.5.3 Sequential Yield Planning Algorithm

The sequential yield planning algorithm works alongside the monolevel planning algorithm. It modifies the manner in which search is performed, such that the sub-goal stages of a combined refinement planning problem are "inserted" and considered sequentially. The planner then searches for a tentative solution that minimally achieves each first sub-goal stage in sequence, until it achieves the last sub-goal stage, and the solution is accepted.

The sequential yield algorithm is summarised in pseudo code in Algorithm 6. The $current\_last\_sgoals$ and seq_ach_fgoals external atoms are used to set the current sub-goal stage index in the $\mathcal{CRM}$ and enforce achievement of the final-goal at the end of the sub-goal sequence in the $\mathcal{MPM}$, see Sections 4.4.4 and 4.4.3 for details. Step parameter h reflects/tracks the search length reached to minimally achieve the current sub-goal stage.

The purpose and benefits of the algorithm is two-fold;

1. *Observable Planning*: The progression of the planner towards a solution to the given problem, in terms of the number of achieved sub-goal stages, can be directly observed during planning (in contrast, in standard search there is no way to observe how close the planner is to a solution until one is found). For a human operator, it is beneficial to be able to gauge the length of time to wait before the robot begins execution.

2. *Faster Planning*: There is the potential that sequential yielding could reduce planning time. The idea is that if no solution to the problem exists at a given search length, then proving so requires exhausting the search space. Whereas, if a solution does exist, it might be found after having exhausted only part of the search space. Given that sequential yield search will contain multiple search steps upon which a tentative solution exists, much of the necessary search could be avoided using the proposed approach.

4.5.4 The Incremental Search System

The incremental search system handles the incremental ASP grounding and solving process used for finding minimal length plans. The system functions as summarised in Algorithm 7. For a description of the primary benefits of incremental ASP search see Section 2.4.3. The algorithm uses the *query* external atom to progress the time step of the solution checking constraint from the $\mathcal{MPM}$ as the search length increases, see Section 4.4.3 for details.

Algorithm 1 Hierarchical Conformance Refinement Planning Algorithm.

procedure HIERARCHICAL_PLAN($HD^{tl}, \sigma_0^1, \omega^1, at, yi, co, pa, sg, blend, s\_lim, t\_lim$)
    Parameters: HD^{tl} is a hierarchical planning domain, σ_0^1 is a ground-level initial state, and ω^1 is a ground-level final-goal, $at \in \{seq\_a, sim\_a\}$ is the sub-goal achievement type, $yi \in \{\top, \bot\}$ enables sequential yield planning, $co \in \{\top, \bot\}$ enables concurrent action planning, $pa \in \{\top, \bot\}$ enables final-goal pre-emptive achievement of ω^{pl}, $sg \in \{\top, \bot\}$ enables saved program groundings, $blend$ is the sub-goal blend quantity, s_lim and t_lim are the maximum search length and time limit.

    Initialise the hierarchical planning problem.
    $\Sigma^{tl}, \Omega^{tl} \leftarrow$ HIERARCHICAL_PROBLEM($HD^{tl}, \sigma_0^1, \omega^1$)
    $HP^{1,tl} \leftarrow \langle HD^{tl}, \Sigma^{tl}, \Omega^{tl} \rangle$

    Solve classical problem, initialise hierarchical plan, and divide its refinement problem.
    $MP^{tl}_{0,0,0} \leftarrow \langle DD^{tl}, \sigma_0^{tl}, \omega^{tl}, \emptyset \rangle$
    $\pi^{tl}_{0,k} \leftarrow$ MONOLEVEL_PLAN($MP^{tl}_{0,0,0}, null, \bot, \bot, \emptyset, s\_lim, t\_lim, null$)
    $\eta^{tl}[tl, 1] \leftarrow \pi^{tl}_{0,k}, \emptyset, \top$
    $\Delta, n \leftarrow$ DIVISION_STRATEGY($\eta^{tl}, tl, 1, 1$)
    $divided(tl) \leftarrow n$

    Loop handling horizontal axis of online planning.
    while The ground-level is incomplete ($\neg complete(1)$) **do**
        Call online method to determine level range for current online planning increment.
        $ll, hl, divide \leftarrow$ ONLINE_METHOD(η^{tl})

        Loop handling vertical axis of hierarchical planning.
        for all $pl \in [ll, hl]$ **in the sequence** $(pl)^{ll}_{pl=hl}$ **do**
            $problems(pl) \leftarrow problems(pl) + 1$
            $\varkappa \leftarrow problems(pl)$
            $\varphi, \vartheta \leftarrow \Delta(pl, \varkappa)$

            Obtain monolevel problem and call monolevel planning algorithm to solve it.
            $MP^{l}_{j,\varphi,\vartheta} \leftarrow$ MONOLEVEL_PROBLEM($HP^{tl}, \eta^{tl}, pl, \varkappa, \varphi, \vartheta$)
            $\pi^{l}_{j,k}, \gamma_{\varphi,\vartheta} \leftarrow$ MONOLEVEL_PLAN($MP^{l}_{j,\varphi,\vartheta}, at, yi, co, pa, \omega^{pl}, \lambda_{\varphi,\vartheta+blend}, s\_lim, t\_lim$)
            $\eta^{pl,tl}[pl, \varkappa] \leftarrow \langle \pi^{l}_{j,k}, \gamma_{\varphi,\vartheta}, \rho \rangle$

            Call division strategy to allow it to divide any undivided plans.
            if Online method requests problem division ($divide = \top \wedge pl \neq 1$) **then**
                $\Delta, n \leftarrow$ DIVISION_STRATEGY($\eta^{tl}, pl, divided(pl), \varkappa$)
                $divided(pl) \leftarrow n$

    declare success
    return $\pi^1_{0,k} = \{\pi^1_{j,i}[\eta^{1,tl}(1, \varkappa)] \mid \varkappa \in problems(1)\}$

Algorithm 2 Generate Conforming Initial States and Final-Goals over the Hierarchy.

function HIERARCHICAL_PROBLEM($HD^{tl}, \sigma_0^1, \omega^1$)
    Parameters: HD^{tl} is a hierarchical planning domain, σ_0^1 is a ground-level initial state, and ω^1 is a ground-level final-goal.

    Build, ground, and solve a program representing the initial state over the hierarchy.
    $\Pi_{0,0}^{1,tl} \leftarrow$ **build** $\Pi_{0,0}^{1,tl}(DD^{tl}, \epsilon(\sigma_0^1), \mathcal{IRM}, \mathcal{SRM}(hr))$
    ground all $\Pi_{0,0}^{1,tl}$
    $Z_{0,0}^{1,tl} \leftarrow$ **solve** $\Pi_{0,0}^{1,tl}$
    if There is exactly one answer set ($Z_{0,0}^{1,tl} = \{\zeta_{0,0,0}^1\}$) **then**
        The ground-level initial-state has a unique interpretation at all abstract levels.
        $\Sigma^{tl} \leftarrow \{\epsilon(\sigma_0^{tl}) \in \zeta_{0,0,0}^1\} \cup \{\delta_0^l \mid \epsilon(\delta_0^l) \in \zeta_{0,0,0}^1\} : l \in [1, tl)$
    else
        The ground-level initial-state has multiple interpretations at some level.
        declare failure

    Build, ground, and solve a program representing the final-goal over the hierarchy.
    $\Pi_{0,0}^{1,tl} \leftarrow$ **build** $\Pi_{0,0}^{1,tl}(DD^{tl}, \epsilon(\omega^1), \mathcal{IRM}, \mathcal{GAM})$
    ground all $\Pi_{0,0}^{1,tl}$
    $Z_{0,0}^{1,tl} \leftarrow$ **solve** $\Pi_{0,0}^{1,tl}$
    if There is exactly one answer set ($Z_{0,0}^{1,tl} = \{\zeta_{0,0,0}^1\}$) **then**
        $\Omega^{tl} \leftarrow \{\omega^l \mid \epsilon(\omega^l) \in \zeta_{0,0,0}^1\} : l \in [1, tl]$
    else
        The ground-level final goal has multiple interpretations at some level.
        declare failure

    return Σ^{tl} **and** Ω^{tl}

Algorithm 3 Obtain Monolevel Problem Definition.

function MONOLEVEL_PROBLEM($HP^{tl}, \eta^{tl}, pl, \varkappa, \varphi, \vartheta$)
    Parameters: HP^{tl} is a hierarchical planning problem, η^{tl} is a hierarchical plan, pl is the current planning level, $\varkappa$ is the current problem number at that level, and $[\varphi, \vartheta]$ is a sub-goal stage range.

    Get sub-goal stages to refine.
    $\pi^{pl}_{0,z} \leftarrow \eta^{tl}(pl+1, problem(pl+1))$
    $\lambda^{pl+1}_{\varphi,\vartheta} \leftarrow \{\lambda^{pl+1}_{\varrho} = \varsigma^{pl+1}_{\varrho}[\tau^p lpl+1_{\varrho}] \mid \tau^p lpl+1_{\varrho} \in \pi^{pl}_{0,z}\}$

    if First sub-goal stage index is not the first $\varphi \neq 1$ **then**
        Start from end state of last partial-plan.
        $\pi^{pl}_{a,b}, \gamma^{pl}_{c,d} \leftarrow \eta^{tl}(pl, \varkappa - 1)$
        $j \leftarrow \gamma^{pl}_{c,d}(\varphi - 1)$
        $\delta^{pl}_j \leftarrow \delta^{pl}_j[\tau^{pl}_j \in \pi^{pl}_{a,b}]$
    else
        Start from the initial-state.
        $j \leftarrow 0$
        $\delta^{pl}_j \leftarrow \sigma^{pl} \in \Sigma^{tl}[HP^{tl}]$

    Determine whether the problem is final.
    if The monolevel plan at previous level is complete and requested sub-goal stage index range includes last sub-goal stage $complete(pl+1) = \top \wedge \vartheta = mlen(pl+1)$ **then**
        $\omega^{pl} \leftarrow \omega^{pl} \in \Omega^{tl}[HP^{tl}]$
        return $\langle DD^{tl}, \delta^{pl}_j, \omega^{pl}, \lambda^{pl+1}_{\varphi,\vartheta} \rangle$
    else
        return $\langle DD^{tl}, \delta^{pl}_j, \emptyset, \lambda^{pl+1}_{\varphi,\vartheta} \rangle$

Algorithm 4 Monolevel Classical and Conformance Refinement Planning Algorithm.

procedure MONOLEVEL_PLAN($MP^{pl}_{j,\varphi,\vartheta}, at, yi, co, pa, \omega^{pl}, sg, \lambda_{\varphi,\vartheta+blend}, s\_lim, t\_lim$)
    Parameters: $MP^{pl}_{j,\varphi,\vartheta}$ is a monolevel planning problem, $at \in \{seq\_a, sim\_a\}$ is the sub-goal achievement type, $yi \in \{\top, \bot\}$ enables sequential yield planning, $co \in \{\top, \bot\}$ enables concurrent action planning, $pa \in \{\top, \bot\}$ enables final-goal pre-emptive achievement of ω^{pl}, $sg \in \{\top, \bot\}$ enables saved program groundings, $\lambda_{\varphi,\vartheta+blend}$ are any blended sub-goal stages, s_lim and t_lim are the maximum search length and time limit.

    Build the program representing the monolevel problem.
    $\Pi^{pl}_{j,k} \leftarrow$ BUILD_PROGRAM($MP^{pl}_{j,\varphi,\vartheta}, at, yi, co, pa, \omega^{pl}, \lambda_{\varphi,\vartheta+blend}$)

    $t\_start \leftarrow$ SYSTEM_TIME()
    if Sequential yield planning is enabled ($pl < tl$ **and** $yi = \top$) **then**
        $\zeta^{pl}_{j,t\_total,0}, i, t\_total \leftarrow$ YIELD_SOLVE($\Pi^{pl}_{j,k}, j, s\_lim, \varphi, \vartheta, af, t\_start, t\_lim$)
    else
        $\zeta^{pl}_{j,t\_total,0}, i, t\_total \leftarrow$ INC_SOLVE($\Pi^{pl}_{j,k}, j, s\_lim, mb, t\_start, t\_lim$)

    if Saved grounding is enabled and problem is non-final ($sg = \top$ **and** $af = \bot$) **then**
        store grounding $\Pi^{pl}_{j,k}$
    else
        delete grounding $\Pi^{pl}_{j,k}$

    Extract and return the monolevel plan and conformance mapping.
    declare success
    $\pi^{l}_{j,k}, \gamma_{\varphi,\vartheta} \leftarrow$ **extract** $\epsilon(\pi^{l}_{j,k})$ **and** $\epsilon(\gamma_{\varphi,\vartheta})$ **from** $\zeta^{pl}_{j,t\_total,0}$
    if Blending is enabled ($blend \neq 0$) **then**
        $\pi^{l}_{j,p} \leftarrow \{\tau^{l}_{i} \mid \pi^{l}_{j,k} : i \in \gamma_{\varphi,\vartheta+blend}(\vartheta)\}$
        return $\pi^{l}_{j,p}$ **and** $\gamma_{\varphi,\vartheta}$
    else
        return $\pi^{l}_{j,k}$ **and** $\gamma_{\varphi,\vartheta}$

Algorithm 5 Build ASP Program for Monolevel Planning.

function BUILD_PROGRAM($MP^{pl}_{j,\varphi,\vartheta}, at, yi, co, pa, \omega^{pl}, \lambda_{\varphi,\vartheta+blend}$)
    Parameters: $MP^{pl}_{j,\varphi,\vartheta}$ is a monolevel planning problem, $at \in \{seq\_a, sim\_a\}$ is the sub-goal achievement type, $yi \in \{\top, \bot\}$ enables sequential yield planning, $co \in \{\top, \bot\}$ enables concurrent action planning, $pa \in \{\top, \bot\}$ enables final-goal pre-emptive achievement of ω^{pl}, $\lambda_{\varphi,\vartheta+blend}$ are any blended sub-goal stages.

    Build a logic program representing the monolevel problem to solve.
    if Problem type is classical ($pl = tl$) **then**
        $\Pi^{pl}_{j,k} \leftarrow$ **build** $\Pi^{pl}_{j,k}(MP^{pl}_{0,0,0}, \mathcal{IRM}, \mathcal{SRM}(cl), \mathcal{MPM}(0, \top, \bot, \bot))$
        base ground $\Pi^{pl}_{j,k}$

    else if Problem type is conformance refinement ($pl < tl$) **then**
        Determine final-goal achievement.
        $af \leftarrow complete(pl + 1) \wedge \vartheta = mlen(pl + 1)$

        Enable and calculate minimum search length bound.
        $mb \leftarrow 0$
        if Sequential yield is disabled $yi = \bot$ **then**
            $mb \leftarrow j + (\vartheta - \varphi)$

        Build the program (currently $j = k$).
        $\Pi^{pl}_{j,k} \leftarrow$ **build** $\Pi^{pl}_{j,k}(MP^{pl}_{j,\varphi,\vartheta}, \mathcal{IRM}, \mathcal{SRM}(re), \mathcal{MPM}(mb, af, yi, co),$
                $\mathcal{CRM}(at, yi), \mathcal{POM}(\omega^{pl}, co, pa))$

        if Blending is enabled ($blend \neq 0$) **then**
            $\Pi^{pl}_{j,k} \leftarrow$ **extend** $\Pi^{pl}_{j,k}$ **with** $\lambda_{\vartheta+1,\vartheta+blend}$

        if Saved grounding is available and enabled ($sg = \top$ **and** $j > 0$) **then**
            load grounding $\Pi^{pl}_{0,j-1}$
            $\Pi^{pl}_{j,k} \leftarrow$ **extend** $\Pi^{pl}_{0,j-1}$ **with** $\Pi^{pl}_{j,k}$
        else
            base ground $\Pi^{pl}_{j,k}$
    return $\Pi^{pl}_{j,k}$

Algorithm 6 Sequential Yield Planning Algorithm.

procedure YIELD_PLAN($\Pi^{pl}_{j,k}, s\_start, s\_lim, \varphi, \vartheta, af, t\_start, t\_lim$)
    Parameters: $\Pi^{pl}_{j,k}$ is an ASP program defining a monolevel planning problem to solve, s_start and s_lim are the start step and step limit, $[\varphi, \vartheta]$ is a sub-goal stage range, $af \in \{\top, \bot\}$ determines if the problem is final, s_lim and t_lim are the maximum search length and time limit.

    Keep track of program and planning time over multiple searches.
    $\Pi^{pl}_{j,h} \leftarrow \Pi^{pl}_{j,k}$
    $t\_total \leftarrow t\_start$

    Generate a partial-plan that reaches up to each sub-goal stage in sequence.
    for all $\varrho \in [\varphi, \vartheta]$ **in the sequence** $(\varrho)^{\vartheta}_{\varrho=\varphi}$ **do**

        Set externals to reflect current progression along sub-goal stage sequence.
        if Sub-goal index surpasses first ($\varrho > \varphi$) **then**
            release external $current\_last\_sgoals = \varrho - 1$ **on** $\Pi^{pl}_{j,h}$
        assign external $current\_last\_sgoals = \varrho$ **on** $\Pi^{pl}_{j,h}$
        if Sub-goal index is last and problem is final ($\varrho = \vartheta$ **and** $af = \top$) **then**
            assign external $seq\_ach\_fgoals = \top$ **on** $\Pi^{pl}_{j,h}$

        Incremental solve until answer set is found achieving up to sub-goal stage ϱ.
        $\zeta^{pl}_{j,h,0}, t\_total \leftarrow$ INC_SOLVE($\Pi^{pl}_{j,h}, h, s\_lim, 0, t\_total, t\_lim$)

    Return if all sub-goal stages have been exhausted.
    declare success
    return $\zeta^{pl}_{j,h,0}$ **and** t_total

Algorithm 7 Incremental Search Algorithm.

procedure INC_SEARCH($\Pi^{pl}_{j,k}, s\_start, s\_lim, mb, t\_start, t\_lim$)
    Parameters: $\Pi^{pl}_{j,k}$ is an ASP program defining a monolevel planning problem to solve, s_start and s_lim are the start step and step limit, mb is the minimum search length bound, s_lim and t_lim are the maximum search length and time limit.

    Solve the program incrementally in ascending order (skip the search length bound).
    for all $i \in (s\_start + mb, s\_lim]$ **in the sequence** $(i)^{s\_lim}_{i=s\_start+mb+1}$ **do**

        $\Pi^{pl}_{j,i} \leftarrow$ **extend** $\Pi^{pl}_{j,k}$ **to step** i
        inc ground $\Pi^{pl}_{j,i}$

        Set externals to reflect current search length.
        if Search step surpasses first ($i > j + 1$) **then**
            release external $query = i - 1$ **on** $\Pi^{pl}_{j,i}$
        assign external $query = i$ **on** $\Pi^{pl}_{j,i}$

        Attempt to solve the logic program.
        $Z^{pl}_{j,i} \leftarrow$ **solve** $\Pi^{pl}_{j,i}$

        if The program is satisfiable ($Z^{pl}_{j,i} \neq \emptyset$) (has an answer set for length i) **then**
            declare success
            $t\_total \leftarrow$ SYSTEM_TIME() - t_start
            return $\zeta^{pl}_{j,i,0} \in Z^{pl}_{j,i}$ **and** t_total

        else if Time limit has been reached ($t\_total \geq t\_limit$) **then**
            declare failure

        simplify $\Pi^{pl}_{j,i}$

    The search could not find an answer set in the given step range.
    declare failure

Chapter Five

Experiments

This chapter contains simulated experimental trials of the ASH implementation of the proposed HCR planning approach. The experiments determine if HCR planning has achieved the aims and objectives of this thesis given in Section 1.4. Specifically, the experiments answer whether HCR allows ASP based planning to: a) scale to problems with long plan lengths, and b) reduce the downtime of robots, for c) small losses in plan quality.

5.1 Experimental Setup

All experimental planning problems where ran in the BWP domain described in Section 3.4 whose abstraction hierarchy structure is depicted in Figure 3.2. The planning problems are solved as if the planner has never solved the domain or problem before, and has no prior experience to guide it. Experiments were run for each of; classical, HCR offline (h-offline), and HCR online (h-online), planning modes. For a given experimental configuration, classical and h-offline planning modes were always ran 25 times. Whereas, the h-online planning mode was ran 50 times, to account for the increased non-determinism of the planning algorithm that occurs from the problem division for online partial-planning. This gives reasonably sized samples to analyse consistency and non-deterministic properties of the planner.

5.1.1 Problem Instances

Three different experimental problem instances in the BWP domain were ran:

1. **P1**: In the first, all doors between the rooms start open, 4 blocks start in the puzzle room table and 2 in the store room, meaning that Talos only has to make one trip to the store room. The minimal ground-level plan length is 39 steps.

2. **P2**: In the second, the doors between the hallway and both the store and puzzle rooms start closed, requiring Talos to first open them to achieve the goal. The minimal ground-level plan length becomes 53 steps (increasing by 14 steps).

3. **P3**: The third extends the second, by moving block 4 to start in cell 0 of the store room (instead of on the puzzle room table), requiring Talos to now make two separate trips to the store room to move all 3 blocks to the puzzle room. The minimal ground-level plan length becomes 67 steps (increasing by another 14 steps).

The same entity constants were used throughout all problems to ensure the problem description size was unchanged. The minimal ground-level plan length for each problem is taken as the length of the shortest possible sequential action plan that can be found by classical ASP based planning. Plan quality is then determined by comparison of the number of actions in the plan to this optimal value as described in the following Sub-Section.

5.1.2 Performance Criteria

The following performance criteria are used to evaluate the performance of HCR planning[1].

The criterion for plan quality is always the quantity of actions in a plan. For sequential action plans, the number of actions in the plan is by definition the same as its length. For concurrent action plans, quality must be determined by the number of actions in the plan and not the length. This is because the worse case must be assumed, in which all concurrently planned actions will actually have to be executed sequentially by the robot.

[1]There are a great many experimental statistics produced by ASH, far too many to present in this thesis. For a full list of all raw results and log files from ASH see experiments results in Kamperis, 2023.

The performance criteria for planning time are as follows;

- The *total planning time* is the grand total time necessary to obtain a complete ground-level plan. For HCR planning, this is the sum of the planning times over all abstraction levels. For classical planning, this is simply the planning time at the ground-level.
- The *execution latency time* is the time taken to yield the first ground-level partial-plan in online planning. This is the time between the robot commencing planning and commencing plan execution. For classical and h-offline, this is equal to the total time.
- The *wait time* is the time the robot has to wait between the planner yielding any given ground-level partial-plan or action and the next partial-plan or action (if there is one).
- The *minimum execution time* is the time it would need to take the robot to execute a given partial-plan or action to exceed the wait time of the next, and therefore avoid downtime having to wait for the next partial-plan or action to be yielded by the planner.

In some cases, the ASP program grounding and solving times are split. The grounding time is the time taken to obtain the ground ASP program, indicating the complexity of the program. Solving time is the time taken to exhaust the search space, check for plan existence, and return any such plan, indicating the complexity of the problem search space.

To allow comparison of performance between different planning problems and configurations, variations on the aggregate scores used in the International Planning Competition (IPC) (Pommerening and A. Torralba, 2018) are used to form plan quality and planning time scores. These are combined to obtain an overall performance grade. The objective is to obtain a grade of 1.0, requiring a quality and time score of 1.0. Anything less than 1.0 is sub-optimal. The functions used to calculate the scores are hyperbolic and logarithmic. They are designed such that scores reduce very rapidly after exceeding the optimal plan quality or acceptable lag time. This heavily penalises even slightly sub-optimal performance.

- The plan quality score is $Q_s = \frac{C*}{C}$ where $C*$ is the optimal plan quality (the minimal

ground-level classical plan length for the given planning problem) and C is the number of actions in the plan being evaluated. This hyperbolic function causes decay in quality score which tends to zero as the number of actions in the plan approaches infinity.

- There are four time scores: the total time TT_s, the execution latency time LT_s, mean non-initial wait time WT_s, and mean minimum execution time per action MT_s scores. A time score is $VT_s = K - (ln(t - (K-1)) \,/\, ln(1800)) \times K$, if the planning time t (in seconds) is greater than K, otherwise the score is 1.0. This logarithmic function causes an exponential decay in time score which tends to zero as the time value approaches the planning time limit. Where 1800 is the planning time limit of half an hour in seconds and the variable K is the acceptable lag time per (partial-)plan or action. The acceptable lag time per (partial-)plan is 5 seconds. This is the maximum total time, execution latency time, or mean non-initial wait time, for the planner's performance to be optimal[2]. The acceptable lag time per action is 1 second. This is the maximum minimum execution time per action, for the planner's performance to be optimal[3].
  - The overall time score is then denoted by T_s. For classical and h-offline planning, the total time score is the overall time score $T_s = CT_s$. For h-online planning, the mean of the latency, non-initial wait, and minimum execution time per action scores, is the overall time score such that $T_s = \frac{1}{3}\{LT_s, WT_s, MT_s\}$.
  - The execution latency time score measures the planner and robot's lag[4]. The average wait time score measures the planner's ability to remain consistent in yielding partial plans after the initial lag. The mean minimum execution time per action score measures the ability to avoid possible downtime of the robot.
- The performance grade G measures the trade-off between planning time and plan quality. The grade is the product of plan quality and overall time score $G = Q_s \times T_s$.

[2]This is therefore the maximum total; completion, latency, or wait time, that we are willing to wait before; the robot can commence plan execution, or between finding one partial-plan and the next, respectively.

[3]This is the maximum time that we are willing to wait between yielding one action and the next.

[4]The assumption of the execution latency time is that the planner has already generated enough actions to occupy it for long enough to obtain the next partial plan before it runs out of actions to execute.

5.1.3 Non-Determinism

The following results indicate spread in timing results, even when generated plan lengths are the same. This is because ASP solvers are non-deterministic and there is no guarantee that the search will happen in the same way or that the same solution will be obtained every time. For classical ASP based planning, it is expected that the non-determinism of search will cause significant variance in planning time between different experimental runs. For HCR planning, it is expected that the conformance constraint will reduce the variance in planning time over classical. This is expected because the sub-goal stage sequence that forms the conformance constraint is intended to guide search towards the final-goal.

5.1.4 Statistical Representations

The median is used as the statistical measure of the central tendency, and the upper and lower quartiles as the measure of the spread of the results over all repeated trails for the same experimental configuration[5]. The timing statistics are obtained from Python's highest resolution performance clock and are the real time to make a call to the ASP system[6].

Results are presented primary by graphical plots. Bar plots show the median value and include error bars indicating the 50% confidence interval over all experimental runs for the given configuration. Box plots instead show the median, upper and lower quartiles, the maximum and minimum values, and any outlier values over all runs. Histogram plots show the density of runs as a percentage that obtained the given statistical value for the given configuration. Line plots show the median value and the 50% confidence interval around the median. Regression plots do not include the confidence interval as it is not meaningful.

The following statistical tests of significance between measurements in different data

[5]The mean and standard deviation is not used as there is no guarantee the data is normally distributed.

[6]Time taken to construct Python objects that encapsulate answer sets and to parse those answer sets, is not included in the timing statistics, since they do not represent any meaningful property of the planner.

sets are used to compare the relative performance between different experimental configurations. The Kruskal-Wallis H-test is used as for global comparisons between many data sets. It tests the null hypothesis that the population median of all of the data sets are equal (it is a non-parametric version of ANOVA). The Mann-Whitney-Wilcoxon signed rank test is used for pair-wise comparisons between two different but related data sets. It tests the null hypothesis that the distribution of the differences between two paired data sets is symmetric around zero (it is a non-parametric version of the paired T-test). If the null hypothesis is true, then this implies that the data sets are drawn from the same distribution. In both cases, the null hypothesis is rejected if the test indicates less than 5% probability of its truth. In this thesis, this is taken as sufficient evidence to support the alternative hypothesis, that a significant difference in the median or distribution of the data sets exists[7].

5.1.5 Hardware and Software Specifications

All experiments were run on the same desktop computer[8], running Microsoft Windows 10 Home 64-bit (10.0, Build 19043) operating system, AMD Ryzen 5 3500X 6-Core Processor @ 3593 Mhz, 16384MB 3000Mhz RAM, and ASUSTeK PRIME A320M-K motherboard.

Python's default CPython version 3.9.3 and the standard Clingo ASP system version 5.5 was used by ASH. Clingo was always invoked in parallel-mode, with 6 competing threads. All other settings for Python and Clingo were default. Pandas version 1.4.4 was used to store results generated by ASH, Xlsxwriter version 3.0.3 to save results in Microsoft Excel files, and Matplotlib version 3.5.2 and tikzplotlib 0.10.1 were used to generate plots.

All timing statistics were gathered using Python 3.9's highest resolution performance clock. The time taken to parse answer sets and extract plans and conformance mappings are not included in timing statistics, as it is not a meaningful property of HCR planning.

[7]This does not mean that the alternative is accepted. It does not prove the data are different, only that there is low probability (less than 5%) that the difference in the data occurred by random chance.

[8]All efforts were made to ensure ASH was the only user process running during experiments.

5.2 Results and Evaluation

This Section presents a variety of experimental trails. To prevent fragmentation, the results and evaluation of each test set are combined, and each set accepts or rejects some hypothesis.

5.2.1 Overview and Structure of Experimental Test Sets

Four test distinct sets are presented in the following Sub-Sections as summarised in 5.3.

Firstly, the ASH planner's performance is tested between the different planning modes; classical, h-offline, and h-online with the Basic problem division strategy and ground-first online planning methods. The intention of this test set is to determine whether the HCR planning paradigm can outperform classical ASP based planning. The hypothesis is that HCR planning will reduce planning time for small losses in plan quality, and scale better to problems with longer minimal plan lengths over classical planning. Furthermore, it is expected that h-online planning will out-perform h-offline planning, with lower total planning and execution latency times, but the chance of dependencies between partial-problems is likely to reduce plan quality of h-online over h-offline. The following then evaluate the properties that affect the performance of different HCR planning configurations.

Secondly, h-offline is tested and compared for all combinations of: simultaneous (sima) and sequential (seqa) sub-goal stage achievement, and all search modes; standard, minimum search length bound enabled (min-bound), and sequential yield (seq-yield). The intention is to determine which attains the best performance. The hypothesis is that these should only affect planning time and there is no reason to believe they will affect plan quality.

Thirdly, h-online with the Basic problem division strategy is tested for all possible combinations of the bounds 2, 4, and 6. This is such that; the possible bounds are $(J, K) \mid J \in \{2, 4, 6\}, K \in \{2, 4, 6\} : J \neq K$, where J is the bound value for the strategy at abstraction

level 2 and K for level 1. Where the bound values of Basic are the number of partial-problems n to divide a combined refinement problem into. The intention is to determine how the bound values affect planning time and plan quality. The hypothesis is that a large number of bounds will reduce planning time but is more likely to reduce plan quality. This is because large bound values, result in more problem divisions, which it is likely to skew the distribution of plan lengths positively, due to the increased chance of problem dependencies occurring. The effects are then evaluated of: final-goal pre-emptive achievement, concurrent action planning, saved program groundings, and partial-problem blending. This is to determine how these additional proposed techniques influence the performance of h-online planning.

Finally, h-online with the size-bound based strategies Hasty and Steady is tested for all possible combinations of the bounds 2, 4, and 6, where the bounds values at lower abstraction levels were not allow to be larger than those at the higher levels. This is such that; the possible bounds are $(J, K) \mid J \in \{2, 4, 6\}, K \in \{2, 4, 6\} : J \geq K$. Where the size-bound values of Hasty and Steady are the size in number of sub-goal stages in each partial-problem. This evaluates whether the move dynamic decision making of the size bound strategies will achieve between balance in problem sizes, and resultantly achieve better and more consistent plan quality. These are also tested for all online planning methods; ground-first, complete-first and hybrid. This is to evaluate trade-offs between time scores and plan quality score for each of these methods. It is expected that: ground-first will achieve the best time scores, due to it prioritising low latency and wait times; complete-first will achieve the best quality scores, due to it only dividing complete refinement problems to reduce imbalance in problem sizes, and; hybrid will achieve a middle-ground between the two other methods.

5.2.2 Performance Comparison of Different Planning Modes

Figure 5.1 contains box plots of the distribution of performance grades and scores for each planning mode and planning problem for sequential and concurrent action planning. H-offline

includes data for all sub-goal achievement types and search modes, and h-online includes all decision bound values for the Basic division strategy and ground-first online method.

Tables 5.1 and 5.2 contain the median grounding time (GT), solving time (ST), total planning time (TT), and the ground-level number of planned actions (AC), for each planning mode, for sequential and concurrent action planning. Where the timing values for HCR planning are the sum over all abstraction levels, and for classical are just the ground-level times. Tables 5.3 and 5.4 contain the aggregate planning times; execution latency time (LT), mean wait time per partial-plan (WT), total planning time (TT), and the ground-level number of planned actions (AC); for h-online and h-offline planning modes as the percentage of the respective value for classical, for sequential and concurrent action planning.

For sequential action planning, the plots in Figure 5.1 indicate that h-online has the best median performance grades and classical has the worst for all problems. Classical and h-offline produce a tight spread around the median grades, whereas h-online produces visibly much higher spread particularly for P2. Quality scores for classical are of course always optimal, and h-offline achieves good median quality scores of at least ≈ 0.92 in all cases. Whereas, h-online rarely achieves optimal plan quality with the lowest average and minimum quality scores, but was at least ≈ 0.90 in all cases. In the best case, plan quality for h-online was greater than or equal to the average case for h-offline. But in the average and worst case, plan quality for h-online was always less than those of h-offline, except for problem P1. Time scores for classical are the worst for all problems. H-offline performs notably better but is still sub-optimal. Whereas, h-online performs excellently, consistently achieving optimal time scores, and even in the worst case of P3 (the most complex problem) achieving a score of at least ≈ 0.90 which is far greater than the best case for h-offline and classical for all problems. Whilst median quality scores for classical and h-offline are similar between problems, their time scores are drastically different and clearly decrease substantially as the problem size and minimal plan length increases. This indicates that the planning time of classical and h-offline are highly dependent on the problem size and

minimal plan length. However, for both quality and time scores, classical and h-offline have a tight spread about the median for all problems. This leads to the low spread in their grades, indicating that classical and h-offline perform consistently. In contrast, h-online has similar medians for both scores between problems, and low spread in time scores, but high spread in quality scores. This leads to the high spread in its grades, indicating that h-online performs relatively inconsistently.

For sequential action planning, Tables 5.1 and 5.3 indicate that the median total, solving, and grounding times were always lower for HCR planning over classical. Total times for h-offline were between 74% and 9% and for h-online between 58% and 2% of classical, from the easiest to hardest problems. The reduced solving time for h-offline confirms the expectation that conformance constraints restrict the search space. The even greater reduction for h-online confirms the expectation that partial-planning reduces problem complexity. Specifically, the median total planning times for classical at 13.5, 38.2, and 607.0 seconds, and for h-online at 7.9, 10.6, and 14.1 seconds, for problems P1, P2, and P3 respectively, indicate an exponential reduction in planning time for h-online over classical as the plan length of a problem increases. The execution latency time for h-online was between 19% and 1% of classical, which is less than 7 seconds in all cases. The wait time for h-online was between 5% and 1% of classical, which is less than 3 seconds in all cases. This confirms the expectation that h-online partial-planning greatly reduces execution latency and average wait times over complete h-offline or classical planning.

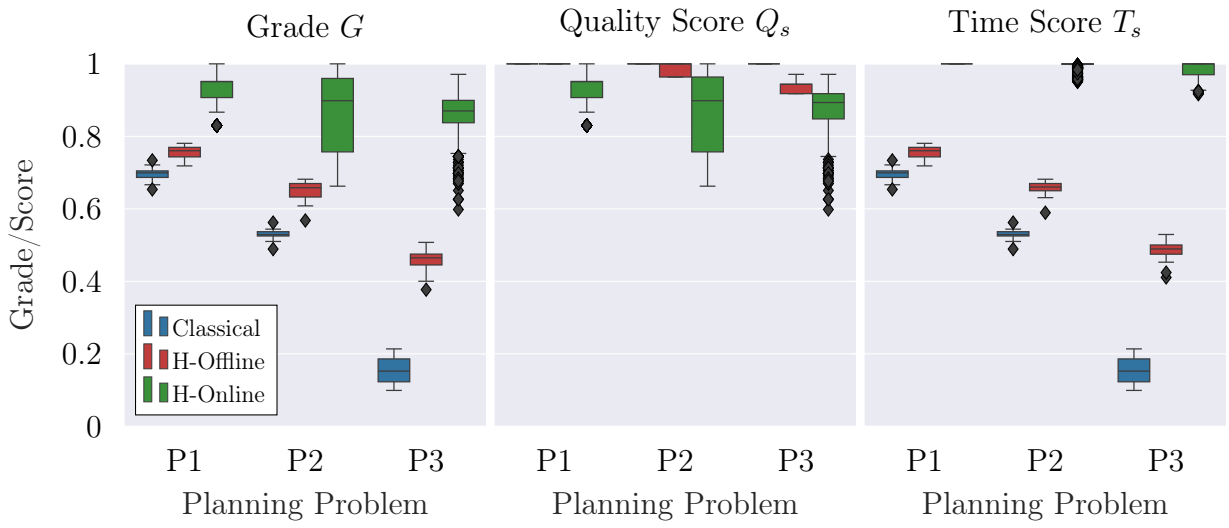
For concurrent action planning, the plots in Figure 5.1 indicate again that h-online has the best median performance grades. However, the performance grades for classical are better than h-offline for problems P1 and P2, and only for P3 does classical perform the worst. Classical and h-offline produce an even tighter spread around the median grades compared with sequential action planning. Whilst h-online continues to produce higher spread, it is reduced over sequential action planning, again particularly for P2. Quality scores for classical are no longer optimal at ≈ 0.98 in all cases. H-offline again achieves good median quality

scores of at least ≈ 0.90. Whereas, h-online continues to have the lowest quality scores, with notably worse best case quality scores, that are always less than or equal to the worst case for h-offline. It was expected that quality scores for HCR planning would improve under concurrent action planning. The conjecture was that it provides more specific information about the ordering over achieving sub-goals than sequential, see Section 3.7.3 for details and examples. However, this is not reflected in the results. The cause of non-optimal and generally lower quality concurrent action plans over sequential even for classical is discussed in Section 5.2.7. Unlike sequential action planning, the time scores for classical exceed that of h-offline for P1 and P2, leading to its better grades in the same cases. However, h-offline does again greatly outperform classical for P3. Whereas, h-online continues to perform excellently in all cases. Notably, h-online time scores for concurrent action planning are slightly worse than sequential action planning. This implies that: the improvement in planning speed attained by online partial-planning, caused by solving a sequence of problems with linearly shorter plan lengths, is greater than that attained by concurrent action planning, caused by the linear compression of concurrent action plans. This is likely because partial-planning is already sufficient to achieve optimal time scores, and cannot be further improved by concurrent action planning. The additional time needed to ground the rules that handle the concurrency conditions may be the cause of the worse time scores. The median quality scores for classical and h-offline continue to be similar between different problems, but their time scores continue to decrease as the problem size and minimal plan length increases. This indicates that concurrent action planning is not sufficient to make classical or h-offline able to handle arbitrarily large problems. However, for both quality and time scores, classical and h-offline have an even tighter spread about the median for each problem compared with sequential action planning, indicating that concurrent action planning does make classical and h-offline performance more consistent overall. H-online continues to have similar medians for both scores between problems, and low spread in time scores. However, its spread in quality scores is notably lower than for sequential action planning, particularly for P2. This leads to lower spread in grades, indicating that concurrent action planning also makes h-

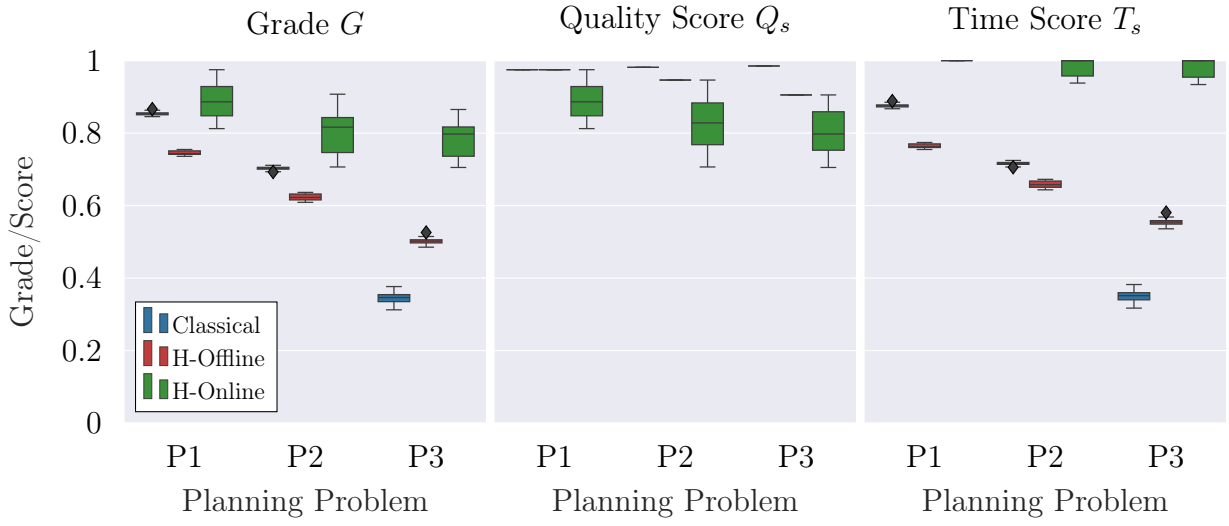
online performance more consistent overall.

For concurrent action planning, Tables 5.2 and 5.4 indicate that the median total, solving, and grounding times lower for HCR planning over classical only for problem P3. Total times for h-offline were between 152% and 24% and for h-online between 153% and 14% of classical, from the easiest to hardest problems. The latency time for h-online was between 39% and 3% of classical and the wait time was between 13% and 1% of classical. Whilst median total planning times are not reduced for concurrent action planning to the extent that they are for sequential action planning, h-online is still able to greatly reduce execution latency and average wait times over h-offline and classical. Note that concurrent action planning has higher grounding times for all problems and planning modes than sequential action planning. This supports conclusion that there is increased time complexity involved in grounding the rules that enforce the concurrency constraints.

The immediate conclusions from this test set are as follows. H-online performs better on average than classical and h-offline for all problems, but is significantly less consistent. Whilst h-online's time scores are excellent, often optimal, and highly consistent for all problems, its quality scores perform the worst, and appear to vary sufficiently enough that the spread of its grades becomes the largest overall. Despite this, the quality of h-online plans are still good, and given its excellent performance grades, the losses appear to be an acceptable trade-off given the substantial increase in planning speed exhibited. Concurrent action planning does make plan quality scores more consistent for classical, h-offline, and h-online, but also leads to a reduction in average quality. Further, whilst concurrent action planning improves time scores for classical and h-offline, this improvement is not sufficient to make them optimal for any problem, and does not improve scalability to the more complex problems by a meaningful amount. In contrast, h-online experiences no significant improvement in time scores under concurrent action planning, this is explored further in Section 5.2.4.



(a) Planner Performance Grades and Scores for Sequential Action Planning.



(b) Planner Performance Grades and Scores for Concurrent Action Planning.

Figure 5.1: Planner Performance Grades and Scores for each Planning Mode, Planning Problem, and Action Planning Type.

Table 5.1: Raw Planning Times in Seconds and Number of Planned Actions for Sequential Action Planning.

Problem	P1				P2				P3			
Mode	GT	ST	TT	AC	GT	ST	TT	AC	GT	ST	TT	AC
Classical	7.3	10.5	13.5	39	10.5	34.6	38.2	53	15.0	592.0	607.0	67
H-Offline	6.2	3.8	10.0	39	9.2	7.4	16.8	53	14.3	36.2	50.0	71
H-Online	6.4	1.5	7.9	41	8.4	2.2	0.9	59	10.7	3.2	14.1	75

Table 5.2: Raw Planning Times in Seconds and Number of Planned Actions for Concurrent Action Planning.

Problem	P1				P2				P3			
Mode	GT	ST	TT	AC	GT	ST	TT	AC	GT	ST	TT	AC
Classical	8.9	1.2	6.5	40	13.0	3.7	12.4	54	19.3	120.1	139.4	68
H-Offline	8.7	1.3	9.9	40	12.9	4.1	17.0	56	18.7	13.5	32.2	74
H-Online	9.1	0.9	10.0	44	13.7	1.4	14.7	64	17.7	1.8	19.5	84

Table 5.3: Aggregate Planning Times in Seconds and Number of Planned Actions for Sequential Action Planning as Percentage of Classical Planning.

Problem	P1				P2				P3			
Mode	LT	WT	TT	AC	LT	WT	TT	AC	LT	WT	TT	AC
H-Offline	74	74	74	100	44	44	44	100	9	9	9	106
H-Online	19	5	58	105	10	2	28	111	1	1	2	112

Table 5.4: Aggregate Planning Times in Seconds and Number of Planned Actions for Concurrent Action Planning as Percentage of Classical Planning.

Problem	P1				P2				P3			
Mode	LT	WT	TT	AC	LT	WT	TT	AC	LT	WT	TT	AC
H-Offline	152	152	152	100	137	137	137	104	24	24	24	109
H-Online	39	13	153	110	31	10	119	119	3	1	14	124

Level-wise Number of Planned Actions between Planning Modes

Figure 5.2 contains bar plots that break-down the number of planned actions over all abstraction levels, for each planning mode[9], planning problem, and action planning type. The plots indicate that, as expected, all plans contain the same number of actions at the classical planning level of 3. Interestingly, in all cases, the median plan lengths at level 2 are also always the same for h-offline and h-online, except for problem P3 under sequential action planning where h-offline and h-online have 4 more actions than classical. In contrast, at level 1, h-offline and h-online typically produce lower quality plans with more actions than classical. For sequential action planning, h-online is never optimal, with its plans containing up to 10 more actions than classical. Whereas, h-offline is optimal for all problems except for P3 where its plans contain 5 more actions than classical. For concurrent action planning, h-online performs even worse, with its plans containing up to 18 more actions than classical. Whereas, for h-offline only performs optimally for problem P1, with P2 and P3 containing up to 8 more actions than classical. It is clear therefore, that the ground-level of 1 reached by refining from the condensed domain model at level 2, has the biggest influence on the reduction in plan quality that occurs during plan refinement. The specific reasons causing sub-optimality of HCR plans are described and discussed later in Section 5.2.6.

Level-wise Planning Times between Planning Modes

Figure 5.3 contains bar plots that break-down the total planning time spent at all abstraction levels, for each planning mode, planning problem, and action planning type. The plots indicate that, as expected, that planning time at the most abstract classical planning level of 3 is always the same for all planning modes, and is much less than the amount of time spent planning at the lower abstraction levels. This confirms our expectations that abstract

[9]In HCR planning, all monolevel plans at each level are generated as part of the same hierarchical problem. In classical, the monolevel plan at each level is generated as an independent monolevel problem.

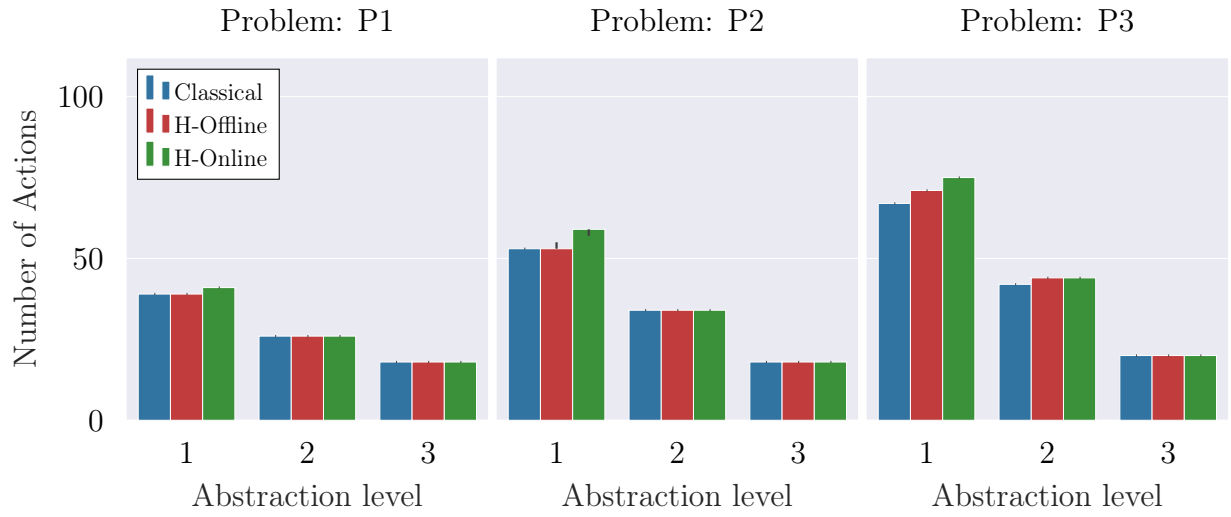
models can reduce planning time exponentially over their original model. For sequential action planning, a clear exponential reduction in planning time can be see for h-offline and h-online over classical at abstraction level 1. This is most noticeable for the problems with longer minimal plan lengths, indicating again that the benefit of HCR planning becomes more pronounced when scaling to larger problems. At level 2, the reduction in planning time for h-offline and h-online planning over classical is small for problems P1 and P2. This suggests that HCR planning is most beneficial when plan lengths are long. In contrast, for concurrent action planning, the exponential reduction in planning time for h-offline and h-online over classical is only clearly observable for P3. Otherwise, no meaningful difference in planning time between the modes for P1 and P2 exists. This indicates that the compression on plan lengths achieved by concurrent action planning can make classical and h-offline planning competitive with h-online for small problems. However, is still not sufficient to achieve arbitrary scalability to large problems. The planning time for h-online is greater when using concurrent action planning. This indicates that it does not provide additional benefit over the time saved by partial-planning, which already achieves optimal time scores.

Step-wise Search Times between Planning Modes

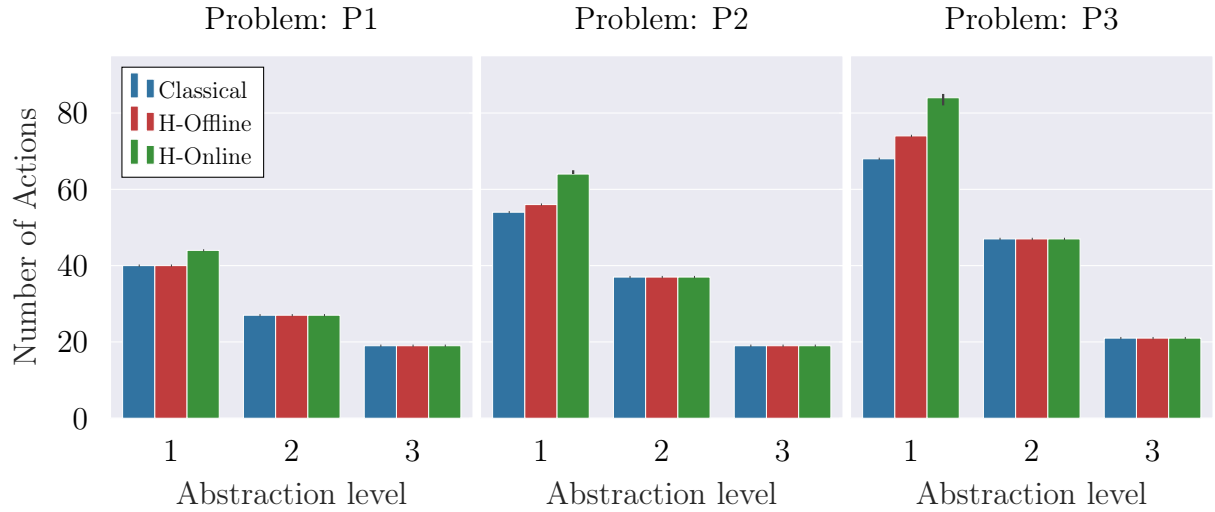
Figure 5.4 contains exponential regression line plots of total search time per search length for each planning mode, planning problem, and action planning type. Note that the Y axis scale is different between the action planning types. The plots indicate clearly that both classical and h-offline display exponentially increasing search times with search length. Classical search times exceed that of h-offline in all cases except P1 and P2 with concurrent action planning. This is expected given that classical's time score is better than h-offline in these cases. Particularly for problem P3, the exponential trend for classical presents an extremely steep curve compared to h-offline. H-offline in contrast displays a relatively shallow and close to linear trend for the majority of the early search length. H-online however, displays a linear increase in search time for the entire search length for all cases. Relative to classical

and h-offline, the search time of h-online appears to remain constant with search length, particularly for sequential action planning. However, the maximum search length for h-online is greater than that of classical and h-offline in all cases. This is because it produces longer and lower quality plans, indicating that although h-online experiences lower search times per step, the trade-off is that it has to search much farther than classical and h-offline.

These results therefore support the expectations that: search times for classical ASP based planning increase exponentially with search length, as was claimed in past work (Jiang et al., 2019); h-offline planning can heavily reduce search times due to the existence of the conformance constraint, but search times still increase exponentially with search length; and that h-online planning does reduce search times from increasing exponentially to increasing linearly with search length, an exponential reduction over classical and h-offline. This confirms the expectation that solving a sequence of partial-problems can reduce overall problem complexity, from exponential down to linear, in the minimum length plan.

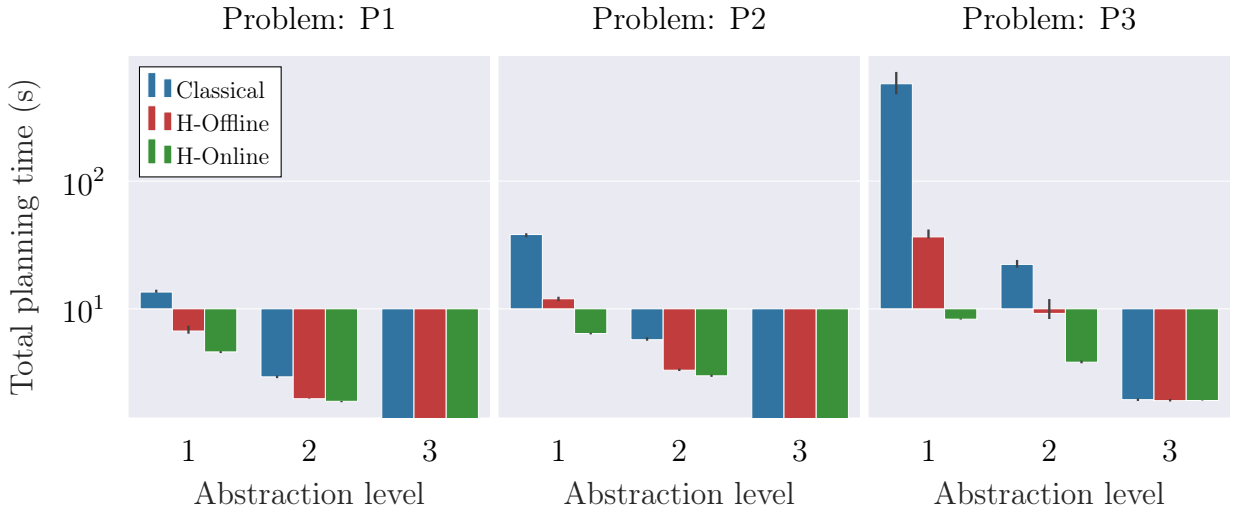


(a) Level-Wise Number of Actions for Sequential Action Planning.

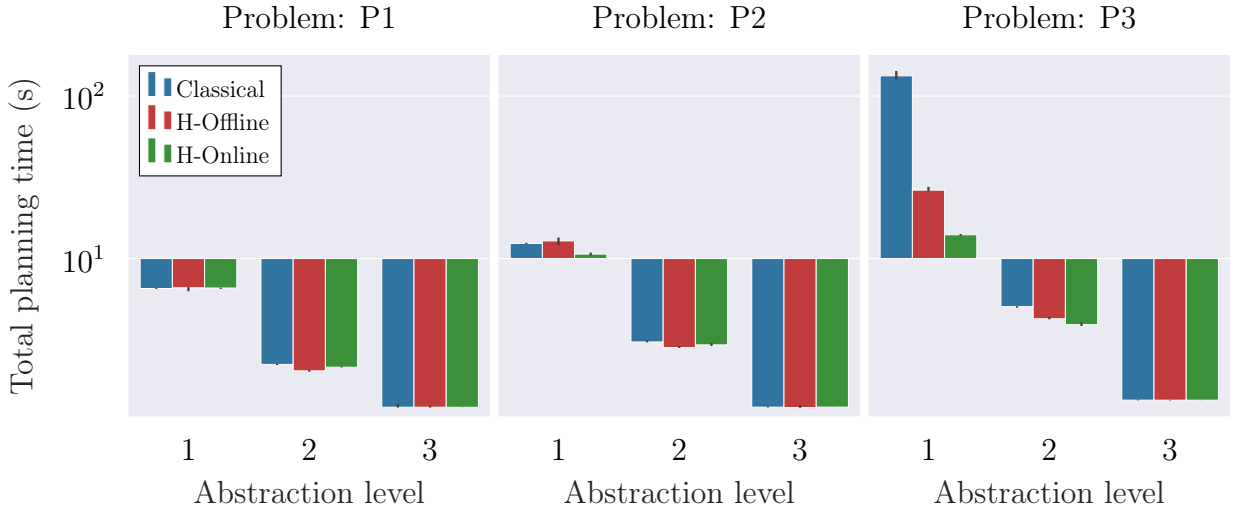


(b) Level-Wise Number of Actions for Concurrent Action Planning.

Figure 5.2: Level-Wise Number of Actions for each Planning Mode, Planning Problem, and Action Planning Type.

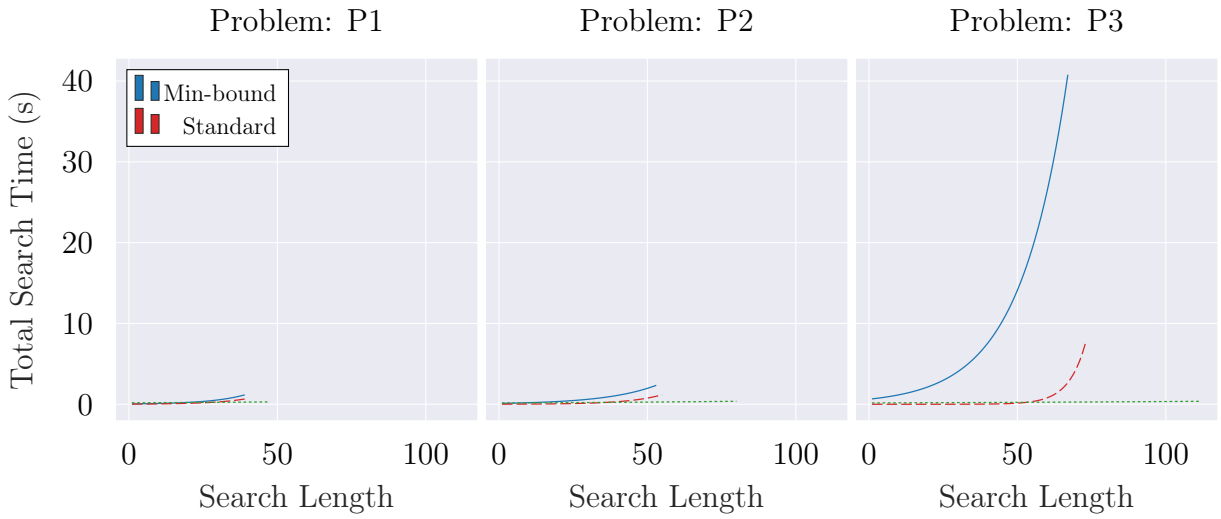


(a) Level-Wise Planning Times for Sequential Action Planning.

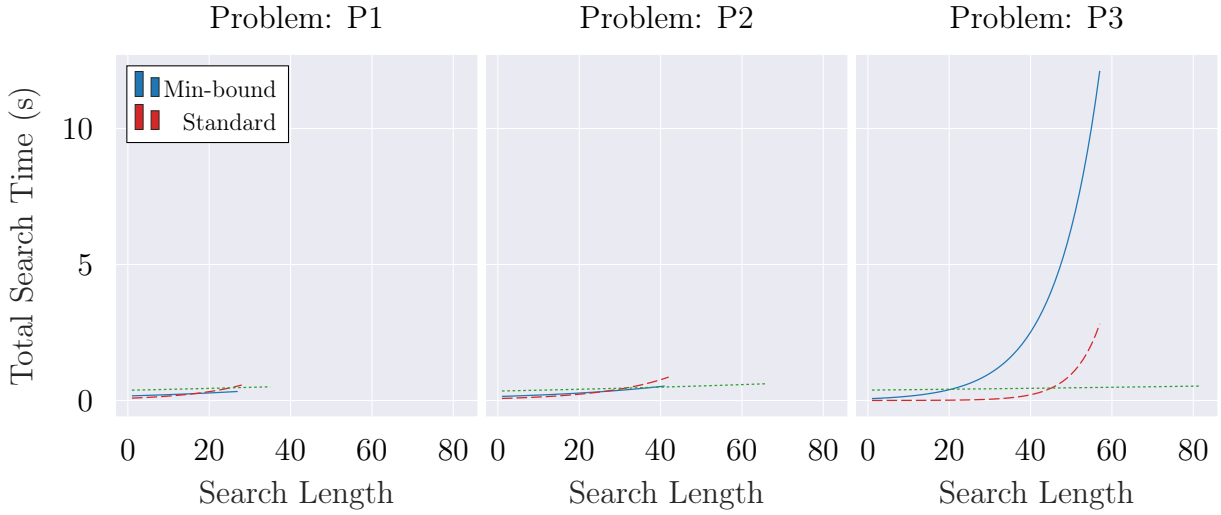


(b) Level-Wise Planning Times for Concurrent Action Planning.

Figure 5.3: Level-Wise Planning Times for each Planning Mode, Planning Problem, and Action Planning Type.



(a) Total Search Time per Search Length for Sequential Action Planning.



(b) Total Search Time per Search Length for Concurrent Action Planning.

Figure 5.4: Exponential Regression Line Plots of Total Search Time per Search Length for each Planning Mode, Planning Problem, and Action Planning Type.

5.2.3 Affect of Sub-Goal Stage Achievement Type and Search Mode

The following evaluates the affects of sub-goal stage achievement type and search mode using h-offline and sequential action planning. Their observed performance are assumed to hold in general, and the best performing type and mode are used in all further experiments.

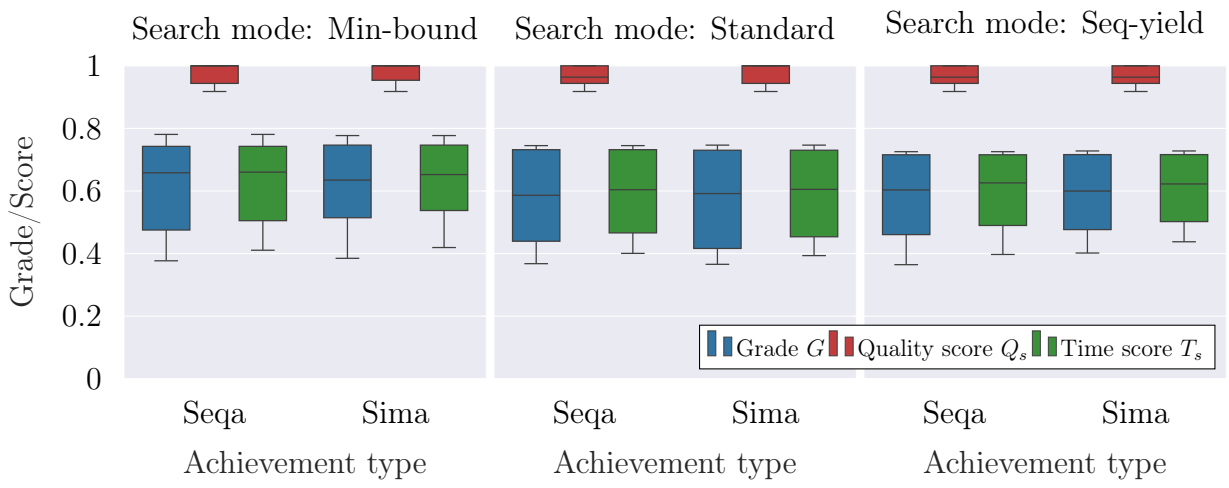
Figure 5.5a contains box plots of the grades, time scores, and quality scores, for each search mode and achievement type taken as median over all planning problems. The plots indicate almost identical values for medians, quartiles, and ranges, for both sub-goal stage achievement types given a particular search mode. Only for sequential yield search is there a very minor benefit in the minimum time score for simultaneous achievement. Paired tests for the same search mode but different achievement type do not indicate statistically significant differences between the median time scores of different achievement types. As such, only sequential achievement will be used throughout the rest of these experimental studies for this reason, and because it is necessary in concurrent action planning. However, a clear limitation of this evaluation is that most sub-goal stages contain exactly one sub-goal literal in the BWP problems, because the vast majority of actions only have one direct effect. Since at least one sub-goal literal has to be uniquely achieved in the end state of a sub-plan, regardless of achievement type, a sub-goal stage containing exactly one literal is always achieved in the same way (see Section 3.7.2 for details). The nature of the conformance constraint is therefore similar for both achievement types. Future work should seek to perform this evaluation for planning problems where more actions have multiple direct effects.

Figure 5.5b is a bar plot of median overall planning time score for each search mode and small problem. The scores indicate that min-bound is the best performing search mode for all problems, scoring ≈ 4 to 5.5% higher than the next best performing mode. Whereas, seq-yield performs better than standard by ≈ 1.5 to 2.5 for all problems except P1. Paired tests for the same problem but different search mode indicate statistically significant differences between the median time scores of different search modes. Further, the confidence

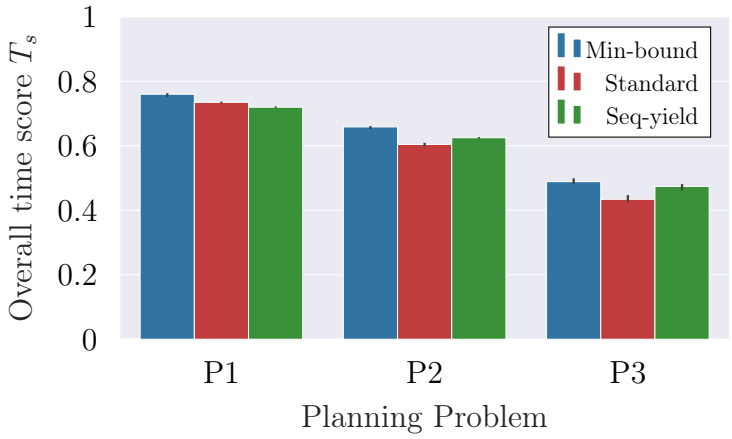
interval of time score for all search modes and problems was always less than or equal to 0.06 and less than 10% of the total time score, indicating consistent performance.

Figure 5.5c is a bar plot of the sum over all abstraction levels of the grounding and solving times as a percentage of the total planning time for each search mode, taken as the median over all achievement types and planning problems. The plot indicates that for standard and min-bound search modes, the percentage contribution of grounding and solving times is very similar at ≈ 52 and ≈ 48 respectively, whereas seq-yield shows $\approx 10\%$ greater grounding time and lesser solving time. The difference between min-bound and standard is not statistically significant but the difference between them and seq-yield is significant.

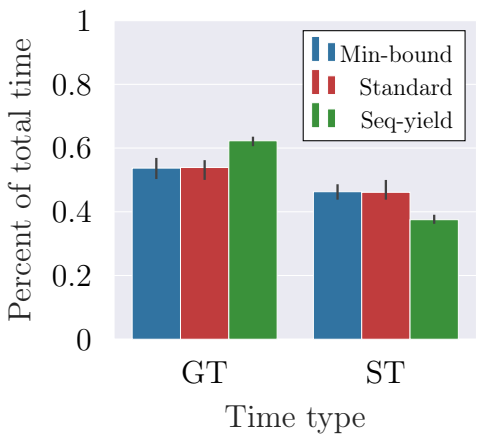
Figure 5.5d is a relational line plot of total search time per search length for each search mode and refinement abstraction level on small problem P3 only. The plots indicate that for both levels 1 and 2, the standard and min-bound search modes both display smooth exponential trends. The majority of the search time occurs near the end of the search length. Except for some notable early spikes, the total search time for min-bound however remains slightly below standard by $\approx$ 1s, for all but the last ≈ 3 steps of abstraction level 2. The early spikes occur when the minimum search length bound is reached on average, and indicates the cumulative time taken to ground the program at all steps below the bound. Below the bound, the search time per step is zero, since search does not actually occur on those steps. For abstraction level 1, two distinct early spikes exists, because there are multiple possible abstract plan lengths at level 2. In contrast, seq-yield planning mode has no clear underlying trend, for either level, with erratic changes and large spread in the search time with search length. This indicates significantly greater non-determinism in the time performance of seq-yield planning mode. At abstraction level 2, seq-yield's search time is lower than both standard and min-bound at all steps. However, at abstraction level 1, seq-yield has higher times than both standard and min-bound until step ≈ 65. After this step, the total search time for seq-yield remains lower than standard and min-bound, and does not experience the rapidly exponentially increasing search times that standard and min-bound do.



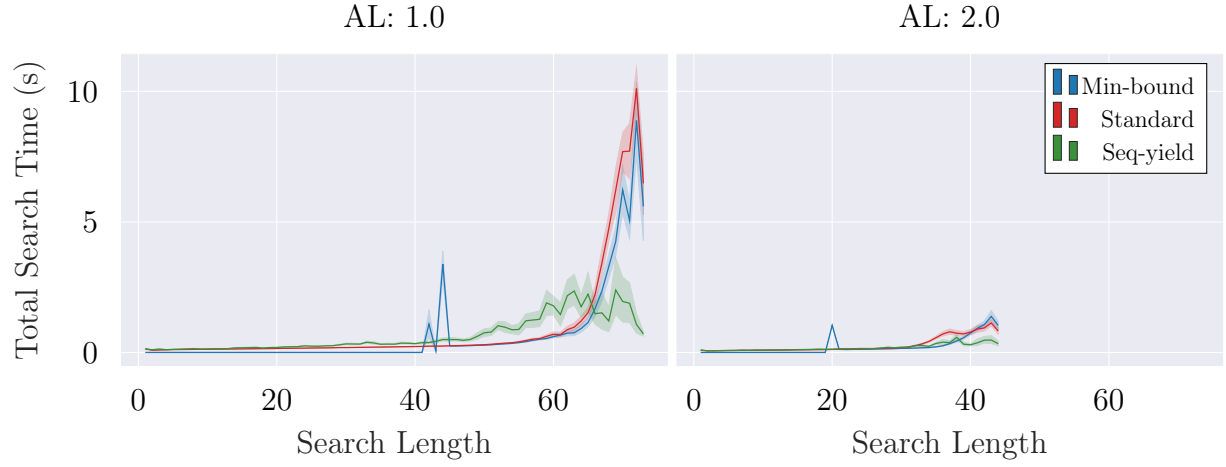
(a) Planner Performance Grades and Scores.



(b) Planning Time Score.



(c) Percent Times.



(d) Total Planning Time per Search Length.

Figure 5.5: Planner Performance for different Search Modes and Achievement Types.

5.2.4 Online Planning with the Basic Division Strategy

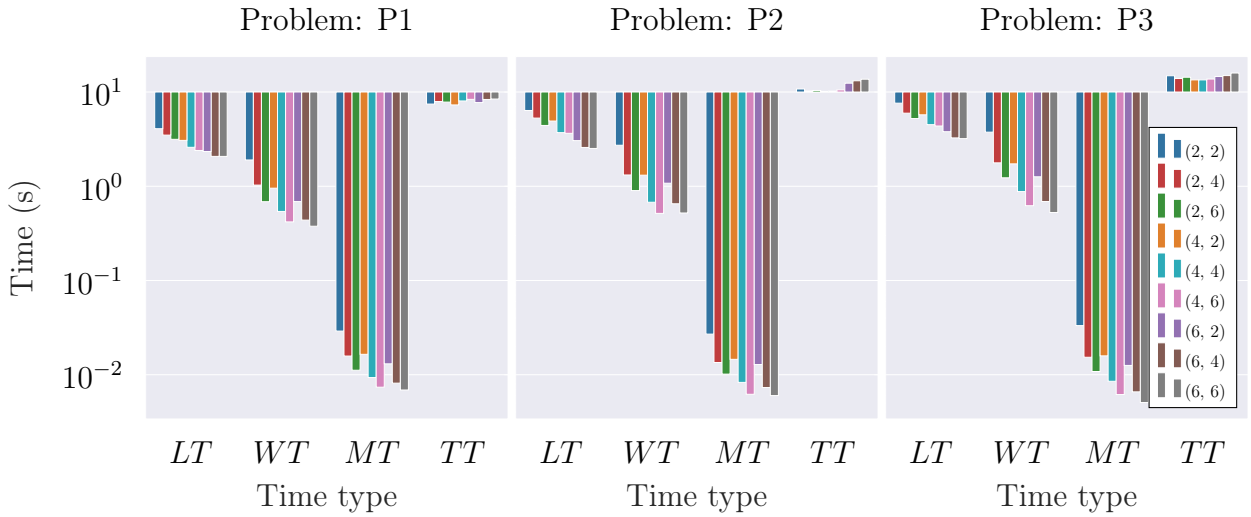
This section first evaluates the performance of different decision bound values for the Basic problem division strategy on all planning problems, with sequential action planning, sequential sub-goal stage achievement, and min-bound search mode. The following sub-sub-sections then evaluate how: concurrent action planning, final-goal pre-emptive achievement, saved program groundings, and partial-problem blending affect planner performance.

Figure 5.6a contains box plots of the grades, time scores, and quality scores, for each planning problem and decision bound value combination $(2, 2)$, $(2, 4)$, $(2, 6)$, $(4, 2)$, $(4, 4)$, $(4, 6)$, $(6, 2)$, $(6, 4)$, and $(6, 6)$. The plots indicates clearly that different bound values do lead to different median and spread of quality scores. This confirms the expectation that the quantity and size of partial-problems does have an impact on plan quality. Simultaneously, different decision bounds that produce the same number of ground-level partial problems can result in different plan quality. The bound value combinations producing the best median and lowest spread in quality scores are $(2, 2)$, $(2, 4)$, $(4, 2)$, $(4, 4)$, with the absolute best being $(4, 2)$. The larger bound values decrease quality scores for problems P2 and P3. This indicates that diminishing returns in terms of time score as the total number of problem divisions increases. Even a small number of divisions is sufficient to exponentially reduce planning complexity over classical and h-offline planning with minimal reduction of plan quality. This substantial difference in quality scores for different bound values explains the spread in the overall quality scores of h-online planning for any given planning problem that were observed in Section 5.2.2. All bound values give optimal time scores for P1 whereas for P2 and P3 only the high bound values are optimal but all are above ≈ 0.92.

Figure 5.6b contains bar plots of median execution latency time (LT), average non-initial wait time per partial-plan (WT), average minimum execution time per action (MT), and total time (TT), for all planning problems and decision bound value combinations. The plot indicates that higher decision bound values can lead to lower execution latency, average

wait, and minimum execution times. Specifically, bounds $(4, 6)$ and $(6, 6)$ produce the lowest wait and minimum execution times for all problems, followed closely by $(4, 4)$ and $(6, 4)$. Note that all bounds with 2 in them perform relatively poorly. This explains the better times scores for the higher bound values of P3, again confirming the expectation that for problems with long plan lengths, more divisions are necessary to achieve optimal time scores. For the ground-first online planning method, smaller partial-problems allow the planner to propagate to and yield ground-level partial-plans faster and more frequently.

(a) Planner Performance Grades and Scores for each Planning Problem.



(b) Aggregate Planning Times for each Planning Problem.

Figure 5.6: Planner Performance for the Basic Strategy and Ground-First Online Method.

Problem and Plan Balancing

Figure 5.7 contains box plots of: median partial-problem size factor s_m, partial-plan expansion factor θ_m, partial-problem balance deviation μ_d, partial-problem balance error μ_e, partial-plan balance deviation β_d, and partial-plan balance error β_e; for each all planning problems taken as the median over all bound values[10]. The plot indicates that at abstraction level 2 the Basic strategy achieves good partial-problem and plan balancing, with less than ≈ 0.25 problem balance deviation and error, and ≈ 0.8 plan balance deviation and error, with low spread around the median values, for all problems. In contrast, at level 1 the Basic strategy shows greatly increased median and spread in partial-problem and plan balance errors for all problems. For P2 and P3 the problem balance error at level 1 is ≈ 10 times greater than level 2. However, the median partial-plan balance deviation is lower at level 1 than level 2 for all problems. This discrepancy between the balance deviation and error may exist because the deviation assumes the length of partial-plans is normally distributed, which there is no reason to believe is ture, and is therefore less robust than the balance error. The balance deviation is however less sensitive to extreme values than the balance error, and therefore the discrepancy may be caused by high positive skew in partial-plan lengths.

The expectation was that the highly naive decision making of the basic problem division strategy and ground-first online planning method may typically lead to an accumulation of errors in problem and plan balance descending the hierarchy. This is because the imbalance in the length of sub-plans means that a sequence of homogeneous size partial-problems is unlikely to refine to a sequence of homogeneous length partial-plans. When those non-homogeneous partial-plans are each sub-divided independently and then further refined again, they will likely each produce greatly different partial-plan lengths. However, due to the discrepancy between the balance deviation and error statistics, it is not possible to accept or reject this. Future work should seek to explore the cause of this discrepancy.

[10]Note that the partial-plan size factor is the median over all partial-problems of the ratio of the partial-problem size to the complete-problem size at that abstraction level. See Section 3.8 for the definition of expansion factor and Section 4.1.1 for the definition of the balancing criteria.

Final-Goal Pre-emptive Achievement

Figure 5.8a contains box plots of the grades, time scores, and quality scores, with final-goal pre-emptive achievement enabled and disabled, for all planning problems, taken as the median over all bound value combinations $(2, 2)$, $(2, 4)$, $(4, 2)$, and $(4, 4)$. The plots indicate that final-goal pre-emptive achievement did greatly improve and increase consistency in grades and quality scores for all problems. Higher median and lower spread is distinct in all cases. Further, time scores do not appear to be adversely affected when pre-emptive achievement is enabled. Paired tests for the same planning problem but comparing pre-emptive achievement enabled against disabled, indicate statistically significant differences between the median grades and quality scores, but insignificant differences between time scores, with and without pre-emptive achievement enabled. This indicates that final-goal pre-emptive achievement does greatly improve plan quality without significant downside.

Figure 5.8b is a relational line plot of total search time per search length with final-goal pre-emptive achievement enabled and disabled for all planning problems, taken as the median over all bound value combinations $(2, 2)$, $(2, 4)$, $(4, 2)$, and $(4, 4)$. The plots indicate that search times both with pre-emptive achievement enabled and disabled are similar at all search lengths until step: ≈ 47 for P1, ≈ 80 for P2, and ≈ 112 for P3. At these points, the search time with pre-emptive achievement disabled diverges and increases in an exponential trend relative to with pre-emptive achievement enabled. Interestingly, despite that time scores do not differ significantly, the plot shows clearly that the majority of the search time with pre-emptive achievement disabled is spent searching beyond the longest plan found with pre-emptive achievement enabled for all problems. Therefore, although even with pre-emptive achievement disabled h-online planning is fast enough to achieve optimal time scores, it can be substantially slower than with pre-emptive achievement enabled, as a direct result of the lower quality plans generated. This benefit of pre-emptive achievement in reducing search times is likely to be more pronounced for problems with longer plan lengths.

Concurrent Action Planning

The section provides more depth to the evaluation of the performance of concurrent action planning in online planning for the Basic division strategy and ground-first online method.

Figure 5.9a contains box plots of the grades, time scores, and quality scores, for each action planning type; sequential and concurrent, and planning problem, taken as the median over all bound value combinations $(2, 2)$, $(2, 4)$, $(4, 2)$, and $(4, 4)$. The plots indicate worse median quality scores for concurrent action planning over sequential by up to ≈ 0.1 for all problems. Time scores are always optimal for both action planning types on problem P1, but median time scores are worse for concurrent action planning for P2 and P3. However, this difference in time scores is small at less than ≈ 0.05. The grades for concurrent action planning resultantly clearly perform worse than sequential by at least ≈ 0.1 for all problems. The quartiles and range for both scores and the grades show a similar spread about the median value for both action planning types. This implies little difference in performance consistency between action planning types for the Basic strategy and ground-first method.

Figure 5.9b contains box plots of the grades, time scores, and quality scores, for each action planning type and bound value combination, taken as the median over all problems. The plots again indicate worse median quality scores for concurrent action planning over sequential in all cases, with the largest bound value combination notably performing the poorest, but again little difference in time scores. Despite this, the grades for concurrent action planning continue to perform worse overall for all bound values. However, the spread in grades and quality scores for concurrent action planning is clearly lower than sequential for all bound values. Unlike the previous plot, this indicates that concurrent action planning does produce more consistent performance than sequential action planning.

Figure 5.9c is a relational line plot of total search time per search length for each action planning type and planning problem, taken as the median over all bound value com-

binations. The plots indicate clearly higher total search time for concurrent action planning over sequential action planning for all problems, with more than double the time for concurrent at many steps. This is likely caused by the increased grounding time of concurrent action planning identified previously. The maximum search length for concurrent action planning is however slightly lower for all problems. This due to the compression of plan lengths that occurs due to the concurrent action sets. This saves some time by avoiding search at the highest search lengths. However, this does not appear to be sufficient to counteract the increased search times of concurrent action planning. The reasons why concurrent action planning achieves worse quality scores despite generating shorter plans is discussed in Section 5.2.7. Interestingly, since time scores do not appear to be greatly adversely affected, the increased search time does not heavily negatively impact performance. This increased cost may become more pronounced for larger problems with longer minimal plan lengths.

These results reinforce the conclusion that concurrent action planning relative to sequential for h-online planning with the Bas division strategy and ground-first online method, reduces median performance grades and scores but does increase consistency. Particularly, quality scores are significantly adversely affected for concurrent action planning.

Saved Groundings

Figure 5.10a contains box plots of the grades, time scores, and quality scores, for each of discarded and saved program groundings and for each planning problem, taken as the median over bound values $(2, 2)$, $(2, 4)$, $(4, 2)$, and $(4, 4)$. The plots indicate that quality scores are identical for both discarded and saved groundings in all cases. This is expected, as the grounding type should not affect plan quality. In contrast, the time scores for saved groundings are identical for P1, but worse for both P2 and P3 by ≈ 0.08. Paired tests for the same bound values and problem but different grounding type indicate statistically significant differences between median times scores of different grounding types for P2 and P3. The

indicates that saved grounds were not successful in reducing planning times.

Figure 5.10b is a bar plot of the sum over all abstraction levels of the grounding and solving times as a percentage of the total planning time for each program grounding type, taken as the median over all planning problems and bound values. The plot indicates a small reduction in the percentage contribution of grounding time and increase in solving time for saved groundings. Paired tests for the same bound values and problem but different grounding type indicate statistically significant differences between median percent grounding times of different grounding types for all planning problems. This confirms our expectations that saved groundings help to reduce the need to re-ground the base part of the ASP program.

Figure 5.10c is a bar plot of the aggregate time scores types used to calculate h-online's overall time score, taken as the median over all bound values and planning problems. Whilst saved groundings perform relatively well for minimum execution time per action score, they perform poorer for execution latency time score average wait time scores relative to discarded groundings. This explains the lower overall time scores for saved groundings.

Figure 5.10d is a relational line plot of grounding times (GT) and solving times (ST) against search length for both grounding types and both refinement levels on planning problem P3 only. This sheds some light on why these differences of scores occur. Note that in saved groundings, there is an overhead on inserting the existing plan into the program and "fixing" it to the grounding to cause a problem division, which is not accounted for in step-wise times. Further, the peaks and troughs caused by the minimum search length bound are not easy to see in the line for discarded groundings, as they are cancelled out by the substantial overlap of the different decision bound values. The plot indicates that the saved grounding provides generally lower and more consistent grounding and solving times that discard groundings at level 1, but in contrast higher times at level 2. For saved groundings, the increased solving time at level 1 is clearly great enough to determinant the average wait time for non-initial partial-problems to reduce the overall time score over the

discarded grounding, despite the lower search times at level 1. This increased solving time at level 1 appears to be partially caused by an unforeseen problem with saved groundings. Whereby, they are no longer able to benefit from the minimum search length bound after the first partial-problem at any given level. This is because, once enabled, the solution checking constraint in the minimal planning module cannot be disabled. Therefore, the incremental search algorithm is must solve the program even below the minimum search length bound.

These results are therefore considered inconclusive as to the effectiveness of saved program groundings. Further research and development is necessary to overcome the limitations of the current implementation. More elaborate evaluation must be done to determine if saved groundings can be useful in general for reducing planning times in HCR planning.

Partial Problem Blending

Figures 5.11a and 5.11b contains box plots of the grades, time scores, and quality scores, for each bound value combination $(2, 2)$, $(2, 4)$, $(4, 2)$, and $(4, 4)$, and blend quantity in the range $[0, 3]$, taken as the median over all planning problems, using sequential and concurrent action planning respectively. The plots indicate that in all cases, blending increases median plan quality scores over not blending, for a minor reduction in time scores. Further, for the higher bound values, that cause a larger number of partial-problems, the higher blend quantities produced better median and reduced spread of plan quality scores.

Figure 5.11c contains box plots of the grades, time scores, and quality scores, for each blend quantity and planning problem taken as the median over both action planning types and bound values. This plot indicates that the above benefits of blending on plan quality holds for all planning problems tested. In particular, it is notable that the worst case minimum quality score is improved when blending in enabled (with any quantity) over not blending for all problems. Further, for problems P2 and P3 a blend quantity of 3 always produced the best upper quartile and maximum performance grades.

Global tests between different blend quantities taken as the average over all decision bound values and problems indicate statistically significant differences in the median quality score in all cases. Paired tests for the same problem and decision bound values but different blend quantities indicate statistically significant differences in median quality scores for P2 and P3 when blending is enabled (with any quantity) as compared to not blending at all for all bound values. However, the differences in median grades and scores between different blend quantities were rarely statistically significant. In contrast for P1, the difference in quality scores was only statistically significant when the difference in the number of partial-problems involved in the two compared data sets was at their extremes.

These results therefore support the conclusion that blending is an effective strategy for improving the quality and consistency of plans generated by h-online planning. This benefit is particularly pronounced for problems with longer minimal plan lengths and where the number of partial-problems involved was large. However, the performance benefit between different blend quantities was not always clear or statistically significant. More evaluation is necessary done to discover the link between the performance of different blend quantities, planning problems, and problem division strategy decision bound values.

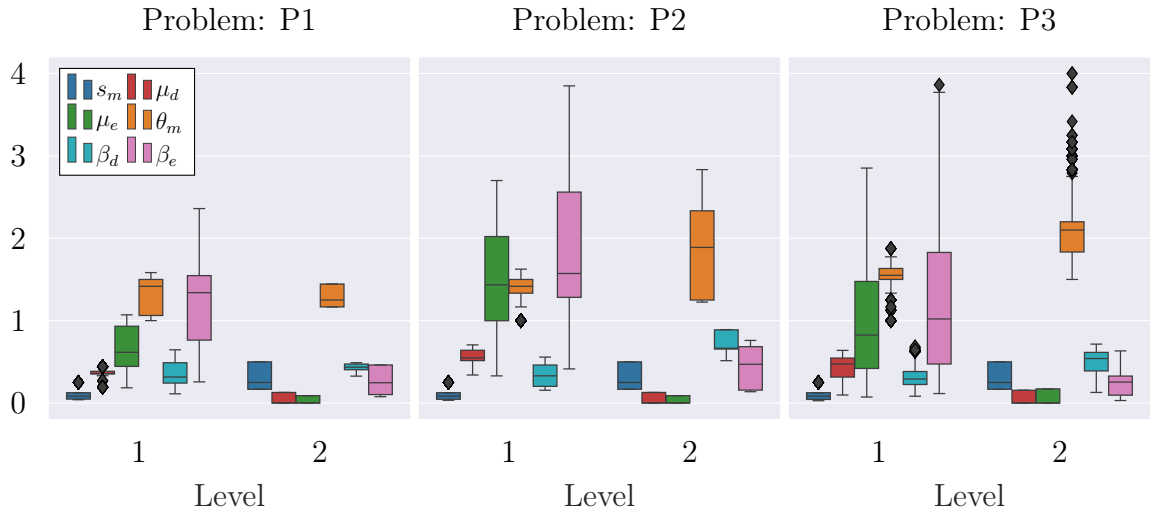
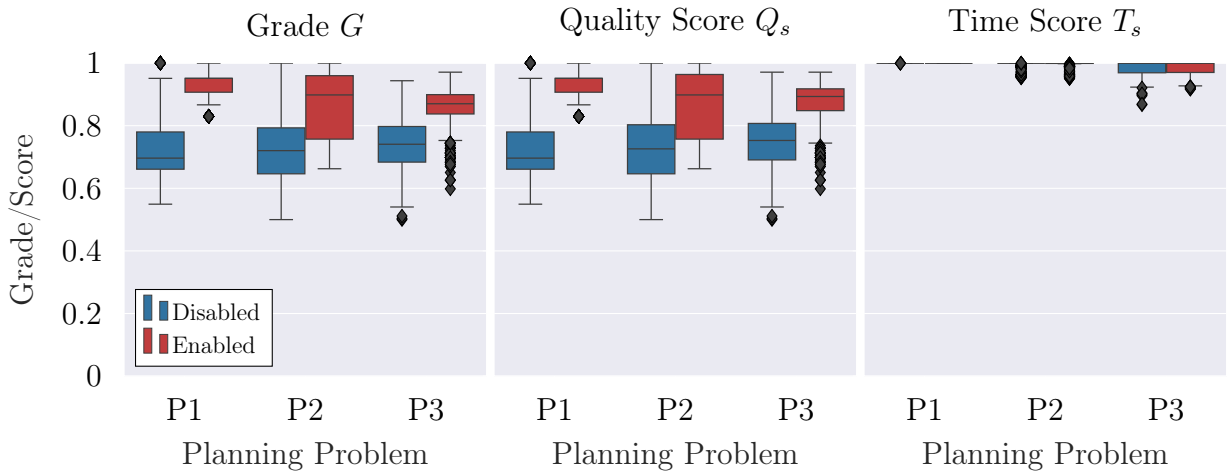
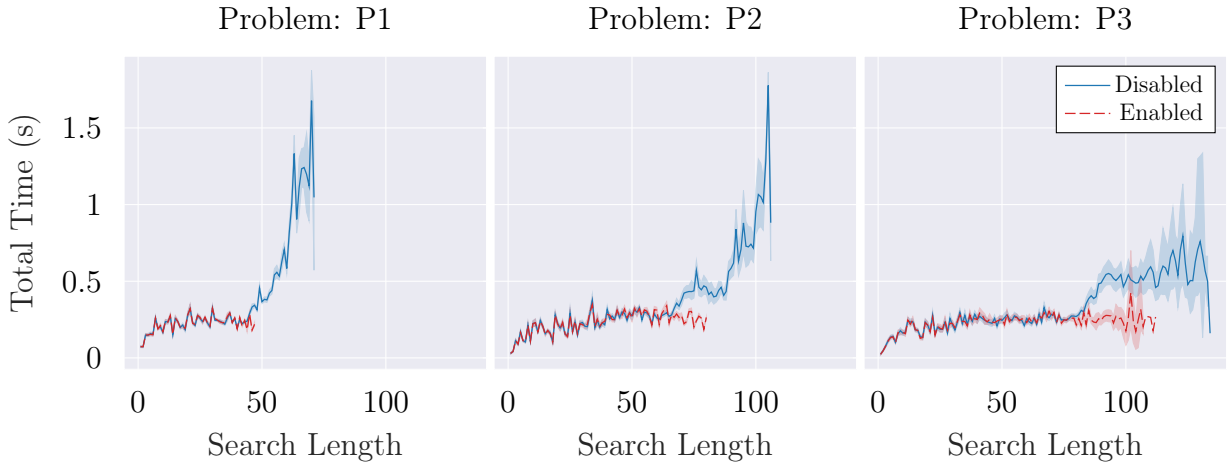


Figure 5.7: Problem and Plan Expansion and Balancing for each Planning Problem taken as median over all Decision Bound Value Combinations.

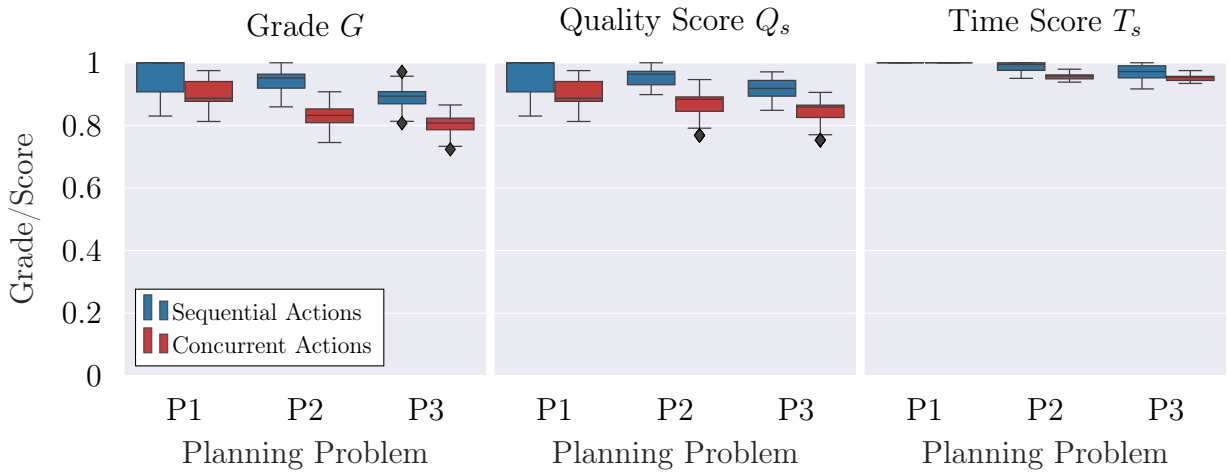


(a) Planner Performance Grades and Scores over Problems.



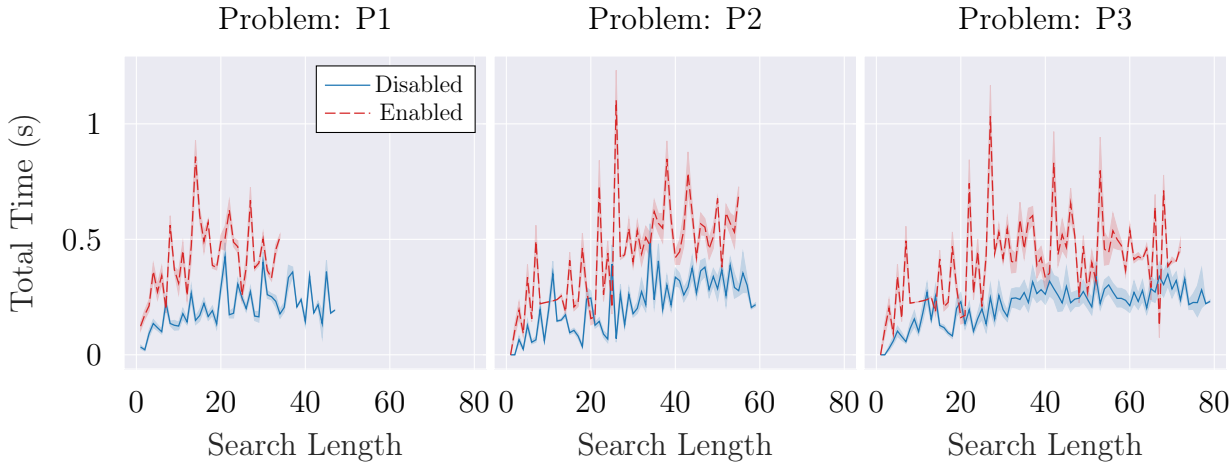
(b) Total Search Time per Search Length over Problems.

Figure 5.8: Planner Performance with Final-goal Pre-emptive Achievement Enabled and Disabled.



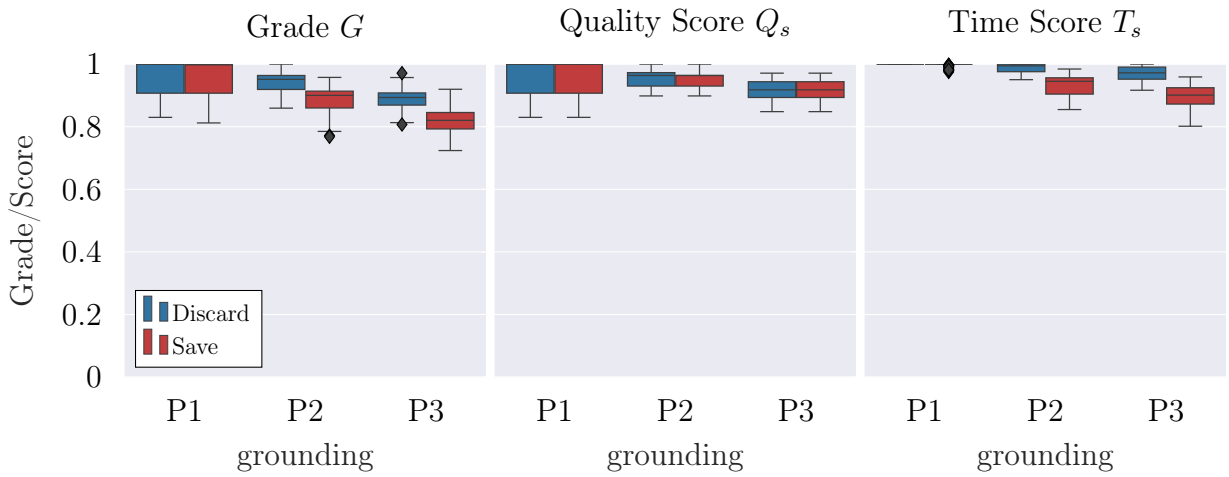
(a) Planner Performance Grades and Scores over Problems.

(b) Quality Scores over Bound Value Combinations.

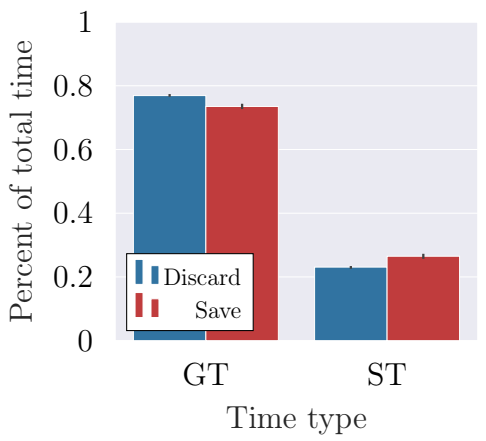


(c) Total Search Time per Search Length over Problems.

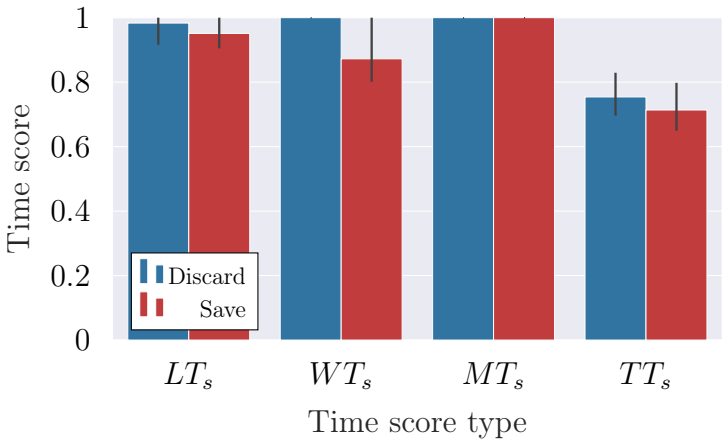
Figure 5.9: Planner Performance for different Action Planning Types.



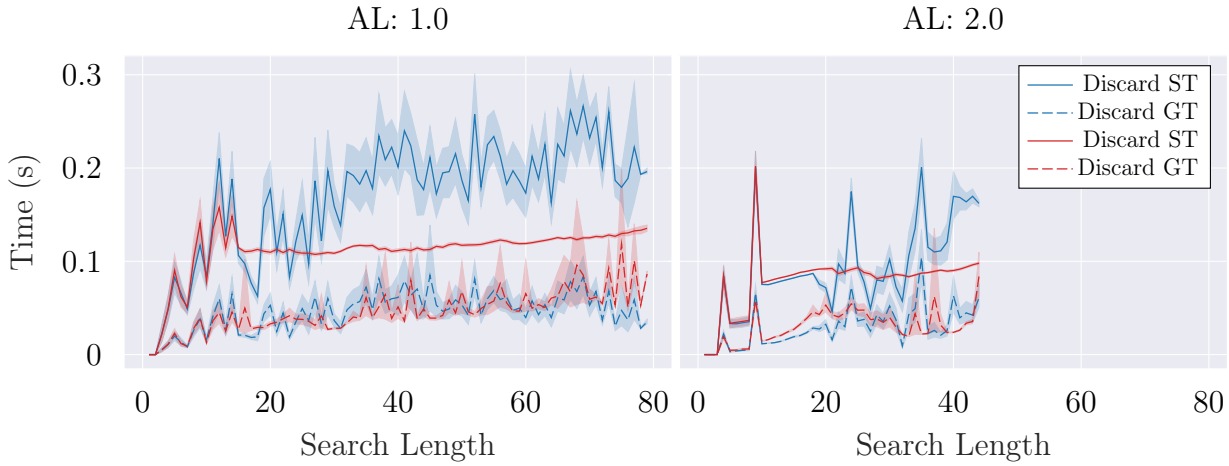
(a) Planner Performance Grades and Scores.



(b) Percent Times.



(c) Planning Time Aggregate Scores.

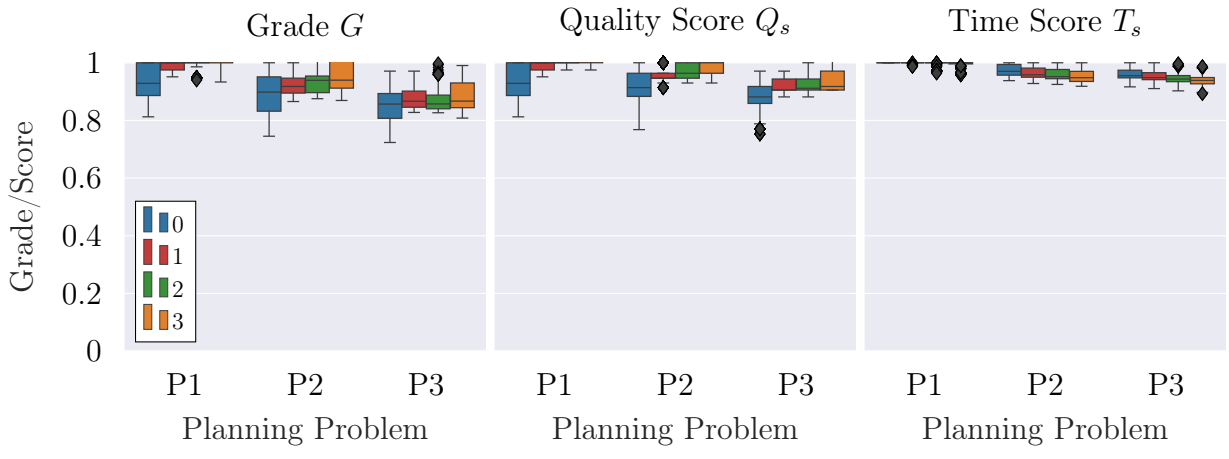


(d) Grounding and Solving Time per Search Length.

Figure 5.10: Planner Performance for different Program Grounding Types.

(a) Planner Performance Grades and Scores over Blend Quantities and Decision Bound Values with Sequential Action Planning.

(b) Planner Performance Grades and Scores over Blend Quantities and Decision Bound Values with Concurrent Action Planning.



(c) Planner Performance Grades and Scores over Blend Quantities and Problems.

Figure 5.11: Planner Performance for different Partial-Problem Blending Quantities.

5.2.5 Online Planning with Size-Bound Division Strategies

This section evaluates the performance of different decision bound values for the Hasty and Steady size-bound based problem division strategies and the three online planning methods; ground-first (gf), complete-first (cf), and hybrid (hy). With sequential action planning, sequential sub-goal stage achievement, and the min-bound search mode.

Figure 5.12 contains box plots of the grades, time scores, and quality scores, for all online planning methods and proactive size-bound strategies, taken as the median over all decision bound value combinations for bound values 2, 4, and 6, on problem P3 only. The plots indicate very marginal differences between configurations. In all cases, both Hasty and Steady have almost identical medians and spread over grades and both scores. Time scores are optimal with no spread and few outliers for both ground-first and hybrid, whereas for complete-first times scores are slightly below optimal at ≈ 0.96. Quality scores are sub-optimal in all cases, but are slightly worse for complete-first at ≈ 0.88 as compared to 0.90 for ground-first and hybrid. This is then reflected similarly in the grades. In comparison to the best bound values for the Basic strategy shown in Figure 5.9a no obvious benefit for the size bound based strategies exists. Paired tests for the same problem but different types of strategy (size-bound against problem-number bound) only indicates a significant difference in quality scores when comparing against the poorest performing bound values of Basic.

Figure 5.13 contains box plots of the quality scores, for all online planning methods, size-bound strategies, and decision bound value combinations $(2, 2)$, $(4, 2)$, $(4, 4)$, $(6, 2)$, $(6, 4)$, $(6, 6)$, on problem P3 only. The plots show that the median plan quality score always exceeds 0.75. In all cases, the highest median score is achieved by bound values $(6, 6)$, and the lowest score by bound values $(2, 2)$. Bound values $(4, 2)$ and $(6, 2)$ (both containing a bound value of 2 within them) are then consistently worse than $(4, 4)$ and $(6, 4)$. This indicates the problem size bound values do have a consistent impact on plan quality. A bound value of 2 is appears too small and leads to worse plan quality, whereas the larger

bound values produce better plan quality on average. Those with bound values of $(2, 2)$ also produce the highest spread and the least consistency in plan quality between runs. Between strategies and online methods, very similar results can be seen in most cases, indicating that the difference between the strategies and methods are marginal.

Global tests between all three different division strategies taken as the average over all decision bound values on problem P3 do not indicate statistically significant differences in the median grades and scores in any case. Paired tests for the same online planning method and decision bound values between Hasty and Steady again do not indicate statistically significant differences in median grades and quality scores in any case.

It was expected that Steady would achieve better plan quality and that Hasty would achieve better planning time score. However, these results indicate that the difference between the strategies is perhaps too small to make a significant difference for planning problems of this size. Similarly, it was expected that the complete-first and hybrid methods would achieve better plan quality and that ground-first would achieve better planning time score. However, whilst complete-first does produce marginally worse grades than hybrid and ground-first, the difference is too small to make a significant difference. The open question is whether the choice of division strategy and online method will become more important when scaling to larger problems with longer minimum plan lengths.

Figure 5.14 contains box plots of partial-plan expansion factor and balancing statistics for both size-bound strategies, taken as the median over all online planning methods and all bound values, on problem P3 only. The plot indicates that at abstraction level 2 the size-bound strategies achieve good partial-problem and plan balancing, with less than ≈ 0.10 problem balance deviation and error, and less than ≈ 1.00 plan balance deviation and error, and with low spread around the median values, for both strategies. At level 1, the size-bound strategies continue to produce good balancing, with less than ≈ 0.60 problem balance deviation and error, and less than ≈ 1.40 plan balance deviation and error, but with higher

spread about the median values, for both strategies. In contrast to the Basic strategy, which exhibited very poor balance error at level 1, the lower balance error of the size-bound based strategies confirms the expectation that they help to keep the size of partial-problems is well balanced over the hierarchy. The accumulation in balance error caused by the Basic strategy does not occur with the size-bound based strategies.

Figure 5.12: Planner Performance Grades and Scores for each Online Planning Method and Size-Bound Problem Division Strategy on Problem P3.

Figure 5.13: Planner Performance Grades and Scores for each Online Planning Method, Size-Bound Problem Division Strategy, and Size-Bound Decision Bound Value Combination, on Problem P3.

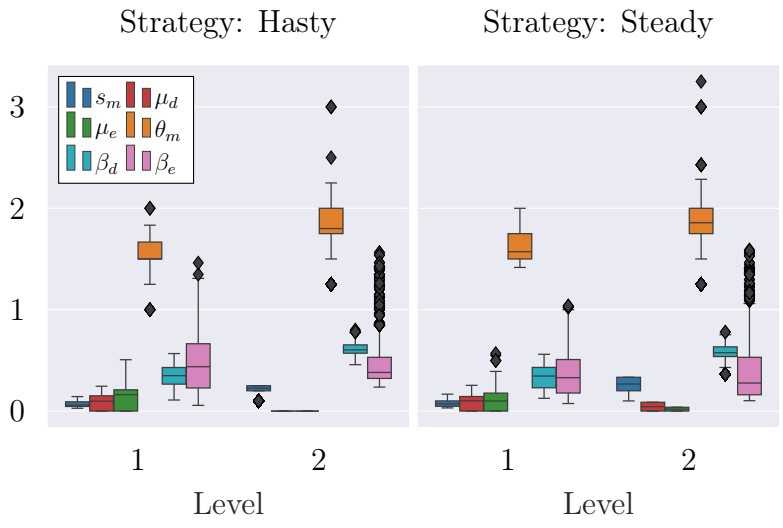


Figure 5.14: Problem and Plan Expansion and Balancing for each Size-Bound Problem Division Strategy on Problem P3 taken as median over all Decision Bound Value Combinations.

5.2.6 The Fundamental Reasoning Problems of HCR Planning

This section first describes the specific issues that lead to reductions in plan quality for HCR planning over classical in the experiments. The generalised fundamental reasoning problems of HCR planning that cause these specific issues are then stated in the following sections.

H-offline for problem P2 could find a sub-optimal plan. For this problem, only a single issue occured. This issues was due to one decision made in the condensed model at level 2. In the condensed and ground models, the existence of closed doors is considered, which Talos needs a free hand to open. It is optimal to open both doors before picking up both blocks from the store room, otherwise he would need to drop a block to open a door. In the ground model, Talos must move to different cells of the hallway to open these doors. It is then optimal to open the puzzle room door first, and then the store room door, before proceeding to collect the blocks. However, in the condensed model, where locations are only considered with respect to rooms rather than cells, Talos can open both doors without having to move. The order of opening the doors can therefore be chosen arbitrarily in the condensed model, because the resulting plan length is identical in both cases. In $\approx 50\%$ of runs, Talos chooses to open the store room door first, requiring two extra actions to refine to the ground-level.

Concurrent action planning can alleviate this issue. It allows the planner to take the path of least commitment to avoid non-deterministically choosing some arbitrary order over opening the doors in the condensed model. This is enabled by planning both the arm extend actions concurrently on the same step, then both the grasp door actions, and then both the door actuate actions. This enables the refinement planning at the original level to choose either order in the refined plan, allowing the puzzle room door to always be opened first. Unfortunately, the solution is imperfect because the two extend, two grasp, and two actuate action sets, are each in their own separate sub-goal stages. Whilst sequential achievement allows each of these individual action effects in the same sub-goal stage to be achieved in any order, the sub-goal stages as wholes still need to be achieved in the given order. As

a result, when Talos opens the first door in the refinement, he has to extend both of his arms in order to achieve the first sub-goal stage before grasping the first door. This means that an additional extend action always has to be included in the refined plan, making it impossible to get the optimal plan with concurrent action planning. Then, because he cannot simultaneously grasp both door handles in the original model, he cannot achieve the second sub-goal stage before opening the first door. He therefore opens the first door, before the third sub-goal stage relating to opening the doors even becomes current. Whilst the plan is refinable, the conformance constraint is clearly not guiding the planner to make monotonic progress towards the final-goal. This leads to a reduction in plan quality.

H-offline P3 always found sub-optimal plans. This occured because Talos has to make two trips to the store room to collect the three blocks that now start there. In the ground model, it is optimal to first open only the store room door, collect one block on the first trip, then open the puzzle room door with his spare hand, and finally place the block on the table. However, in the condensed model, it is always optimal to open both doors immediately upon entering the hallway, as they can be opened from the same location. Since this decision must be maintained in the refinement, the optimal ground plan can never be found. Resultantly, the choice of whether to take one or two blocks on the first trip to the store room becomes arbitrary in the condensed model. A further issue occurs if Talos takes only one block in the first trip in the relaxed model. In this case, when he reaches the puzzle table, it is possible for him to then arbitrarily use his spare arm to move blocks on the table. However, in the ground model, Talos must extend his arm before being able to move blocks. This demands extra redundant actions to be inserted during the refinement to use this spare arm. Those extra actions would be unnecessary if those blocks were moved later, when the remaining two blocks are carried from the store room. This is because his second arm has to be extended anyway to place the second block being carried from the store room onto the table.

H-online for all planning problems also experienced dependencies between the partial-problems involved. The most prolific example was when a problem division fell between Talos

extending his arm and grasping a block. The plan Talos generates in the condensed model involves entering the store room, immediately extending his arms and then grasping the blocks. This is because Talos can grasp the blocks from anywhere in the store room in this model. In the ground model however, before he can grasp the blocks, Talos needs one additional action to move from cell 1 to cell 0 of the store room. If a division is made between the sub-goal stages generated from the extend and grasp actions from the condensed model, Talos minimally achieves extension of his arms whilst still in cell 0 and unable to grasp the blocks. The planner cannot insert the additional move action in the first partial-problem. This is because it has no knowledge that the precondition of the grasp action necessary to achieve the next sub-goal stage that requires Talos to be in cell 1 and that this movement is disabled by extending his arms. As a result, Talos must retract, move, and then re-extend his arms in the next partial problem, as he is unable to move with his arms extended.

A less prolific but more extreme issue occured if Talos never has to enter the puzzle room in the first partial-problem in the condensed model of planning problem P3. This means that the puzzle room door is left closed at the end of the first partial-plan. If Talos chooses to take two blocks in the first trip to the store room in the relaxed model, then he will then end the first partial-plan with both hands filled with blocks when he enters the hallway. Therefore, he must plan additional actions to drop one of the blocks to be able to open the door, and then pick the block back up again. This dependency is always propagated to level 1. However, this happens rarely, because the sub-goal stage that requires entering the puzzle room is always either the 3rd or 4th in sequence out of 20, depending on whether the planner chooses to move only one or both blocks from the store room on the first trip in relaxed model. Therefore, this issue only happens if the first partial-problem is only of size 2 or 3, and both blocks are moved in the first trip. In all other cases, it does not happen, because of the ignorance problem in the condensed model at level 2. Whereby, it is always optimal to open both doors at the same time when first entering the hallway, and therefore the planner always opens both the doors in the first partial-problem.

The Ignorance Problem

The ignorance problem is the most intrinsic reasoning problem in HCR planning. It occurs from the lost knowledge about the constraints of the original problem in the abstract. This lost knowledge can cause the planner to make decisions in such ignorance, that are optimal in the context of the abstract problem, but which when maintained in the refinements (to achieve conformance) may be poor with respect the constraints of the original. This can occur repeatedly at all abstract levels, leading to an accumulation of errors down the hierarchy.

There exist two general causes of the ignorance problem identified from the experiments presented in this thesis. These are derived from; the loss and generalisation of action enabling constraints, within condensed and relaxed abstract models, respectively:

1. *Loss of enabling constraints*, can allow an abstract action to be planned, that is unconstrained in an abstract state, but the matching original actions that achieve the same effects, are constrained in many of the original states that map to the abstract. When such an abstract action is refined, additional original actions may be required to enable an action that achieves the same effects. These additional original actions may be unnecessary had a different (but still valid) abstract action been selected.

2. *Generalisation of enabling constraints*, can allow a contiguous sub-sequence of abstract actions, that are unconstrained in an abstract model, to be planned in an arbitrary order. If these actions become constrained in the original model, additional original actions may be required to achieve the effects of the abstract actions in the same order during refinement. These may be unnecessary were the ordering changed.

The ignorance problem cannot currently be avoided automatically. This is because HCR planning does not currently support backtracking, and the planner cannot measure relative plan quality[11]. Therefore, the planner cannot know what abstract plan is the best

[11]This would require generating multiple different abstract plans and comparing all possible refinements.

to select for refinement and cannot return to higher levels to revise abstract decisions that appear poor during refinement. The open question is whether the properties of abstract models that cause these problems can be minimised whilst still getting the speed benefits.

The Dependency Problem

The dependency problem occurs in online planning due to the division of complete-problems into partial-problems. If a division falls between dependent partial-problems, decisions made in the early partial-problems which are locally optimal (in the short term), might not be globally optimal (in the long term) respective of the complete problem. More specifically, after solving the earlier problem, and accepting the partial-plan as the partial solution up to the minimal achievement of its last in sequence sub-goal stage, the planner might be left in a state that makes it harder to solve the next partial-problem, because the planner is unable to consider and prepare for, what it will need to more easily solve the next problem. The dependency problem can lead to an accumulation of errors across the problem sequence, which can then cascade to and further accumulate at the lower-levels of the hierarchy.

In the BWP, the dependency problem was caused by interacting enabling conditions between abstract actions. It occured when placing a division before a sub-goal stage produced from an abstract action that has multiple enabling conditions, achieved by multiple earlier abstract actions, but where one enabling action disables the other. If the enabling conditions are generalised in the condensed model, a more specialised enabling condition may exist for actions at the original level that achieve the same effects as the abstract action. The more specialised preconditions may however not be satisfied by achieving the same effects as the abstract actions. Therefore, multiple abstract actions that achieve the enabling conditions of a later abstract action, must all be refined together to avoid the dependency problem.

The open question is whether the dependency between partial-problems can be predicted and then minimised with more intelligent and dynamic problem division strategies.

5.2.7 Limitations of Concurrent Action Planning

Concurrent action planning was effective for reducing time scores of classical and h-offline planning, but conversely reduced quality scores in both cases. Further, concurrent action planning with h-online increased planning times and reduced plan quality, and did not achieve the same scaling to large problems as h-online with sequential action planning did.

An unexpected observation was that compressing plans into fewer steps could make the quality of plans worse, by requiring more actions to reach the goal. This can occur because abstract plans may not just contain arbitrarily ordered contiguous sub-sequences of actions. It is possible for an abstract plan to contain multiple different sub-sequences of actions, where the sub-sequences are arbitrarily ordered, but the actions in each sub-sequence are strictly ordered. In the experiments, this occurred in the condensed model when opening the doors to the store room and puzzle room from inside the hallway. Recall that the ordering of opening the doors is arbitrary, because the constraints of the problem allow from both orders to achieve a valid minimal length plan because the robot does not have to move between opening them. However, the order does affect plan quality when refining at the next level, because one of the orderings is better quality than the other. The planner must however choose this order in ignorance when planning at the abstract level and the choice enforces that order in the refinement. This cannot be solved by concurrent action planning, because the arbitrary ordering exists between two distinct contiguous sub-sequences of actions, each of which has a strict ordering over their actions in that sub-sequence. Therefore, these action sub-sequences cannot be combined into a single unordered set on one step.

Since the planner currently always returns when it finds a minimal length plan and then sub-sequentially minimises the quantity of actions within the minimal length plan, there is no current methodology that allows it to trade longer plan lengths for fewer actions. To resolve this, developing a method to search beyond the minimum plan length, only whilst the quantity of actions in a plan monotonically decreases, should overcome this.

5.3 Summary of Findings

The following summarises the primary experimental findings of this chapter.

- Comparison of planning modes:
  - Classical ASP planning experienced an exponential increase in search time with search length. Therefore, the total planning time of classical planning increased exponentially with the minimum plan length of a planning problem.
  - Offline HCR planning reduced total planning time over classical. However, it still produced an exponential increase in search time with search length.
  - Online HCR planning reduced search times from increasing exponentially with search length to increasing linearly. This coupled with the capacity to incrementally generate and yield partial-plans reduced: total planning times, execution latency, average wait time per partial-plan, and average minimum execution time per action; exponentially over complete classical or offline HCR planning.
  - The scalability of the online HCR planning approach to problems with long plan lengths resultantly greatly exceeds that of classical ASP planning.
  - Concurrent action planning improved the performance of classical and offline HCR planning, but did not consistently improve the performance of online HCR planning. Classical and offline HCR planning with concurrent actions was not sufficient to outperform online HCR planning with sequential actions.
- Sub-goal stage achievement type and search mode:
  - The minimum search length bound enabled search mode marginally but consistently produced the best time score on all planning problems.
  - The time score benefit of using the minimum search length bound comes at the cost of preventing the ability to observe the progression of planning along the sub-goal stage sequence that is enabled by sequential yield planning.

  - No significant difference between the performance of sub-goal achievement types was found. This indicates that there was no overhead cost for relaxing the conformance constraint to sequential as is needed for concurrent action planning.

- Online planning with the Basic division strategy and the ground-first online method:
  - The choice of bound value (number of divisions) for the Basic strategy did clearly affect the planner's performance. A small number of divisions is sufficient for the problems tested. A large number of divisions gave diminishing returns in terms of time score, but gave increasingly worse plan quality scores.
  - The Basic strategy achieved poor balancing of partial-problems and plans, which was believed to be the primary cause of reduced plan quality.
  - Final-goal preemptive achievement and partial-problem blending proved consistently effective for all planning problems, in increasing plan quality for minor losses in planning speed, providing an acceptable trade-off.
  - However, concurrent action planning and saved groundings were not effective.

- Online planning with the size-bound based division strategies Hasty and Steady, and the ground-first, complete-first, and hybrid online methods:
  - The choice of bound value (problem sizes) for both Hasty and Steady did again clearly affect the planner's performance. The large bound values produced the best quality scores and always performed optimally for time scores.
  - The choice between the different size-bound based division strategies and online methods exhibited very little significant difference in performance.
  - The size-bound based strategies produced better balancing of partial-problems and plans over the Basic strategy. However, this unfortunately does not appear to have consistently increased plan quality scores by any significant amount.
  - These strategies and methods must be evaluated on a greater variety of larger problems to better understand their properties and behaviour.

Chapter Six

Conclusions

This thesis has proposed, implemented, and experimentally validated, Hierarchical Conformance Refinement (HCR) planning, a novel approach to domain-independent online task and high-level planning for robots using Answer Set Programming (ASP). HCR planning has successfully combined ASP planning with Hierarchical Refinement (HR) planning. In the union of their complementary capabilities, HCR planning finds the reinforcement of their strengths and overcoming over their weaknesses. In particular, HCR planning now supports rapid online planning, in which plans are generated partly overlapped with execution, to minimise the downtime and maximise the productivity of robots. The following summarises the novel contributions and the aims and objectives successfully achieved by this thesis.

6.1 Summary of Contributions and Achievements

The primary novel contribution of HCR planning is its new refinement method. HCR uses conformance refinement, where plans are required to achieve the same effects and remain structurally similar at all hierarchical levels. The conformance constraint enforces this requirement, whilst enabling a more flexible plan refinement and problem division mechanism, and supporting a more general hierarchy representation, than previous HR planners. The constraint simply encodes the effects of abstract actions generated in an abstract model of a domain and problem, as sequences of sub-goal stages, which must then be achieved in the same order when generating a refined plan in a more concrete model. This technique has successfully provided the following novel benefits to the field of HR and ASP planning.

- *High Quality Satisficing Planning*: By encoding the effects of abstract actions into sub-goal stages which must only be achieved in the same order during refinement planning enables the interleaving property. Whereby, stages included in the same refinement problem are pursued simultaneously to eliminate dependencies between them. This is possible without the plan length estimation function needed in past ASP HR planners.

- *Rapid Online Planning*: Complete refinement problems can be divided into partial-problems, each of which refine any contiguous sub-sequence of abstract actions simultaneously. This avoids the limitation of past HR planners that required all abstract actions to be refined independently, which exaggerated dependencies between them. Online partial-planning successfully reduced the time complexity of planning from increasing exponentially in plan length, to increasing only linearly. Further, because partial-plans are yielded to a robot independently, a robot can begin execution as soon as the first is generated, and exponentially faster than generating a complete plan. This greatly increases the scalability of ASP planners to problems with long plan lengths.

- *Flexible Problem Division*: Partial-problems enable the use of general and versatile problem division strategies and online planning methods. These are external decision making systems that dynamically decide how to divide problems and traverse over the hierarchy during online partial-planning. Three simple division strategies and three online planning methods were implemented to prove generality of these systems. To improve overall plan quality when online planning, two approaches; final-goal pre-emptive achievement and partial-problem blending, were implemented and proved effective in experimental studies, for only very minorly increased planning times.

- *Generalised Hierarchy Representation*: By refining plans based on achieving action effects, rather than action preconditions as done in previous HR planners, and by then adding a state abstraction mapping over state variables, HCR planning increases the generality of its hierarchy representation. This allows HCR planning to support any form of abstract model to which a deterministic and exhaustive state abstraction

mapping can be defined from the original model, including state abstractions that modify or replace actions and state variables, and not just those that remove them.

- *Intrinsic Support for Condensed Domain Models*: Condensed abstract domain models are state abstractions that generalise the state representation into a more granular form. They extend past work by handling this type of abstraction through a class-based relation, in which the designer specifies a logical coordination (a conceptional or semantic relationship) between two sub-classes that both subordinate from the same super-class, where one sub-class is called an ancestor and the other the descendent. In a condensed model, all state variables referring to the descendent type are then generalised to instead refer to the ancestor type. This combines a sub-set of original state literals into a single abstract state literal, therefore reducing the size of representing a single state and the number of possible states. This specifications also generates the abstract versions of the domain laws automatically (avoiding the need to manually rewrite them), and gives a general way to obtain the state abstraction mapping.

The experimental studies in this thesis show that the HCR approach has successfully increased the speed and scalability of ASP planning on a combined blocks world and navigation domain, and significantly outperforms the classical approach to ASP based planning. Specifically, HCR planning exponentially reduces total planning and execution latency times, over classical ASP planning, for only small losses in plan quality. Robots equipped with the HCR planning algorithm can therefore begin plan execution in a fraction of the time of classical ASP planners, increasing productivity and reducing down times. Using an optimal configuration for the most complex planning problem tested, this algorithm can reduce total planning time by 98% (from 607.0 to 14.1 seconds), and execution latency by 99% (to 7 seconds), for only a 11.0% increase in total plan length. This proves that online HCR planning is effective, and for the first time makes ASP based planning fast enough for practical general-purpose robotics applications. However, HCR planning still has limitations, these are summarised in the following, and possible paths for future work are proposed.

6.2 Summary of Limitations and Future Work

In this thesis, only one experimental and example domain was used and the test problems are all similar in nature and structure. Some additional example domains and problems are available on the online appendix at Kamperis, 2023. The BWP domain does represent general reasoning challenges and the types of complex constraints that occur commonly in interesting physical domains. However, to prove generality, further experiments are needed to determine if the performance of HCR holds for a wider range of domains and problems.

The main limitation on the scalability of HCR planning is how the ASP solver currently handles solving partial-problems. Partial-problems reduce plans lengths linearly and therefore search complexity exponentially. However, they do nothing to reduce the problem description size over the complete-problem, despite that many components of a planning domain will not be relevant to a single partial-problem. This means that when solving a partial-problem, the planner still generates a ASP ground program the represents the complete-domain. In order to continue scaling the HCR planning algorithms to arbitrarily large problems, future research must develop a general method for reasoning about relevancy (of entities and domain knowledge) to individual partial-problems, to obtain partial-domain descriptions for partial-problems. This may need to involve inductive learning techniques that can decide upon the relevancy of entities to partial-problems based on prior experience.

Several specific issues causing loss of plan quality were identified, from which the two fundamental reasoning problems of HCR planning were identified. The ignorance problem occurs from the lost knowledge in the abstractions which means that not all abstract plans are optimal. This combined with a lack of capacity to measure abstract plan quality directly (without comparing against known optimums) means that it is not possible to select the best quality abstract plan to refine. The dependency problem occurs from the division of complete refinement problems into sequences of partial refinement problems which means that dependencies can occur between sub-goal stages in different partial-problems. This

combined with a lack of capacity to predict dependencies, means that problem dependencies, although uncommon, are difficult to avoid entirely in HCR planning. The open question is whether the properties of abstract domain models and partial-problems that cause these reasoning problems can be minimised whilst still achieving the speed benefits.

The problem division strategies used in this thesis proved effective for the domain tested but are highly naive and only make decisions based on pre-defined decision making bounds. As problems become larger, the decisions of the division strategies are likely to become increasingly impactful on planning performance. To make division strategies more effective may require the capacity to learn and generalize how long it takes to refine abstract actions, how long it takes to execute ground actions, and where problem dependencies are likely to exist. This would allow the strategies to make informed decisions about when to divide problems to maximize productivity of the robot and always ensure that there are actions ready to execute, whilst minimising dependencies to maximise plan quality.

HCR planning currently assumes that the robot has complete knowledge of the problem's initial state and that its plans will never fail when executed. These assumptions rarely hold in real-world dynamic, partially-observable, and uncertain domains, and resultantly plans fail often. Part of the original concept of HCR planning, was that the tree-like structure of HCR plans could allow plan flaws occurring from unexpected observations obtained during plan execution, that would usually cause plans to fail, to be rapidly repaired online, by identifying and repairing only the flawed branches of a plan, rather than having to completely re-generate the entire plan as is necessary in classical planning. Due to time and space constraints, this was not developed, but remains a fruitful path for future work.

The ASH planner currently only optimises plan quality by minimising the quantity of actions in a plan. In real-world domains, other quality criteria, such a predicted execution time and resource usage, are very likely to be of importance. Introducing the capacity to optimise plans based on such a criteria is therefore a crucial path for future development.

Appendix One

Translation of HCR Domain Laws to Action Language BC

The following describes the translation from the domain laws used in this thesis into the form proposed by the action language BC in Lee, Lifschitz, and F. Yang, 2013, with modifications made to use the action and state representation predicates used by this thesis defined in Section 3.5.1. Note that encoding of state variables relations, state variable constraints, and state abstraction mappings used in this thesis is the same as the encoding that is proposed by action language BC, and as such translations are not necessary for them.

The action language BC would encode the domain laws for action effects and preconditions directly as follows:

- For each Action Effect;
  - Replace: $R\$\kappa_r \Rightarrow a_i^{pl}(\bar{T}\$\bar{\kappa}_n^a)$ **causes** $f_i^{pl}(\bar{T}\$\bar{\kappa}_m^f) = \Upsilon\$\kappa_v$ **if** $cons_{i-1}^{pl}$
  - With: $holds(pl, f(\bar{T}_m^f), \Upsilon, i) \leftarrow occurs(pl, R, a(\bar{T}_n^a), i), \epsilon(cons_{i-1}^{pl}), body^{pl}(a, f)$
- For each Positive Action Precondition;
  - Replace: $R\$\kappa_r \Rightarrow a_i^{pl}(\bar{T}\$\bar{\kappa}_n^a)$ **requires** $f_{i-1}^{pl}(\bar{T}\$\bar{\kappa}_m^f) = \Upsilon\$\kappa_v$ **if** $cons_{i-1}^{pl}$
  - With:

    $\leftarrow occurs(pl, R, a(\bar{T}_n^a), i), not\ holds(pl, f(\bar{T}_m^f), \Upsilon, i-1), \epsilon(cons_{i-1}^{pl}), body^{pl}(a, f)$
- For each Negative Action Precondition;
  - Replace: $R\$\kappa_r \Rightarrow a_i^{pl}(\bar{T}\$\bar{\kappa}_n^a)$ **requires** $f_{i-1}^{pl}(\bar{T}\$\bar{\kappa}_m^f) \neq \Upsilon\$\kappa_v$ **if** $cons_{i-1}^{pl}$

– With:$\leftarrow occurs(pl, R, a(\bar{T}_n^a), i), holds(pl, f(\bar{T}_m^f), \Upsilon, i-1), \epsilon(cons_{i-1}^{pl}), body^{pl}(a, f)$

Recall that the method of encoding the domain laws in HCR planning as proposed by this thesis is as follows:

- For each Action Effect;
  - Replace: $R\$\kappa_r \Rightarrow a_i^{pl}(\bar{T}\$\bar{\kappa}_n^a)$ **causes** $f_i^{pl}(\bar{T}\$\bar{\kappa}_m^f) = \Upsilon\$\kappa_\upsilon$ **if** $cons_{i-1}^{pl}$
  - With: $effect(pl, R, a(\bar{T}_n^a), i, f(\bar{T}_m^f), \Upsilon, i) \leftarrow \epsilon(cons_{i-1}^{pl}), body^{pl}(a, f)$
- For each Positive Action Precondition;
  - Replace: $R\$\kappa_r \Rightarrow a_i^{pl}(\bar{T}\$\bar{\kappa}_n^a)$ **requires** $f_{i-1}^{pl}(\bar{T}\$\bar{\kappa}_m^f) = \Upsilon\$\kappa_\upsilon$ **if** $cons_{i-1}^{pl}$
  - With: $precond(pl, R, a(\bar{T}_n^a), i, f(\bar{T}_m^f), \Upsilon, true, i) \leftarrow \epsilon(cons_{i-1}^{pl}), body^{pl}(a, f)$
- For each Negative Action Precondition;
  - Replace: $R\$\kappa_r \Rightarrow a_i^{pl}(\bar{T}\$\bar{\kappa}_n^a)$ **requires** $f_{i-1}^{pl}(\bar{T}\$\bar{\kappa}_m^f) \neq \Upsilon\$\kappa_\upsilon$ **if** $cons_{i-1}^{pl}$
  - With: $precond(pl, R, a(\bar{T}_n^a), i, f(\bar{T}_m^f), \Upsilon, false, i) \leftarrow \epsilon(cons_{i-1}^{pl}), body^{pl}(a, f)$

Further recall the domain independent logic rules on the minimal planning module of ASH that applies the domain laws for action effects and preconditions:

- Rule A.1: Apply the direct effects of an action on the time step it was planned at.
- Rules A.2-A.3: The action precondition integrity constraints, ensuring that an action cannot be planned in a state that does not satisfy (violates) its preconditions.

$$holds(l, f(\bar{t}), \upsilon, i) \leftarrow effect(l, r, a(\bar{t}), f(\bar{t}), \upsilon, i),\ occurs(l, r, a(\bar{t}), i) \tag{A.1}$$

$$\leftarrow precond(l, r, a(\bar{t}), f(\bar{t}), \upsilon, true, i), \tag{A.2}$$

$$not\ holds(l, f(\bar{t}), \upsilon, i-1),\ occurs(l, r, a(\bar{t}), i)$$

$$\leftarrow precond(l, r, a(\bar{t}), f(\bar{t}), \upsilon, false, i), \tag{A.3}$$

$$holds(l, f(\bar{t}), \upsilon, i-1),\ occurs(l, r, a(\bar{t}), i)$$

Note that the domain independent rules were written as ground rules for conciseness.

To translate from the encoding of domain laws used by HCR planning to the standard encoding of BC the following process is followed, using the effect rule as example:

1. The head of the encoding of the domain law for HCR is deleted and replaced by the head of the domain independent logic rule;
   - Replace: $effect(pl, R, a(\bar{T}_n^a), i, f(\bar{T}_m^f), \Upsilon, i) \leftarrow \epsilon(cons_{i-1}^{pl}), body^{pl}(a, f)$
   - With: $holds(pl, f(\bar{T}_m^f), \Upsilon, i) \leftarrow \epsilon(cons_{i-1}^{pl}), body^{pl}(a, f)$
2. Add the body of the domain independent logic rule to the body of the encoded domain law with the exception of the atom that was the old head of the law;
   - Replace: $holds(pl, f(\bar{T}_m^f), \Upsilon, i) \leftarrow \epsilon(cons_{i-1}^{pl}), body^{pl}(a, f)$
   - With: $holds(pl, f(\bar{T}_m^f), \Upsilon, i) \leftarrow occurs(pl, R, a(\bar{T}_n^a), i), \epsilon(cons_{i-1}^{pl}), body^{pl}(a, f)$

The result is the same rule as the standard encoding of action language BC.

Appendix Two

Complete BWP Domain Definition

The following Tables B.1, B.2, and B.3 define the class hierarchy of the BWP domain, and Tables B.4, B.5, B.6, and B.7 contain complete lists of all domain sorts in the BWP domain.

Table B.1: Class types present in each Domain Model of the BWP.

Class	**Description**	**1**	**2**	**3**
location	A location an object can be in.	x	x	x
room	A room that an object can be located in.	x	x	x
object	A physical object in the domain.	x	x	x
robot	A robot which can take actions.	x	x	x
grounded	An object that cannot change location.	x	x	x
graspable	An object that can be grasped.	x	x	x
placeable	An object that can be placed on a surface.	x	x	x
block	A cuboid block.	x	x	x
surface	A surface that can have objects placed upon it.	x	x	x
table	A table.	x	x	x
tower	A surface a tower can be built on.	x	x	x
colour	The colour of an object.	x	x	x
arm	A manipulator arm assembly.	x	x	x
grasper	An object that can grasp other objects.	x	x	x
door	A door between locations.	x	x	
handle	A handle of a door.	x	x	
component	An object that is a component part of another object and which has changeable configuration states.	x	x	
state	A possible configuration state of a component object.	x	x	
extensible	An object that can be extended or retracted.	x	x	
cell	A cell of a room that objects can be located in.	x		
limb	The upper limb of a manipulator arm.	x		
end	The end effector hand of a manipulator arm.	x		
side	The side of a table.	x		

Table B.2: Class Inheritance Relations present in the BWP.

Super-Class	**Sub-Classes**	**Description**
location	room, cell	Rooms and cells are locations.
object	robot, arm, limb, hand, table, side, block, grounded	A robot, its manipulator arms, and their components; limb and hand are objects. A table and its sides are objects. A block is an object. Grounded objects are objects.
grounded	handle	Door handles are grounded and cannot be moved.
configurable	extensible, grasper, door	Extensible components, components that can grasp objects, and doors all have configuration states.
extensible	arm, limb	Manipulator arms as a whole, and their descendent (upper) limbs are extensible.
grasper	arm, hand	Manipulator arms as a whole, and their descendent hands are graspers.
graspable	block, handle	Blocks and door handles are graspable.
placeable	block	Blocks can be placed on surfaces.
surface	table, side, block	Tables, their sides, and blocks are all surfaces that other blocks can be placed upon.
tower	table, side	Tables and their sides are surfaces that towers can be built on.

Table B.3: Class Override Relations present in the BWP.

Ancestor Class	**Descendent Class**	**Override Class**	**Description**
room	cell	location	Rooms are composed of a set of cells. Both can refer to locations that an object can be in.
table	side	surface	Tables have two sides. Both the table as a whole, and its sides can be considered target surfaces to place objects upon.
arm	limb	extensible	Manipulator arms have an upper limb which makes them extensible.
arm	hand	grasper	Manipulator arms have a hand which allows them to grasp objects.

Table B.4: Actions present in each Domain Model of the BWP.

Action	**Description**	**1**	**2**	**3**
$R\$robot \rightarrow move(L\$location)$	Move to a location L.	x	x	x
$R\$robot \rightarrow grasp(G\$grasper, O\$graspable)$	Grasp an object O with a grasper G.	x	x	x
$R\$robot \rightarrow release(G\$grasper, O\$graspable)$	Release an object O grasped with grasper G.	x	x	x
$R\$robot \rightarrow put(G\$grasper, O\$placeable, S\$surface)$	Put an object O grasped with grasper G on a surface S.	x	x	x
$R\$robot \rightarrow lift(G\$grasper, O\$placeable)$	Lift an object O grasped with grasper G.	x	x	x
$R\$robot \rightarrow actuate(G\$grasper, D\$door)$	Actuate door D with grasper G.	x	x	
$R\$robot \rightarrow configure(C\$configurable, S\$state)$	Configure a component C of the robot into state S.	x	x	

Table B.5: Inertial Fluents present in each Domain Model of the BWP.

Fluent	**Description**	**1**	**2**	**3**
$in(\$object) = \$location$	Locations of objects.	x	x	x
$on(\$object) = \$surface$	Placement of objects.	x	x	x
$grasping(\$robot, \$grasper) = \$object$	Grasped objects.	x	x	x
$config(\$configurable) = \$state$	Configuration of components.	x	x	
$config(\$door) = \$state$	Configuration of doors.	x	x	

Table B.6: Defined Fluents present in each Domain Model of the BWP.

Fluent	**Description**	**1**	**2**	**3**
$tower\_base(B\$block, T\$tower)$	Whether a block is at the base of a tower.	x	x	x
$in\_tower(B_1\$block_1, B_2\$block\$_2)$	Whether a block is part of the tower with the given base.	x	x	x
$unordered\_tower(B\$block, T\$tower)$	Whether the tower with the given base is not built in ascending order (from bottom to top) of block numbers.	x	x	x
$complete\_tower(C\$colour, B\$block)$	Whether the tower with the given base is built of blocks of all the same colour.	x	x	x

Table B.7: Static State Variables present in each Domain Model of the BWP.

Static	**Description**	**1**	**2**	**3**
in($grounded) = $location	Locations of grounded objects (which cannot be moved).	x	x	x
in($grounded) = $location	Locations of grounded objects (which cannot be moved).	x	x	x
colour_of(B$block, C$colour)	The colour of the blocks.	x	x	x
connected(L_1location, L_2location)	Connections between locations (defining a graph-like world map).	x	x	
connected_by_door(L_1location, L_2location, D$door)	Doorways on location connections.	x	x	

The following lists all laws and state abstraction mappings in the BWP domain.

- Action Effects:
  - When a robot moves its location changes accordingly;

$$R\$robot \Rightarrow \ move^{pl}(L\$location) \ \textbf{causes} \ in^{pl}(R\$object) = L\$location$$

  - When a robot grasps an object with a grasper the object becomes grasped;

$$R\$robot \Rightarrow grasp^{pl}(G\$grasper, O\$graspable)$$

$$\textbf{causes} \ grasping^{pl}(R\$robot, G\$grasper) = O\$object$$

  - When a robot releases an object the object is no longer grasped;

$$R\$robot \Rightarrow release^{pl}(G\$grasper, O\$graspable)$$

$$\textbf{causes} \ grasping^{pl}(R\$robot, G\$grasper) = nothing$$

  - When a robot releases an object from an end effector, the end effector is no longer aligned with that object;

$$R\$robot \Rightarrow release^{pl}(G\$grasper, O\$graspable)$$

$$\textbf{causes} \ configuration^{pl}(G\$grasper) = aligned\_with(nothing)$$

– A robot can change the configuration of its components;

$$R\$robot \Rightarrow configure^{pl}(C\$component, S\$state)$$

$$\textbf{causes } configuration^{pl}(C\$component) = S\$state$$

– When a robot retracts a manipulator limb, any of its sibling end effectors are no longer aligned with any objects unless it is grasping an object;

$$R\$robot \Rightarrow configure^{pl}(E\$extensible, retracted)$$

$$\textbf{causes } configuration^{pl}(E\$component) = aligned\_with(nothing)$$

$$\textbf{if } grasping^{pl}(E\$grasper) = nothing$$

– When a robot puts a placeable object on a surface the object is on that surface;

$$R\$robot \Rightarrow put^{pl}(G\$grasper, O\$placeable, S\$surface)$$

$$\textbf{causes } on^{pl}(O\$object) = S\$surface$$

– When a robot lifts an object the object is on nothing;

$$R\$robot \Rightarrow lift^{pl}(G\$gripper, O\$graspable) \textbf{ causes } on^{pl}(O\$placeable) = nothing$$

– When a robot actuates a door its configuration is flipped;

$$R\$robot \Rightarrow actuate^{pl}(G\$grasper, D\$door)$$

$$\textbf{causes } configuration^{pl}(D\$configurable) = S_1\$state$$

$$\textbf{if } configuration^{pl}(D\$configurable) = S_2\$state, S_1 \neq S_2, S \in \{open, closed\}$$

- Action Preconditions:

– A robot can only move between connected locations (ignored in relaxed model);

$$R\$robot \Rightarrow move^{pl}(L\$location)$$

$$\textbf{requires } connected^{pl}(L_1\$location, L_2\$location) = true$$

$$\textbf{if } in^{pl}(R\$robot) = L_2\$location), pl < 3$$

– A robot can only move between locations that are connected by a door if that

door is open (ignored in relaxed model);

$$R\$robot : move^{pl}(L\$location)$$

$$\textbf{requires } configuration(D\$configurable) = open$$

$$\textbf{if } connected\_by\_door^{pl}(L_1\$location, L_2\$location, D\$door),$$

$$in^{pl}(R\$robot) = L_2\$location), pl < 3$$

– A robot can only open a door if it is grasping one of its handles (ignored in relaxed model);

$$R\$robot \Rightarrow actuate^{pl}(D\$door)$$

$$\textbf{requires } 1 = \#count\{grasping^{pl}(R\$robot, G\$grasper) = H\$handle,$$

$$part^{pl}(D\$object, H\$component)\}$$

$$\textbf{if } pl < 3$$

– A robot can only move if all of its manipulator arms are retracted (ignored in relaxed model);

$$R\$robot \Rightarrow move^{pl}(L\$location)$$

$$\textbf{requires } configuration^{pl}(R\$robot, C\$component) = retracted$$

$$\textbf{if } part^{pl}(R\$robot, C\$component), isa^{pl}(C\$extensible)$$

– A robot can only grasp an object which shares its location;

$$R\$robot \Rightarrow grasp^{pl}(G\$grasper, O\$graspable)$$

$$\textbf{requires } in^{pl}(R\$object) = in^{pl}(O\$object)$$

– A robot can only grasp an object with a given grasper if it is not currently grasping another object with that grasper;

$$R\$robot \Rightarrow grasp^{pl}(G\$grasper, O\$graspable)$$

$$\textbf{requires } grasping^{pl}(R\$robot, G\$grasper) = nothing, O \neq nothing$$

– A robot cannot grasp an object that has another object on top of it;

$$R\$robot \Rightarrow grasp^{pl}(G\$grasper, O_1\$graspable)$$

$$\textbf{requires } on^{pl}(O_2\$placeable) \neq O_1\$surface, O_1 \neq O_2, O_1 \neq nothing$$

– A robot can only grasp an object with a grasper if the grasper is extensible and it is extended or if it is attached to another extensible component that is extended;

$$R\$robot \Rightarrow grasp^{pl}(G\$grasper, O\$graspable)$$

$$\textbf{requires } configuration^{pl}(R\$robot, G\$component) = extended$$

$$\textbf{if } part^{pl}(R\$robot, G\$component), isa^{pl}(G, extensible)$$

$$R\$robot \Rightarrow grasp^{pl}(G\$grasper, O\$graspable)$$

$$\textbf{requires } configuration^{pl}(R\$robot, E\$component) = extended$$

$$\textbf{if } part^{pl}(R\$robot, G\$component), part^{pl}(R\$robot, E\$component),$$

$$sib^{pl}(G, E), isa^{pl}(E, extensible)$$

– A robot can only grasp an object with an end effector if that end effector is aligned with that object;

$$R\$robot \Rightarrow grasp^{pl}(G\$grasper, O\$graspable)$$

$$\textbf{requires } configuration^{pl}(G\$component) = aligned\_with(O), O \neq nothing$$

– A robot can only release an object when it is grasping that object;

$$R\$robot \Rightarrow release^{pl}(G\$grasper, O\$graspable)$$

$$\textbf{requires } grasping^{pl}(R\$robot, G\$grasper) = O\$graspable, O \neq nothing$$

– A robot can only release an object from a grasper if the grasper is extensible and it is extended or if it is attached to another extensible component that is extended;

$$R\$robot \Rightarrow release^{pl}(G\$grasper, O\$graspable)$$

$$\textbf{requires } configuration^{pl}(R\$robot, G\$component) = extended$$

$$\textbf{if } part^{pl}(R\$robot, G\$component), isa^{pl}(G, extensible)$$

$$R\$robot \Rightarrow release^{pl}(G\$grasper, O\$graspable)$$

requires $configuration^{pl}(R\$robot, E\$component) = extended$

if $part^{pl}(R\$robot, G\$component), part^{pl}(R\$robot, E\$component),$

$$sib^{pl}(G, E), isa^{pl}(E, extensible)$$

– A robot can only align its hand with an object if it is grasping nothing with that hand;

$$R\$robot \Rightarrow configure^{pl}(G\$component, aligned\_with(O))$$

requires $grasping^{pl}(R\$robot, G\$grasper) = nothing, O \neq nothing$

– A robot can only put an object on a surface which shares its location;

$$R\$robot \Rightarrow put^{pl}(G\$grasper, O\$placeable, S\$surface)$$

requires $in^{pl}(R\$object) = in^{pl}(S\$object)$

– A robot cannot put an object on top of another object that already has some other object on top of it;

$$R\$robot \Rightarrow put^{pl}(G\$grasper, O_1\$placeable, S\$surface)$$

requires $on^{pl}(O_2\$object) \neq S\$surface, isa^{pl}(S, block), O_1 \neq O_2$

– A robot can only put an object on a surface when it is grasping that object;

$$R\$robot \Rightarrow put^{pl}(G\$grasper, O\$placeable, S\$surface)$$

requires $grasping^{pl}(R\$robot, G\$grasper) = O\$graspable, O \neq nothing$

– An armed robot cannot put an object on another object if any robot is grasping the other object with any of its graspers;

$$R\$robot \Rightarrow put^{pl}(G_1\$grasper, O\$placeable, S\$surface)$$

requires $grasping^{pl}(R\$robot, G_2\$grasper) = S\$graspable, O \neq nothing$

– A robot can only put an object on a surface with a grasper if the grasper is extensible and it is extended or if it is attached to another extensible component

that is extended;

$$R\$robot \Rightarrow put^{pl}(G\$grasper, O\$graspable, S\$surface)$$

$$\textbf{requires } configuration^{pl}(R\$robot, G\$component) = extended$$

$$\textbf{if } part^{pl}(R\$robot, G\$component), isa^{pl}(G, extensible)$$

$$R\$robot \Rightarrow put^{pl}(G\$grasper, O\$graspable, S\$surface)$$

$$\textbf{requires } configuration^{pl}(R\$robot, E\$component) = extended$$

$$\textbf{if } part^{pl}(R\$robot, G\$component), part^{pl}(R\$robot, E\$component),$$

$$sib^{pl}(G, E), isa^{pl}(E, extensible)$$

- A robot cannot lift a grasped object that has another object on top of it;

$$R\$robot \Rightarrow lift^{pl}(G\$grasper, O_1\$graspable)$$

$$\textbf{requires } on^{pl}(O_2\$placeable) \neq O_1\$surface, O_1 \neq O_2, O_1 \neq nothing$$

- A robot can only lift a block that it is grasping;

$$R\$robot \Rightarrow lift^{pl}(G\$grasper, O\$placeable)$$

$$\textbf{requires } grasping^{pl}(R\$robot, G\$grasper) = O\$graspable, O \neq nothing$$

- State Variable Relations:
  - An object that is grasped by a robot shares the location of the robot;

$$in^{pl}(R\$object) = in^{pl}(O\$object)$$

$$\textbf{if } grasping^{pl}(R\$robot, G\$grasper) = O\$graspable$$

  - A block that is on a table or the side of a table is the base of a tower;

$$tower\_base^{sl}(B\$block, T\$tower) = true \textbf{ if } on^{sl}(B\$block) = T\$surface$$

  - A block that is the base of a tower is in that tower;

$$in\_tower^{sl}(B\$block, B\$block) = true \textbf{ if } tower\_base^{sl}(B\$block) = true$$

– A block that is on top of another block is in the same tower as the block it is on;

$$in\_tower^{sl}(B_1\$block, B_2\$block) = true$$

$$\textbf{if } on^{sl}(B_1\$object) = B_2\$object,$$

$$in\_tower^{sl}(B_2\$block, B_3\$block) = true$$

– A tower that is not stacked in descending order from top to bottom is unordered;

$$unordered\_tower^{sl}(B\$block, T\$tower) = true$$

$$\textbf{if } on^{sl}(B_i\$object) = B_j\$object,$$

$$in\_tower^{sl}(B\$block, B_i\$block) = true,$$

$$in\_tower^{sl}(B\$block, B_j\$block) = true,$$

$$tower\_base^{sl}(B\$block, T\$tower) = true, B_i \neq B_j, i > j$$

– A tower is complete if it contains of the blocks of a particular colour and they are stacked in descending order from top to bottom;

$$complete_tower^{sl}(C\$colour, T\$tower) = true$$

$$\textbf{if } 3 = \#count\{B_i : in\_tower^{sl}(B\$block, B_i\$block) = true,$$

$$colour\_of^{sl}(B_i\$block, C\$colour)\},$$

$$unordered\_tower^{sl}(B\$block, T\$tower) = false,$$

$$tower\_base^{sl}(B\$block, T\$tower) = true,$$

$$colour\_of^{sl}(B\$block, C\$colour)$$

- State Variable Constraints:

– If any of a robot is grasping an object then that object must share its location;

$$\textbf{impossible } grasping^{sl}(R\$robot, G\$grasper) = O\$graspable,$$

$$in^{sl}(R\$object) \neq in^{sl}(O\$object), isa^{sl}(R\$robot), O \neq nothing$$

– If any of a robot's end effectors are aligned with an object then that object must

share its location;

$$\textbf{impossible } configuration^{sl}(G\$configurable) = aligned\_with(O\$graspable),$$
$$in^{sl}(R\$object) \neq in^{sl}(O\$object),$$
$$part^{pl}(R\$robot, G\$component), isa^{sl}(R\$robot)$$

– If a robot is grasping an object with an end effector then that end effector must be aligned with the object;

$$\textbf{impossible } grasping^{sl}(R\$robot, G\$grasper) = O\$graspable,$$
$$configuration^{sl}(G\$configurable) \neq aligned\_with(O)$$

– An object must share the location of a surface it is on;

$$\textbf{impossible } on^{sl}(O\$placeable) = S\$surface, in^{sl}(O\$object) \neq in^{sl}(S\$object)$$

– A block cannot have more than one block on top of it;

$$\textbf{impossible } on^{sl}(B_1\$block, B_2\$block), on^{sl}(B_1\$block, B_3\$block), B_2 \neq B_3$$

– A block cannot be on a block that is on nothing;

$$\textbf{impossible } on^{sl}(B_1\$block, B_2\$block), on^{sl}(B_1\$block, nothing), B_2 \neq B_3$$

– A robot cannot be grasping a block that has another block on top of it;

$$\textbf{impossible } grasping^{sl}(R\$robot, G\$grasper, B_1\$block), on^{sl}(B_1\$block, B_2\$block)$$

– A robot cannot grasp an object with two graspers simultaneously;

$$\textbf{impossible } grasping^{sl}(G_1\$grasper, O\$graspable),$$
$$on^{sl}(G_2\$grasper, O\$graspable), G_1 \neq G_2$$

- State Abstraction Mappings:

– If an object is in a location that has no ancestors, then it is in that location at the previous abstraction level;

$$in^{pl+1}(O\$object) = L\$location$$
$$\textbf{if } in^{pl}(O\$Object) = L\$location$$

– If an object is in a location that is a descendant of some ancestor location, then that object is also located in the ancestor location;

$$in^{pl+1}(O\$object) = AL\$location$$

$$\textbf{if } in^{pl}(O\$Object) = DL\$location, des^{pl}(AL, DL)$$

– If an object has any descendants then those descendants share the location of the object;

$$in^{pl+1}(AO\$object) = L\$location$$

$$\textbf{if } in^{pl}(DO\$Object) = L\$location, des^{pl}(AO, DO)$$

– If an object has any descendants then the object shares the location of its descendants;

$$in^{pl+1}(DO\$object) = L\$location$$

$$\textbf{if } in^{pl}(AO\$Object) = L\$location, des^{pl}(AO, DO)$$

– If a robot is grasping a graspable object with any of its graspers which is itself a descendant of an ancestor grasper then the robot is also grasping the same object with the ancestor grasper;

$$grasping^{pl+1}(R\$robot, AG\$grasper) = O\$graspable$$

$$\textbf{if } grasping^{pl+1}(R\$robot, DG\$grasper) = O\$graspable, des^{pl}(AO, DO)$$

– If a robot is grasping a graspable object with a grasper that has no ancestors, then it is grasping that object with that grasper at the previous abstraction level;

$$grasping^{pl+1}(R\$robot, G\$grasper) = O\$graspable$$

$$\textbf{if } grasping^{pl+1}(R\$robot, G\$grasper) = O\$graspable$$

– A component has the same configuration state at the previous abstraction level;

$$configuration^{pl+1}(R\$robot, C\$component) = S\$state$$

$$\textbf{if } configuration^{pl}(R\$robot, C\$component) = S\$state$$

– If an object is on a surface that has no ancestor, then it is on that surface at the

previous abstraction level;

$$on^{pl+1}(O\$object) = AS\$surface$$

$$\textbf{if } on^{pl}(O\$Object) = DS\$surface, des^{pl}(AS, DS)$$

– If an object is on a surface that is a descendant of some ancestor surface, then that object is also on the ancestor surface;

$$on^{pl+1}(O\$object) = AS\$surface$$

$$\textbf{if } on^{pl}(O\$Object) = DS\$surface, des^{pl}(AS, DS)$$

– If locations that have no ancestors are connected, then they are connected at the previous abstraction level;

$$connected^{pl+1}(L_1\$location, L_2\$location) = true$$

$$\textbf{if } connected^{pl}(L_1\$location, L_2\$location) = true$$

– If descendant locations are connected then so are their ancestors;

$$connected^{pl+1}(AL_1\$location, AL_2\$location) = true$$

$$\textbf{if } connected^{pl}(DL_1\$location, DL_2\$location) = true,$$

$$desc^{pl}(AL_1, DL_1), desc^{pl}(AL_2, DL_2)$$

– If locations that have no ancestors are connected by a door, then they are connected by that door at the previous abstraction level;

$$connected\_by\_door^{pl+1}(L_1\$location, L_2\$location) = true$$

$$\textbf{if } connected\_by\_door^{pl}(L_1\$location, L_2\$location) = true$$

– If descendant locations are connected by a door then so are their ancestors;

$$connected\_by\_door^{pl+1}(AL_1\$location, AL_2\$location) = true$$

$$\textbf{if } connected\_by\_door^{pl}(DL_1\$location, DL_2\$location) = true,$$

$$desc^{pl}(AL_1, DL_1), des^{pl}(AL_2, DL_2)$$

References

Aker, Erdi, Volkan Patoglu, and Esra Erdem (2012). "Answer Set Programming for Reasoning with Semantic Knowledge in Collaborative Housekeeping Robotics". In: *IFAC Proceedings Volumes* 45.22, pp. 77–83.

Alford, Ron et al. (2016). "Hierarchical Planning: Relating Task and Goal Decomposition with Task Sharing." In: *IJCAI*. https://pdfs.semanticscholar.org/877e/a69187dfc05129c574e56e4
pdf, pp. 3022–3029.

Alviano, Mario et al. (Sept. 2013). "WASP: a native ASP solver based on constraint learning". In: DOI: 10.1007/978-3-642-40564-8_6.

Andres, Benjamin, Philipp Obermeier, et al. (2013). "ROSoClingo: A ROS package for ASP-based robot control". In: *CoRR* abs/1307.7398.

Andres, Benjamin, David Rajaratnam, et al. (2015). "Integrating ASP into ROS for reasoning in robots". In: *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, pp. 69–82.

Andrew, A. M. (2004). "Knowledge Representation, Reasoning and Declarative Problem Solving". In: *Kybernetes* 33.

Babb, Joseph (2015). "Action Language BC +". In.

Bacchus, Fahiem and Qiang Yang (1994). "Downward refinement and the efficiency of hierarchical problem solving". In: *Artificial Intelligence* 71.1, pp. 43–100.

Balai, Evgenii (2016). "Combining Logic and Probability: P-log Perspective". In: *IJCAI*. https://www.ijcai.org/Proceedings/16/Papers/561.pdf.

Balai, Evgenii, Michael Gelfond, and Yuanlin Zhang (2013a). "SPARC - Sorted ASP with Consistency Restoring Rules". In: *CoRR* abs/1301.1386. https://arxiv.org/pdf/1301.1386.pdf.

— (2013b). "Towards Answer Set Programming with Sorts". In: *LPNMR*. http://www.depts.ttu.edu/cs/research/documents/20.pdf.

— (Feb. 2019). "P-log: refinement and a new coherency condition". In: *Annals of Mathematics and Artificial Intelligence*. https://link.springer.com/content/pdf/10.1007%2Fs10472-019-09620-2.pdf, pp. 1–44.

Balduccini, Marcello (2007). "cr-models: An Inference Engine for CR-Prolog". In: *LPNMR*, pp. 18–30.

Balduccini, Marcello and Michael Gelfond (2003). "Logic Programs with Consistency-Restoring Rules". In.

— (Jan. 2008). "The AAA architecture: An overview". In: https://pdfs.semanticscholar.org/af3f/85072b5576239797286904c0e3751adf0715.pdf, pp. 1–6.

Balduccini, Marcello, Daniele Magazzeni, et al. (2018). "CASP Solutions for Planning in Hybrid Domains". In: *CoRR* abs/1704.03574. https://arxiv.org/pdf/1704.03574.pdf.

Balduccini, Marcello, William C. Regli, and Duc N. Nguyen (2014). "An ASP-Based Architecture for Autonomous UAVs in Dynamic Environments: Progress Report". In: *CoRR* abs/1405.1124.

Ballard, Dana H and Leo Hartman (1986). "Task frames: Primitives for sensory-motor coordination". In: *Computer Vision, Graphics, and Image Processing* 36.2-3, pp. 274–297.

Banbara, Mutsunori et al. (2017). "Clingcon: The Next Generation". In: *TPLP* 17. https://www.cs.uni-potsdam.de/wv/publications/DBLP_journals/tplp/BanbaraKOS17.pdf, pp. 408–461.

Baral, Chitta (2003). *Knowledge Representation, Reasoning and Declarative Problem Solving with Answer Sets*. Cambridge University Press. DOI: 10.1017/CBO9780511543357.

Baral, Chitta, Michael Gelfond, and Nelson Rushton (Jan. 2009). "Probabilistic Reasoning with Answer Sets". In: *Theory Pract. Log. Program.* 9.1. http://www.public.asu.edu/~cbaral/papers/plogJune20-08.pdf, pp. 57–144.

Bercher, Pascal, Ron Alford, and Daniel Höller (2019). "A Survey on Hierarchical Planning-One Abstract Idea, Many Concrete Realizations." In: *IJCAI*, pp. 6267–6275.

Bishop and Christopher M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag. ISBN: 0387310738.

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

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

Buccafurri, Francesco, Nicola Leone, and Pasquale Rullo (1997). "Strong and Weak Constraints in Disjunctive Datalog". In: https://link.springer.com/content/pdf/10.1007%2F3-540-63255-7_2.pdf.

— (Sept. 2000). "Enhancing Disjunctive Datalog by Constraints". In: *IEEE Trans. on Knowl. and Data Eng.* 12.5, pp. 845–860. ISSN: 1041-4347.

Bylander, Tom (1994a). "The computational complexity of propositional STRIPS planning". In: *Artificial Intelligence* 69.1-2, pp. 165–204.

— (1994b). "The computational complexity of propositional STRIPS planning". In: *Artificial Intelligence* 69.1-2. https://ai.dmi.unibas.ch/_files/teaching/hs19/po/misc/bylander-aij1994.pdf, pp. 165–204.

Calimeri, Francesco, Wolfgang Faber, et al. (2012). "ASP-Core-2: Input language format". In: *ASP Standardization Working Group*.

Calimeri, Francesco, Wolfgang Fabery, et al. (2005). "Declarative and Computational Properties of Logic Programs with Aggregates". In: *Proceedings of the 19th International Joint Conference on Artificial Intelligence*. IJCAI'05. Edinburgh, Scotland: Morgan Kaufmann Publishers Inc., pp. 406–411. URL: http://dl.acm.org/citation.cfm?id=1642293.1642358.

Cashmore, Michael et al. (Jan. 2015). "Rosplan: Planning in the robot operating system". In: *Proceedings International Conference on Automated Planning and Scheduling, ICAPS* 2015. https://pdfs.semanticscholar.org/ca6c/e423ca1034dec71261d7b4a4524ac66b1a2f.pdf, pp. 333–341.

Chen, Kai, Fangkai Yang, and Xiaoping Chen (2016). "Planning with Task-Oriented Knowledge Acquisition for a Service Robot". In: *IJCAI*. https://www.ijcai.org/Proceedings/16/Papers/120.pdf, pp. 812–818.

Chen, Yixin, Chih-Wei Hsu, and Benjamin W Wah (2004a). "SGPlan: Subgoal partitioning and resolution in planning". In: *Edelkamp et al.(Edelkamp, Hoffmann, Littman, & Younes, 2004)*.

— (2004b). "SGPlan: Subgoal partitioning and resolution in planning". In: *Edelkamp et al.(Edelkamp, Hoffmann, Littman, & Younes, 2004)*. Saved.

Chinchalkar, Shirish (1996). "An upper bound for the number of reachable positions". In: *ICGA Journal* 19.3, pp. 181–183.

Clark, Keith L (1978). "Negation as failure". In: *Logic and data bases*. Springer, pp. 293–322.

Culberson, Joseph C and Jonathan Schaeffer (1998). "Pattern databases". In: *Computational Intelligence* 14.3, pp. 318–334.

Davis, Martin, George Logemann, and Donald Loveland (July 1962). "A Machine Program for Theorem-proving". In: *Commun. ACM* 5.7, pp. 394–397. ISSN: 0001-0782. DOI: 10.1145/368273.368557. URL: http://doi.acm.org/10.1145/368273.368557.

De Raedt, Luc (2008). *Logical and relational learning*. Springer Science & Business Media.

— (2010). "Inductive logic programming". In: *Encyclopedia of machine learning*. https://core.ac.uk/download/pdf/34496633.pdf, pp. 529–537.

Dell'Armi, Tina et al. (2003). "Aggregate Functions in DLV". In: *Answer Set Programming*.

Dimopoulos, Yannis et al. (2017). "plasp 3: Towards effective ASP planning". In: *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, pp. 286–300.

Dix, Jürgen, Ugur Kuter, and Dana Nau (2003). "Planning in answer set programming using ordered task decomposition". In: *Annual Conference on Artificial Intelligence*. Springer, pp. 490–504.

Dodaro, Carmine, Giuseppe Galatà, et al. (2018). "An Overview of ASP Applications in the Health-care Domain". In: *RiCeRcA@AI*IA*. https://pdfs.semanticscholar.org/aee3/64fb3b86534c81add1e079f0146129573741.pdf?_ga=2.232518047.1651806985.1554656987-369975188.1540140277.

Dodaro, Carmine, Philip Gasteiger, et al. (2016). "Combining Answer Set Programming and Domain Heuristics for Solving Hard Industrial Problems (Application Paper)". In: *CoRR* abs/1608.00730. https://arxiv.org/pdf/1608.00730.pdf.

Dovier, Agostino et al. (Jan. 2015). "Parallel execution of the ASP computation - An investigation on GPUs". In: *CEUR Workshop Proceedings* 1433.

Edelkamp, Stefan (2014). "Planning with pattern databases". In: *Sixth European Conference on Planning*.

Eiter, Thomas and Weston R. Faber (2000). "Declarative problem-solving using the DLV system". In: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.3.5125&rep=rep1&type=pdf.

Eiter, Thomas, Nicola Leone, et al. (1998). "The KR system dlv: progress report, comparisons and benchmarks". In: *KR 1998*. https://pdfs.semanticscholar.org/08a8/1b4a3f427bf65950d13abfdf0e1740a5c013.pdf?_ga=2.37730212.52501231.1554124483-369975188.1540140277.

Eiter, Thomas, Zeynep G Saribatur, and Peter Schüller (2019). "Abstraction for zooming-in to unsolvability reasons of grid-cell problems". In: *arXiv preprint arXiv:1909.04998*.

Erdem, Esra, Halit Erdogan, and Umut Öztok (2011). "BIOQUERY-ASP: Querying Biomedical Ontologies using Answer Set Programming". In: *RuleML America*.

Erdem, Esra and Volkan Patoglu (2018). "Applications of ASP in Robotics". In: *KI - Künstliche Intelligenz* 32, pp. 143–149.

Erol, Kutluhan, James Hendler, and Dana S Nau (1994a). "HTN planning: Complexity and Expressivity". In: *AAAI*. Vol. 94. https://www.aaai.org/Papers/AAAI/1994/AAAI94-173.pdf, pp. 1123–1128.

— (1994b). "HTN planning: Complexity and expressivity". In: *AAAI*. Vol. 94, pp. 1123–1128.

— (1996). "Complexity results for HTN planning". In: *Annals of Mathematics and Artificial Intelligence* 18.1, pp. 69–93.

Erol, Kutluhan, James A Hendler, and Dana S Nau (1995). *Semantics for hierarchical task-network planning*. Tech. rep. MARYLAND UNIV COLLEGE PARK INST FOR SYSTEMS RESEARCH.

Esra Erdem Michael Gelfond, Nicola Leone (2016). "Applications of Answer Set Programming". In.

Evangelou, George et al. (2021). "An approach for task and action planning in human–robot collaborative cells using AI". In: *Procedia Cirp* 97, pp. 476–481.

Falkner, Andreas et al. (Aug. 2018). "Industrial Applications of Answer Set Programming". In: *KI - Kunstliche Intelligenz* 32.2. https://link.springer.com/content/pdf/10.1007%2Fs13218-018-0548-6.pdf, pp. 165–176.

Fikes, Richard and Nils J. Nilsson (Dec. 1971). "STRIPS: A new approach to the application of theorem proving to problem solving". In: *Artificial Intelligence* 2, pp. 189–208. DOI: 10.1016/0004-3702(71)90010-5.

Gebser, Martin, Carito Guziolowski, et al. (Jan. 2010). "Repair and Prediction (under Inconsistency) in Large Biological Networks with Answer Set Programming". In.

Gebser, Martin, Amelia Harrison, et al. (2015). "Abstract gringo". In: *Theory and Practice of Logic Programming* 15.4-5, pp. 449–463. DOI: 10.1017/S1471068415000150.

Gebser, Martin, Tomi Janhunen, et al. (2015). "ASP Solving for Expanding Universes". In: *LPNMR*. https://www.cs.uni-potsdam.de/wv/publications/DBLP_conf/lpnmr/GebserJJKS15.pdf.

Gebser, Martin, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, et al. (2019a). *Potassco - Clingo*. https://github.com/potassco/clingo. Accessed: 01/04/2019. University of Potsdam.

— (Jan. 2019b). *Potassco User Guide*. 2nd. Available at: https://github.com/potassco/guide/releases, Accessed: 18/01/2021. University of Potsdam. Am Neuen Palais 10, 14469 Potsdam, Germany.

— (2022a). *Clingo Python Module version 5.3.0*. https://potassco.org/clingo/python-api/current/clingo.html. Accessed: 01/04/2019. University of Potsdam.

— (Jan. 2022b). *Potassco User Guide*. 2.2.0. Available at: https://github.com/potassco/guide/releases, Accessed: 28/01/2022. University of Potsdam. Am Neuen Palais 10, 14469 Potsdam, Germany.

Gebser, Martin, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, et al. (Dec. 2008). "Engineering an Incremental ASP Solver". In: https://link.springer.com/content/pdf/10.1007/978-3-540-89982-2_23.pdf, pp. 190–205.

Gebser, Martin, Roland Kaminski, Benjamin Kaufmann, Javier Romero, et al. (2015). "Progress in clasp Series 3". In: *LPNMR*. https://www.cs.uni-potsdam.de/wv/publications/DBLP_conf/lpnmr/GebserKK0S15.pdf.

Gebser, Martin, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub (2009). "On the Implementation of Weight Constraint Rules in Conflict-Driven ASP Solvers". In: *ICLP*. https://www.cs.uni-potsdam.de/wv/publications/DBLP_conf/iclp/GebserKKS09.pdf.

— (2012). *Answer Set Solving in Practice*. Morgan & Claypool Publishers. ISBN: 1608459713.

— (2014). "Clingo = ASP + Control: Extended Report". In: *CoRR* abs/1405.3694.

— (2019). "Multi-shot ASP solving with clingo". In: *TPLP* 19, pp. 27–82.

Gebser, Martin, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub, and Bettina Schnor (May 2011). "Cluster-Based ASP Solving with claspar". In: vol. 6645. https://www.researchgate.net/publication/221306925_Cluster-Based_ASP_Solving_with_claspar, pp. 364–369. DOI: 10.1007/978-3-642-20895-9_42.

Gebser, Martin, Roland Kaminski, Arne König, et al. (2011). "Advances in Gringo Series 3". In: *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning*. LPNMR'11. https://core.ac.uk/download/pdf/143868665.pdf. Vancouver, Canada: Springer-Verlag, pp. 345–351. ISBN: 978-3-642-20894-2. URL: http://dl.acm.org/citation.cfm?id=2010192.2010238.

Gebser, Martin, Roland Kaminski, Philipp Obermeier, et al. (Jan. 2015). "Ricochet Robots Reloaded: A Case-study in Multi-shot ASP Solving". In: 9060. https://iccl.inf.tu-dresden.de/w/images/4/40/Gekaobsc15a.pdf.

Gebser, Martin, Benjamin Kaufmann, and Torsten Schaub (Sept. 2009). "The Conflict-Driven Answer Set Solver clasp: Progress Report". In: vol. 5753. https://core.ac.uk/download/pdf/143866143.pdf.

— (Aug. 2012a). "Conflict-driven Answer Set Solving: From Theory to Practice". In: *Artificial Intelligence* 187-188. https://core.ac.uk/download/pdf/143877093.pdf, pp. 52–89.

— (2012b). "Multi-threaded ASP Solving with clasp". In: *CoRR* abs/1210.3265. URL: %5Curl%7Bhttps://www.cs.uni-potsdam.de/wv/publications/DBLP_journals/tplp/GebserKS12.pdf%7D.

Gebser, Martin, Nicola Leone, et al. (2018). "Evaluation Techniques and Systems for Answer Set Programming: a Survey". In: *IJCAI*.

Gebser, Martin, Marco Maratea, and Francesco Ricca (2017a). "The Design of the Seventh Answer Set Programming Competition". In: *Logic Programming and Nonmonotonic Reasoning*. Ed. by Marcello Balduccini and Tomi Janhunen. Cham: Springer International Publishing, pp. 3–9.

— (Sept. 2017b). "The Sixth Answer Set Programming Competition". In: *J. Artif. Int. Res.* 60.1, pp. 41–95. ISSN: 1076-9757. URL: http://dl.acm.org/citation.cfm?id=3207692.3207694.

— (2019). "The Seventh Answer Set Programming Competition: Design and Results". In: *CoRR* abs/1904.09134.

Gebser, Martin, Torsten Schaub, and Sven Thiele (May 2007). "GrinGo: A New Grounder for Answer Set Programming". In: vol. 4483. https://www.cs.uni-potsdam.de/wv/publications/DBLP_conf/lpnmr/GebserST07.pdf, pp. 266–271. DOI: 10.1007/978-3-540-72200-7_24.

Gelfond, Michael and Daniela Inclezan (2013). "Some properties of system descriptions of ALd". In: *Journal of Applied Non-Classical Logics* 23, pp. 105–120.

Gelfond, Michael and Yulia Kahl (2014a). *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. New York, NY, USA: Cambridge University Press. ISBN: 1107029562, 9781107029569.

— (2014b). *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. New York, NY, USA: Cambridge University Press. ISBN: 1107029562, 9781107029569.

Gelfond, Michael and Vladimir Lifschitz (1988). "The Stable Model Semantics for Logic Programming". In: *Proceedings of International Logic Programming Conference and Symposium*. Ed. by Robert Kowalski, Bowen, and Kenneth. MIT Press, pp. 1070–1080.

— (Aug. 1991). "Classical Negation in Logic Programs and Disjunctive Databases". In: *New Generation Computing* 9, pp. 365–385. DOI: 10.1007/BF03037169.

— (1998). "Action Languages". In: *Electronic Transactions on Artificial Intelligence* 3, pp. 195–210.

Georgievski, Ilche and Marco Aiello (2015a). "HTN planning: Overview, comparison, and beyond". In: *Artificial Intelligence* 222, pp. 124–156.

— (2015b). "HTN planning: Overview, comparison, and beyond". In: *Artificial Intelligence* 222. https://www.sciencedirect.com/science/article/pii/S0004370215000247, pp. 124–156.

— (2015c). "HTN planning: Overview, comparison, and beyond". In: *Artificial Intelligence* 222. https://www.sciencedirect.com/science/article/pii/S0004370215000247, pp. 124–156.

Ghallab, Malik, Craig Knoblock, et al. (Aug. 1998). "PDDL - The Planning Domain Definition Language". In.

Ghallab, Malik, Dana Nau, and Paolo Traverso (2004). *Automated Planning: theory and practice*. https://www.cs.umd.edu/~nau/cmsc421/planning.pdf. Elsevier.

— (2016). *Automated planning and acting*. https://paradise.fi.muni.cz/data/pres/2018-autumn/11_planning_and_acting.pdf. Cambridge University Press.

Giunchiglia, Fausto (1999). "Using Abstrips Abstractions–Where do We Stand?" In: *Artificial Intelligence Review* 13.3. https://www.researchgate.net/profile/Fausto_Giunchiglia/publication/220637969_Using_Abstrips_Abstractions_-_Where_do_We_Stand/links/552c571e0cf29b22c9c445e6.pdf, pp. 201–213.

Giunchiglia, Fausto and Toby Walsh (1992). "A theory of abstraction". In: *Artificial intelligence* 57.2-3, pp. 323–389.

Grasso, Giovanni, Nicola Leone, and Francesco Ricca (2013). "Answer Set Programming: Language, Applications and Development Tools". In: *Web Reasoning and Rule Systems*. Ed. by Wolfgang Faber and Domenico Lembo. https://www.mat.unical.it/ricca/downloads/rr2013-tutorial.pdf. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 19–34.

Guerrilla (2017). https://www.guerrilla-games.com/read/the-ai-of-horizon-zero-dawn.

Guo, Huihui et al. (2023). "Recent trends in task and motion planning for robotics: A survey". In: *ACM Computing Surveys* 55.13s, pp. 1–36.

Gupta, Naresh and Dana S Nau (1992). "On the complexity of blocks-world planning". In: *Artificial Intelligence* 56.2-3, pp. 223–254.

Hart, Peter E, Nils J Nilsson, and Bertram Raphael (1968). "A formal basis for the heuristic determination of minimum cost paths". In: *IEEE transactions on Systems Science and Cybernetics* 4.2, pp. 100–107.

Hayes, Bradley and Brian Scassellati (2016). "Autonomously constructing hierarchical task networks for planning and human-robot collaboration". In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 5469–5476.

Helmert, Malte (2003). "Complexity results for standard benchmark domains in planning". In: *Artificial Intelligence* 143.2, pp. 219–262.

Helmert, Malte, Patrik Haslum, Jörg Hoffmann, et al. (2007). "Flexible Abstraction Heuristics for Optimal Sequential Planning." In: *ICAPS*, pp. 176–183.

Helmert, Malte, Patrik Haslum, and Jörg Hoffmann (2008). "Explicit-State Abstraction: A New Method for Generating Heuristic Functions." In: *AAAI*, pp. 1547–1550.

Hoffmann, Jörg (2005). "Where 'ignoring delete lists' works: Local search topology in planning benchmarks". In: *Journal of Artificial Intelligence Research* 24. https://arxiv.org/pdf/1109.5713.pdf, pp. 685–758.

Hoffmann, Jörg, Ashish Sabharwal, and Carmel Domshlak (2006). "Friends or Foes? An AI Planning Perspective on Abstraction and Search." In: *ICAPS*, pp. 294–303.

Jiang, Yu-qian et al. (2019). "Task planning in robotics: an empirical comparison of PDDL- and ASP-based systems". In: *Frontiers of Information Technology & Electronic Engineering* 20.3, pp. 363–373.

Kaminski, Roland, Torsten Schaub, and Philipp Wanko (June 2017). "A Tutorial on Hybrid Answer Set Solving with clingo". In: pp. 167–203.

Kamperis, Oliver Michael (2020). *An ASP based Abstraction Planner*. https://github.com/OllieKampo/ASH_Planner. University of Birmingham.

— (2023). *The ASP based Hierarchical Conformance Refinement Planner*. https://github.com/OllieKampo/ASH-Planner. University of Birmingham.

Kelly, John Paul, Adi Botea, Sven Koenig, et al. (2008). "Offline Planning with Hierarchical Task Networks in Video Games". In: *AIIDE*. https://www.aaai.org/Papers/AIIDE/2008/AIIDE08-010.pdf.

Kelly, John-Paul, Adi Botea, Sven Koenig, et al. (2007). "Planning with hierarchical task networks in video games". In: *Proceedings of the ICAPS-07 Workshop on Planning in Games*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.4477&rep=rep1&type=pdf.

Khandelwal, Piyush et al. (2017). "Bwibots: A platform for bridging the gap between ai and human–robot interaction research". In: *The International Journal of Robotics Research* 36.5-7, pp. 635–659.

Knoblock, Craig A (1990a). "A theory of abstraction for hierarchical planning". In: *Change of Representation and Inductive Bias*. Springer, pp. 81–104.

— (1990b). "Learning Abstraction Hierarchies for Problem Solving." In: *AAAI*, pp. 923–928.

— (1991). *Automatically Generating Abstractions for Problem Solving*. Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.

— (1992). "An analysis of ABSTRIPS". In: *Artificial Intelligence Planning Systems*. Elsevier, pp. 126–135.

Knoblock, Craig A, Josh D Tenenberg, and Qiang Yang (1991). "Characterizing Abstraction Hierarchies for Planning". In: *AAAI*, pp. 692–697.

Koller, Daphne and Nir Friedman (2009). *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press. ISBN: 0262013193, 9780262013192.

Koller, Daphne, Nir Friedman, et al. (2007). *Introduction to statistical relational learning*. MIT press.

Korf, Richard E (1987). "Planning as search: A quantitative approach". In: *Artificial intelligence* 33.1, pp. 65–88.

Lee, Joohyung, Vladimir Lifschitz, and Fangkai Yang (2013). "Action Language BC: Preliminary Report". In: *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*.

Leone, Nicola et al. (2006). "The DLV system for knowledge representation and reasoning". In: *ACM Trans. Comput. Log.* 7. https://arxiv.org/pdf/cs/0211004.pdf, pp. 499–562.

Lifschitz, Vladimir (2002). "Answer set programming and plan generation". In: *Artificial Intelligence* 138.1-2. https://core.ac.uk/download/pdf/82211403.pdf, pp. 39–54.

Lifschitz, Vladimir (2008). "What is Answer Set Programming?" In: *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*. AAAI'08. Chicago, Illinois, pp. 1594–1597. ISBN: 978-1-57735-368-3.

Marques-Silva, J. P. and K. A. Sakallah (May 1999). "GRASP: a search algorithm for propositional satisfiability". In: *IEEE Transactions on Computers* 48.5, pp. 506–521. ISSN: 0018-9340. DOI: 10.1109/12.769433.

McCluskey, Thomas Leo (2000). "Object Transition Sequences: A New Form of Abstraction for HTN Planners." In: *AIPS*. https://pdfs.semanticscholar.org/c7c4/784c8b0ef09a830b2cb1b982pdf, pp. 216–225.

Meadows, Ben, Mohan Sridharan, and Zenon Colaco (2016). "Towards an Explanation Generation System for Robots: Analysis and Recommendations". In: https://www.cs.bham.ac.uk/~sridharm/Papers/robotics16_explainCompare.pdf.

Mellarkod, Veena S., Michael Gelfond, and Yuanlin Zhang (Aug. 2008). "Integrating Answer Set Programming and Constraint Logic Programming". In: *Annals of Mathematics and Artificial Intelligence* 53.1-4. http://isaim2008.unl.edu/PAPERS/SS1-AI+Logic/MGelfond-ss1.pdf, pp. 251–287.

Menkes Van Den Briel, Romeo Sanchez, Minh B Do, and Subbarao Kambhampati (2004). "Effective approaches for partial satisfaction (over-subscription) planning". In: *Proceedings of the 19th national conference on Artifical intelligence*. https://www.aaai.org/Papers/AAAI/2004/AAAI04-090.pdf, pp. 562–569.

Mirkis, Vitaly and Carmel Domshlak (2013). "Abstractions for oversubscription planning". In: *Twenty-Third International Conference on Automated Planning and Scheduling*. http://icaps13.icaps-conference.org/documents/dc/VitalyMirkis.pdf.

Nau, Dana S et al. (2003). "SHOP2: An HTN planning system". In: *Journal of artificial intelligence research* 20. https://arxiv.org/pdf/1106.4869.pdf, pp. 379–404.

Nau, Dana S. (2019). *Pyhop version 1.2.2*. https://bitbucket.org/dananau/pyhop/src/default/. Accessed: 10/10/2019. University of Maryland.

Nayak, P Pandurang and Alon Y Levy (1995). "A Semantic Theory of Abstractions". In: *IJCAI*. https://pdfs.semanticscholar.org/6655/e30da9faa486b5a775ce6f13fdd3753789c0.pdf, pp. 196–203.

Newell, Allen, Herbert Alexander Simon, et al. (1972). *Human problem solving*. Vol. 104. Prentice-Hall Englewood Cliffs, NJ.

Nielsen, Mogens, Grzegorz Rozenberg, and Pazhamaneri S Thiagarajan (1990). "Elementary transition systems". In: *DAIMI Report Series* 310.

Niemelä, Ilkka (2008). "Answer set programming without unstratified negation". In: *International Conference on Logic Programming*. https://www.researchgate.net/profile/Ilkka_Niemelae/publication/220986022_Answer_Set_Programming_without_Unstratified_Negation/links/00b4951ba9a90858b6000000.pdf. Springer, pp. 88–92.

Niemelä, Ilkka, Patrik Simons, and Tommi Syrjänen (2000). "Smodels: A System for Answer Set Programming". In: *CoRR* cs.AI/0003033. URL: http://arxiv.org/abs/cs.AI/0003033.

Pearl, Judea (1984). *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc.

— (2000). *Causality: models, reasoning and inference*. Vol. 29. Springer.

Pommerening, Florian and Alvaro Torralba (2018). *International Planning Competition 2018*. https://ipc2018-classical.bitbucket.io/. Accessed: 11/12/2021.

Quigley, Morgan et al. (2009). "ROS: an open-source Robot Operating System". In.

Ricca, Francesco and Nicola Leone (2007). "Disjunctive logic programming with types and objects: The DLV+ system". In: *J. Applied Logic* 5. https://www.academia.edu/12647830/Disjunctive_logic_programming_with_types_and_objects_The_DLV_system, pp. 545–573.

Russell, Stuart J and Peter Norvig (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited.

Sacerdoti, Earl D (1974). "Planning in a hierarchy of abstraction spaces". In: *Artificial intelligence* 5.2, pp. 115–135.

Saribatur, Zeynep G (2020). "Abstraction for ASP Planning". In.

Saribatur, Zeynep G and Thomas Eiter (2018a). "Towards Abstraction in ASP with an Application on Reasoning about Agent Policies". In: *arXiv preprint arXiv:1809.06638*.

— (2018b). "Towards Abstraction in ASP with an Application on Reasoning about Agent Policies". In: *arXiv preprint arXiv:1809.06638*. https://arxiv.org/pdf/1809.06638.pdf.

— (2021). "Omission-based abstraction for answer set programs". In: *Theory and Practice of Logic Programming* 21.2, pp. 145–195.

Saribatur, Zeynep G, Volkan Patoglu, and Esra Erdem (2019). "Finding optimal feasible global plans for multiple teams of heterogeneous robots using hybrid reasoning: an application to cognitive factories". In: *Autonomous Robots* 43.1. https://publik.tuwien.ac.at/files/publik_283570.pdf, pp. 213–238.

Saribatur, Zeynep G, Peter Schüller, and Thomas Eiter (2019). "Abstraction for non-ground answer set programs". In: *European Conference on Logics in Artificial Intelligence*. Springer, pp. 576–592.

Shafranov, Alex (2019). *PyHTN: Python HTN Planner*. https://github.com/mrceresa/PyHTN. Accessed: 10/10/2019.

SHAKEY (1966-1972). *Milestone-Proposal: Shakey: The World's First Mobile, Intelligent Robot, 1972*. http://ieeemilestones.ethw.org/Milestone-Proposal:Shakey:_The_World%E2%80%99s_First_Mobile,_Intelligent_Robot,_1972. Accessed: 01/04/2019.

Shannon, Claude E (1950). "XXII. Programming a computer for playing chess". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41.314, pp. 256–275.

Shivashankar, Vikas (2015). "Hierarchical Goal Networks: Formalisms and Algorithms for Planning and Acting". In.

Shivashankar, Vikas et al. (2013). "Hierarchical goal networks and goal-driven autonomy: Going where ai planning meets goal reasoning". In: *Goal Reasoning: Papers from the ACS Workshop*. http://www.cs.umd.edu/~nau/papers/shivashankar2013hierarchical.pdf, p. 95.

SHOP (2019). *Simple Hierarchical Ordered Planner*. http://www.cs.umd.edu/projects/shop/index.html. Accessed: 24/07/2019.

Silva, João P. Marques and Karem A. Sakallah (1996). "GRASP&Mdash;a New Search Algorithm for Satisfiability". In: *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*. ICCAD '96. https://pdfs.semanticscholar.org/ab9e/db02517a570a9056600e02048092e63a0124.pdf. San Jose, California, USA: IEEE Computer Society, pp. 220–227. ISBN: 0-8186-7597-7. URL: http://dl.acm.org/citation.cfm?id=244522.244560.

Simons, Patrik, Ilkka Niemelá, and Timo Soininen (June 2002). "Extending and Implementing the Stable Model Semantics". In: *Artif. Intell.* 138.1-2.

Smith, David E (2004). "Choosing Objectives in Over-Subscription Planning". In: *ICAPS*. Vol. 4. https://www.aaai.org/Papers/ICAPS/2004/ICAPS04-046.pdf, p. 393.

Smith, David E and Mark A Peot (2014). "A critical look at Knoblock's hierarchy mechanism". In: *Proceedings of the 1st International Conference on Artificial Intelligence Planning Systems (AIPS92)*. http://de2smith.samling.us/publications/AIPS92-Knoblock-critique.pdf, pp. 307–308.

Sridharan, Mohan, Michael Gelfond, et al. (2019). "REBA: A Refinement-Based Architecture for Knowledge Representation and Reasoning in Robotics". In: *Journal of Artificial Intelligence Research* 65, pp. 87–180.

Sridharan, Mohan and Ben Meadows (2019). "A Theory of Explanations for Human-Robot Collaboration".

Sutton, Richard S. and Andrew G. Barto (2018). *Introduction to Reinforcement Learning*. 2nd. Cambridge, MA, USA: MIT Press. ISBN: 0262039249.

Syrjänen, Tommi (1998). *Implementation Of Local Grounding For Logic Programs With Stable Model Semantics*.

— (Sept. 2001). "Omega-Restricted Logic Programs". In: pp. 267–279. DOI: 10.1007/3-540-45402-0_20.

Tenenberg, Josh (1988). *Abstraction in planning*. Tech. rep. University of Rochester, Department of computer science.

Thrun, Sebastian, Wolfram Burgard, and Dieter Fox (2005). *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press. ISBN: 0262201623.

Timpf, Sabine et al. (1992). "A conceptual model of wayfinding using multiple levels of abstraction". In: *Theories and methods of spatio-temporal reasoning in geographic space*. Springer, pp. 348–367.

Wang, Weitian et al. (2018). "Robot action planning by online optimization in human–robot collaborative tasks". In: *International Journal of Intelligent Robotics and Applications* 2, pp. 161–179.

Washington, Richard (1994). *Abstraction Planning in Real Time*. Tech. rep. STANFORD UNIV CA DEPT OF COMPUTER SCIENCE.

Wichlacz, Julia, Álvaro Torralba, and Jörg Hoffmann (2019). "Construction-Planning Models in Minecraft". In.

Zhang, Shiqi, Mohan Sridharan, and Jeremy L. Wyatt (2015). "Mixed Logical Inference and Probabilistic Planning for Robots in Unreliable Worlds". In: *IEEE Transactions on Robotics* 31, pp. 699–713.

Zhang, Shiqi and Peter Stone (2015). "CORPP: Commonsense Reasoning and Probabilistic Planning, as Applied to Dialog with a Mobile Robot". In: *AAAI Spring Symposia*.

— (2017). "Integrated Commonsense Reasoning and Probabilistic Planning". In: https://pdfs.semanticscholar.org/dacc/34b8dcaa7afcaf672b2935510fa8632ebe65.pdf?_ga=2.50590505.1677710238.1556009678-369975188.1540140277.

Zhang, Shiqi, Fangkai Yang, et al. (2015). "Mobile Robot Planning Using Action Language $\mathcal{BC}$ with an Abstraction Hierarchy". In: *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, pp. 502–516.