

Conformance Refinement for Hierarchical Abstraction Planning

Oliver Michael Kamperis^{a,*}, Marco Castellani^{b,1} and Yongjing Wang^{b,1}

^aDepartment of Computer Science, University of Birmingham, UK

^bDepartment of Mechanical Engineering, University of Birmingham, UK

ARTICLE INFO

Keywords:

Automated planning
Hierarchical planning
Abstraction in reasoning

ABSTRACT

For complex real-world robotics applications, classical planning is too time consuming to be practical, and finding heuristics to guide search is often too difficult. This paper presents the theoretical foundations, implementation, and evaluation of a novel planning paradigm called conformance refinement, built in Answer Set Programming (ASP), a declarative non-monotonic logic programming paradigm. In conformance refinement, a planning domain is defined over a hierarchy of domain models at different levels of abstraction. Unlike typical heuristic search, search is guided by a constraint addition based approach, which requires that a plan generated over such a hierarchy achieves the same effects and remains structurally similar at all levels. This reduces search time by guiding the solver towards solutions that satisfy the constraint. Further, it enables online planning, in which a planning problem is divided into a sequence of partial problems, only the first of which must be solved prior to execution. This allows a robot to; find plans and begin execution, exponentially faster than by ASP based classical planning. The trade-off, is that plans are not guaranteed to be optimal. Simulated experiments have been run on an extension of the classic blocks world domain, involving a novel form of abstraction, called a condensed model, which views the planning domain in a more granular manner, to reduce search spaces. Results show that; offline planning reduced search time by 77.4-94.4% but increased plan lengths by 0.0-9.0%, and online planning reduced time before execution by 97.6-99.8% but could increase plans length by 0.0-32.0%, over classical planning, from our easiest to hardest problems.

1. Introduction

When robots are deployed into real-world domains, such as the residential or retail sectors, humans will expect them to act promptly on instructions given, and to complete tasks as quickly as possible. Robots deployed into the real-world must thus be able to rapidly find high quality plans, with little latency before execution. However, in the field of automated planning for robotics, such abilities are yet to be achieved.

In automated planning, a robot is given an abstract world description and must find a sequence of actions/operations, which if executed in order, will transition some initial state(s) to some desired goal state(s). This requires reasoning about and making a series of related decisions, prior to acting and into large future horizons, to optimise long-term goals.

Most existing planning algorithms are based on classical planning. Unfortunately, these cannot solve real-world problems quickly enough to be useful for any interesting practical applications [9]. This is primarily caused by two factors.


Firstly, the state space is usually exponential in problem description size [40]. This is common in combinatorics. E.g. in chess, up to $10^{46.25}$ valid states exist [12], the exact bound is unknown as enumerating them is not feasible. Real-world problems are very likely to have many orders more states.

Secondly, agents often have many legal actions per state. As a result, each state may lead to many others and thus many paths may exist between any two states. In a search graph or tree, the mean number of legal actions per state is its branching factor. Complexity of exhaustive search is $O(b^d)$, where b is branching factor and d depth of the shallowest goal state. Thus, for even small branching factors, the exponential increase in search space with search depth (i.e. minimum plan length), rapidly makes problems prohibitively complex. E.g. in chess, each player has “just” ≈ 30 average moves per turn, yet after only 5 turns, over $\approx 6.9 \times 10^{13}$ possible games exist [50]. Real-world problems will allow many more actions.

These issues are especially problematic in classical planning, because a complete plan must always be generated, and to ensure its optimality the search space must be exhausted. Further, the real-world is unpredictable and observability is limited, causing (particularly long) plans to fail very often. For large problems, with high search depths, classical planning is thus both highly infeasible and futile in reality.

The common solution for this issue is to use application-specific heuristics. This can achieve efficient algorithms that scale to big problems, e.g. A^* path finding [33]. Heuristics reduce search time by exploiting the nature of a specific application to guide search towards good solutions. However, finding heuristics (especially admissible ones) requires significant prior knowledge of possible problem instances. This makes heuristic planners highly specialised and non-general.

*Corresponding authors:

 o.m.kamperis@gmail.com (O.M. Kamperis)

¹These authors contributed equally, they are listed alphabetically.

This paper presents an approach to planning based upon the use of abstraction in reasoning, and which emulates the human method for solving planning problems. The proposed planner is called ASH, the Answer Set programming based Hierarchical abstraction planner. ASH is an online domain-independent planner which uses abstraction to generate high quality plans, extremely fast, without requiring heuristics.

ASH instead uses a novel hierarchical plan *conformance refinement* technique to automatically find and apply abstract search constraints in a general way, to give two benefits:

1. Search spaces are restricted, reducing overall planning time exponentially over ASP based classical planning.
2. Planning problems can optionally be divided into a sequence of partial problems, and solved incrementally and online, reducing execution latency exponentially.

The method is intuitive. A planning domain and problem is defined over an abstraction hierarchy, containing a unique model of the domain dynamics for each level. The planning problems specified by each level are solved iteratively, in descending order. Only at the most abstract level is a complete classical plan found. Such a plan is then successively *refined* at all lower levels, by enforcing a *conformance constraint* to search, which contains a sequence of *subgoal stages*, formed by the effects of actions planned at the previous level, which must be achieved in the same order at the next level. This restricts the search space, reducing planning time whilst maintaining high quality plans, and allows problems to be divided in partial problems, defined by any contiguous subsequence of subgoal stages. Solving just the first partial problem prior to execution reduces latency, but also long-term plan quality.

Creation of abstract models of a planning domain is fundamental to the approach. This involves deciding; which elements of a domain are details that can be abstracted away, or what assumptions can be made to simplify search. In this paper, abstraction is explored in two different settings:

1. *Relaxed Models*: Obtained by removing preconditions of actions and founded on assumptive reasoning. This model reduces plan lengths and search spaces [48].
2. *Condensed Models*: Our novel form of abstract model. The state space is made more granular by combining sets of related entities into abstract descriptors viewed as the union of the original entities that map to them, and which together form their whole. This model reduces state spaces, plan lengths, and search spaces.

Although only these types of abstract model are explored in this paper. The approach is shown to work for any abstract model type where a deductive mapping can be defined between the state spaces of the abstract and original models.

Answer Set Programming (ASP) was chosen as the basis for the approach due to its effectiveness for Knowledge Representation and Reasoning (KRR) [7]. In particular, ASP can deal with the large number of facts and relations required for planning [18]. We maintain that this makes ASP based KRR very well suited to representing abstraction hierarchies. To enable this capability, our work extends past ASP encodings, to represent the abstraction level of predicates explicitly.

2. Background

In the design and review of a planning paradigm or system, we maintain the most major factors to consider are;

1. *Expressivity*: What is the extent of the problems it can solve? Can it deal with the desired application?
2. *Efficiency*: Is it tractable? Does it find plans fast enough to be useful for the desired application?
3. *Optimality*: Is it guaranteed to find an optimal plan?

However, there is always a clear trade-off between these. No planner exists that achieves these all in a general manner.

The simplest type of automated planning is classical planning. A single finite sequence of actions is generated through a discrete deterministic state transition system. The seminal classical planner was the Stanford Research Institute Problem Solver (STRIPS) [19], which was used by “Shakey” the first ever intelligent mobile robot [49]. Its formulation is still in use by many now very expressive modern paradigms (e.g. PDDL [30]). However, classical planning is highly computationally expensive. Complexity is NP-hard as standard, reduced to polynomial with severe restrictions on expressivity, determining plan existence is PSPACE-complete [9]. Without powerful heuristics, classical planning is intractable.

2.1. Abstraction for Hierarchical Planning

The use of abstraction has been studied extensively and is widely accepted as an effective method for simplifying reasoning tasks for humans and Artificial Intelligence (AI) [46]. Humans employ abstraction extensively when solving problems, e.g. planning and giving explanations [31]. 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 [52].

Abstraction planning seeks to emulate these human abilities to greatly increase efficiency of the planner, (usually) at the expense of plan optimality. The concept is simple; only initially decide on the abstract goals or stages of a plan, deal with the details later and as they appear. Without this ability, the complexity of real-world planning is overwhelming.

An example of abstraction planning for a delivery robot problem is given by the directed graph in Figure 1. The goal is to deliver an item to each red node, starting from and returning to the blue node. The robot first generates an abstract plan by deciding the order to visit the red nodes, based only on their euclidean distance from blue, without considering what paths connect them. The numbered labels indicate this plan, which the robot treats as a series of way-points between which to travel. When the robot executes this abstract plan, it then only need find a path from the current to the next way-point. The solid black arcs are the complete resulting path.

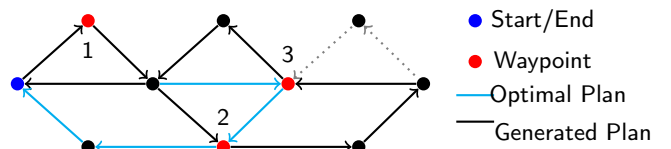


Figure 1: Navigation Way-point Planning Example

The important property is that finding the path between a consecutive pair of way-points is a subproblem or subtask which can be solved independently. Using abstraction in this manner to divide-and-conquer large planning problems can significantly reduce overall search complexity² [39, 53, 11]. The clear trade-off is that the obtained plan is non-optimal.

Abstraction planners of this form can be broadly placed in two groups; *decompositional* and *refinement* based techniques. We review these in the following subsections.

2.1.1. Decompositional Techniques

Decompositional techniques involve the hierarchical decomposition of tasks into subtasks [8]. The standard method is Hierarchical Task Network (HTN) planning [16]. This enriches classical planning with *task decomposition methods*.

HTN planners are given; concrete actions, abstract tasks, subtasks, and task decomposition methods. They start with an initial “network” of abstract tasks to complete. Its methods then define strategies for replacing these tasks, with (sequences of) subtasks or actions. The planner returns when a classical plan, which achieves the initial tasks, is found.

Decompositional planners are more expressive than classical, but are limited by their “brittleness” caused by the reliance on the completeness of their decompositional methods [17]. Significant domain knowledge is needed to define effective strategies for decomposing problems. Despite this, HTN planning is popular and many planners exist [29].

2.1.2. Refinement Techniques

Refinement techniques represent a classical planning domain over an abstraction hierarchy, containing multiple different abstract domain models. Planning involves first finding a classical plan at the most abstract level, which forms a “skeleton” plan, that is then successively *refined* at all lower levels, until the lowest level (ground level) is reached. The standard refinement process recursively divides a problem into multiple independent subproblems over the hierarchy.

The seminal refinement based abstraction planners were ABSTRIPS (extension of STRIPS) [48] and ALPINE [41]. The prior used *relaxed domain models* obtained by removing action preconditions, and the latter used *reduced domain models* obtained by removing preconditions and related facts. In both, each skeleton action forms a subproblem. Solving a subproblem *refines* the given skeleton action, by inserting (a sequence of) new actions (a subplan) at the next level, to satisfy any preconditions that were previously removed.

The basic intuition of refinement based planning is that, if the most abstract level is sufficiently simple, solving it by classical planning is tractable, as every level can reduce the search space exponentially [40]. Refining the skeleton plan is then exponentially faster than directly solving by classical planning, as each subplan is linearly shorter, and the search space of each subproblem exponentially less, than the complete plan/problem [5]. The sum of complexities of all subproblems is then exponentially less than the complete.

²This example is trivial to solve as a complete problem, but as its complexity increases (e.g. size of graph and number of red nodes) the ability to divide and solve its parts independently is exponentially more beneficial.

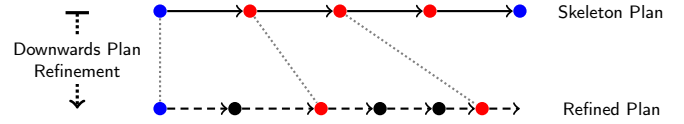


Figure 2: Way-point Plan Refinement Example

A plan refinement of this form, for the previous example, is indicated in Figure 2. The top level is the skeleton plan (of way-points), whose actions are passed to the ground level for refinement. The skeleton plan ignores a precondition requiring travel along only directed arcs (in Figure 1). Planning at the ground level then requires adding new actions, between consecutive pairs of skeleton actions, to satisfy this reintroduced precondition, i.e. finding a path to each way-point.

The trade-off is that each subplan is only locally optimal and the resulting complete plan (obtained by concatenating the subplans) is rarely globally optimal. This has occurred in our example. The subplan between each pair of way-points is optimal, but the complete plan is not. This is because ordering of way-points was decided on a naive basis of euclidean distance from start. It is common for ordering constraints enforced by skeleton plans of this form to cause this issue. The cyan paths on Figure 1 indicate “shortcuts” that can achieve the optimal plan, had the classical problem been solved.

Refinement based planners are generally faster and need less domain knowledge than decompositional planners. The main challenge is finding an effective abstraction hierarchy, that solves quickly and finds “good” quality plans. Bacchus provides a thorough debate on the challenges of designing these hierarchies [5]. Particularly, abstractions must be carefully balanced, extensive abstractions may increase planning speed at the abstract level, but the resulting skeleton plans might be more computationally challenging to refine. Further, existing refinement planners, e.g. ABSTRIPS, are inherently limited to abstractions based on preconditions.

The two main problems with abstractions for refinement planning, that reduce plan optimality and search efficiency, we classify as; the *ignorance* and *dependency* problems:

1. *The ignorance problem* occurs from the lost knowledge in the abstractions themselves³. Abstract domain models do not (by design) fully represent the original model, as details were removed to reduce complexity. Thus, the planner can make decisions at the abstract levels, that are non-optimal with respect to the ground level planning problem. As such, an optimal abstract plan might not refine to an optimal ground level plan, respective of the original classical planning problem.
2. *The dependency problem* occurs from the division of planning problems into subproblems, that are assumed mutually independent [43]. In practice however, this assumption often fails, indicating that the planner has failed to find an effective division of a problem. When there is a sufficient dependency between subproblems, skeleton plans may not be refineable at all, forcing the planner to “backtrack” across abstractions to revise its abstract plans, greatly degrading performance [42].

³This is the problem occurring in our way-point planning example.

2.2. Answer Set Programming

ASP is an established approach to non-monotonic logical declarative programming, that supports defeasible reasoning processes, including abductive and default reasoning [45]. It is effective at KRR for large knowledge bases and at solving highly complex combinatorial optimisation problems [23].

In contrast to classic logics, non-monotonic logic allows three truth values; *true*, *false*, and *unknown*, i.e. lack of support for truth does not imply falsehood. The special logical connective *not* symbolises *negation as failure* (*naf*). Unlike *classical negation*, which stands for “*p* is false”, *naf* stands for “*p* is not known to be true”, i.e. false or unknown [13].

2.2.1. Syntax and Semantics

An ASP logic program Π contains a set of logic rules of the abstract form $head \leftarrow body$. Intuitively, if a rule’s body is satisfied, then so must its head. An answer set of a program is a set of head atoms $\zeta = \{a_1, \dots, a_q\}$ which simultaneously satisfy all of its rules. If a program has at least one answer set it is *satisfiable*, otherwise it is *unsatisfiable* [26, 27].

The body of a rule is a logical conjunction and the head is either an atom or an aggregate⁴ [10]. E.g. as follows:

$$a_0 \leftarrow a_1, \dots, a_m, \text{not } a_{m+1}, \dots, \text{not } a_n \quad (1)$$

Where a is an *atom* of the form $p(t_1, \dots, t_r)$, abbreviated to $p(\bar{t}_o)$, or its classical negation $\neg p(\bar{t}_o)$, p is a *predicate* of arity o , and t is a term. A term is some 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 $d(\bar{t}_o)$. The atoms a_0 is its head, a_1, \dots, a_m and $\text{not } a_{m+1}, \dots, \text{not } a_n$ are *positive* and *negative body literals*, respectively. Intuitively, the head is satisfied *iff* $a_0 \in \zeta$, i.e. its atom is in the answer set. A positive literal is satisfied *iff* $a_i \in \zeta$. A negative literal is satisfied *iff* $a_i \notin \zeta$, i.e. its atom is not in the answer set.

A rule with a head atom and at least one body literal is a *normal* rule. Normal rules may be used in planning to define action effects, e.g. the axiom “when a robot moves its location changes”. A rule with a head atom and no body is a *fact*, all answer sets must satisfy its head as the body is always satisfied. Facts may be used to declare available actions and the state variables. A rule with no head atom and at least one body literal is called an *integrity constraint*, no answer set may satisfy its body as the head can never be satisfied. Integrity constraints may be used to define state constraints, e.g. the axiom “two objects cannot be in the same place”.

2.2.2. Planning with ASP

ASP is highly applicable for discrete deterministic planning. It supports the complex reasoning capabilities and can deal with large numbers of facts and relations, required by a planning agent, e.g. deriving the indirect effects of actions or recursive variable relations. Conversely, these are very difficult to represent in standard programming languages, e.g. Python. In ASP, a planning domain and problem is encoded as a logic program, whose answer sets are its solutions [25].

⁴Aggregates are used to express more complex set operations, e.g. cardinality and weight constraints. See [11] for a review of aggregates in ASP.

The paradigm is unlike classical rule based expert systems. The program rules do not tell the robot what to do or how to complete any given task. Instead, they axiomatically specify the robots’ capabilities, and the laws that govern the system they operate in, e.g. the effects and preconditions of actions. The ASP solver searches for transitions of that system which satisfy all program rules, and solve the planning problem, via satisfiability solving [11,11], doing away with the need for classic tree or graph search algorithms, e.g. A* search.

We maintain that ASP has three benefits for planning:

1. *Complete*, guaranteed to return despite satisfiability.
2. *Optimal*, guaranteed to find optimal plan if satisfiable.
3. *Expressive*, versatile and elaboration tolerant coding.

One of the few trade-offs is that ASP is ineffective at dealing with action costs⁵, and is not applicable for continuous costs.

Classical planning in ASP has been used in many robotics domains [15]. E.g. simulated teams of house keeping robots [1], delivery robots [2, 3], unmanned air vehicles [6], general PDDL planning [14], and real-world service robots [38].

Despite this success, existing approaches to planning using ASP continue to prove highly intractable for non-trivial problems, especially those with long plan lengths [36]. Some attempts have been made towards supporting refinement-like planning in ASP based “action languages”⁶ in an attempt to overcome this. However, these have had limited success.

A three-level abstraction hierarchy written in the action language *BC* [44] was used for logistics [55]. Abstract plans were used to “bottleneck” search at lower levels, using an alternative constraint addition based approach to refinement, supported by the nature of ASP based satisfiability solving. This gave “orders of magnitude” faster planning for 11.25% loss of plan optimality. However, this had major limitations:

1. They give no clear structure for designing abstraction hierarchies, or representing hierarchical plans. Thus, their planning domains appear obscure and ill-defined.
2. A minimum step bound for the length of the subplan that refines each abstract action must be pre-generated. This amounts to already knowing all possible action refinements prior to planning. This caveat is ignored, and makes their technique and results inadmissible.

A two-level “coupled” transition system written in action language *AL_d* [24] was used for mixed ASP and probabilistic navigation [54, 51]. They proposed an abstract model that reduced the state space, by making the representation more granular. This broke down navigation planning to finding an abstract plan between rooms, that was then refined to a plan between cells of rooms. However, this also had limitations:

1. Only two abstractions are supported. The theory does not generalise to larger hierarchies, primarily because abstraction levels were not represented in predicates.
2. Abstract models are defined only by relations between constants, and not their class types. As a result, additional abstract domain rules must be given manually.

⁵For this reason, we do not consider any action costed problems here.

⁶An action description language [28] is a high-level natural language notation for ASP programs used for describing state transition systems [4].

3. ASH the Hierarchical Abstraction Planner

ASH is a domain-independent abstraction planner which uses a novel technique we call *conformance refinement*. Instead of refining plans and defining abstract models based on action preconditions, conformance refinement refines plans based on action effects and allows any abstract model where a state space mapping exists from the model it abstracts.

The approach combines the benefits and overcomes the limitations of problem division and constraint addition based refinement planning, making the methods suitable for online planning. The most major distinctions over past work are;

1. Refinements are defined by *conformance constraints*. These enforce that low level plans achieve a sequence of abstract intermediate subgoal stages, each defined by the effects of an action planned at the previous level.
2. Flexible *online planning*. Problems can optionally be divided into partial problems, defined by a contiguous subsequence of subgoal stages. Only the first of which is solved prior to execution, and then extended online.

Conformance refinement only enforces an ordering constraint over the achievement of subgoal stages. This guides search by forcing plans to pass through these stages, restricting the search space. Thus, the speed of refinement planning is attained, without needing to divide problems, or require a hard constraint on the achievement time of subgoals. Problem division is still supported however, and is extended to enable online planning. Our partial problems generalise subproblems by refining a subsequence of actions (i.e. subgoal stages) from the previous level, rather than a single action.

When problems are not divided, the dependency problem is entirely eliminated, by considering the global problem (including all subgoals). This increases search efficiency for very little reduction in plan optimality. When problems are divided, only a local problem is considered, the dependency problem is partly alleviated, by allowing dependent subgoals in the same subsequence to interact. This reduces execution latency at the possible cost of long-term plan optimality.

We also propose our *condensed abstract domain models*. These extend previous models based on state granularity, by additionally relating class types over different abstractions. This generates rules for condensed models automatically and provides a more precise relation with the ground model.

To make our approach easy to understand, we introduce *refinement trees*, a data structure defining plan refinements. A tree's head is a subgoal stage, and children a partial plan achieving its head. A hierarchical plan is a sequence of trees.

3.1. Structure of ASH

The diagram in Figure 3 shows the structure of ASH. Its parts, their functions/dependencies, and control/information flow between them (as indicated by arrows) are as follows;

1. The central red box is the ASH planner itself, containing its four operational modules described overleaf.
2. The left-hand blue boxes represent a planning domain definition, an arbitrary size hierarchy of domain models, with state spaces linked by abstraction mappings.

3. The top purple box are the problem-specific inputs to the planner, the relevant entity constants, ancestry relations over constants, and initial state and goals.
4. The right-hand pink boxes indicate the flow of hierarchical conformance refinement planning, plan at each level in descending order, passing conformance constraints generated at the previous level to the next.
5. The bottom green box is the resulting hierarchical plan, a sequence of refinement trees defining a hierarchy of conforming classical plans over all abstractions.

3.2. Operational Modules of ASH

ASH is built around the Clingo ASP system [21]. Clingo finds the answer sets of an ASP program in two steps, *grounding* and *solving*. In the prior, the ground program is obtained by replacing all non-ground program rules with their ground instances by substituting all variables with constants⁷. In the latter, the ground program is solved to find its answer sets.

Clingo's API [37] allows us to integrate ASP with scripting language Python. The result, is a light-weight and highly portable planner, which combines the expressive power and defeasible reasoning capabilities of ASP with the control of Python. The API gives two particularly powerful features:

1. *Selective grounding via program parts*, which allows modularisation of ASP programs, as described below.
2. *Incremental solving*, which allows planning even when the planning horizon (plan length) is not known [22].

The four operational modules of ASH are Clingo ASP program parts, each has a distinct function in the planner:

1. *Instance Relations Module IM*: Deals with creating the instance relation constraints, containing type constraints and ancestry constraints, which control what entity constants replace variables occurring in rules.
2. *State Module SM*: Deals with ensuring that the state representation is complete and valid at all time steps.
3. *Plan Module PM*: Deals with ensuring that planned state transitions are legal and plans achieve all goals.
4. *Conformance Module CM*: Deals with ensuring that plans at adjacent abstraction levels are conforming by generating and enforcing the conformance constraints.

⁷A rule/program with no variables is *ground*, else it is *non-ground*.

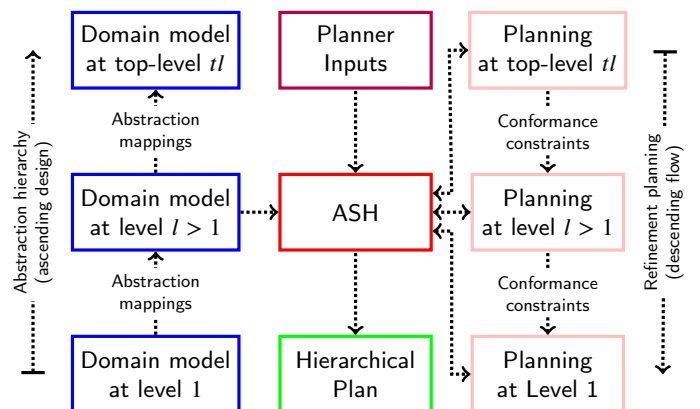


Figure 3: Structure of ASH

4. Blocks World Plus Example Domain

The Blocks World (BW) is a classic problem in robotics [32]. The BW consists of a set of cuboid blocks “scattered” on a table, that must be stacked into towers. Logistics is also a classic problem in robotics [34]. This requires finding the easiest path between locations whilst transporting cargo.

To demonstrate ASH, we use a combined BW and logistics domain, called the Blocks World Plus (BWP) domain. Complexity of the BW is increased by adding numbers and colours to the blocks and requiring the blocks to be stacked in descending order towers of unique colour. Complexity of logistics is increased by adding doors between certain rooms and requiring the robot to have a free hand to open them.

Figure 4c shows the static structure of the BWP domain. It consists 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 initially 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 in Figure 4a and two red blocks start in cell 0 of the store room⁸. The objective is to stack the blocks into the configuration in Figure 4b. These configurations present the classic reasoning problem known as the “Sussman anomaly” (see Page 398 [47]).

Talos, a mobile robot, starts in cell 0 of the starting room. Talos is an anthropomorphic robot, who has two prehensile manipulator arms. Each arm 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 its hand, with it. However, Talos cannot move with an arm extended.

The BWP domain presents many complex reasoning challenges, often ignored in past works, e.g. needing a free hand to open doors. In particular, the domain is designed to promote both the ignorance and independence problems, allowing us to review their effects on conformance refinement.

⁸Whilst the initial state is usually problem specific, rather than domain specific, we have fixed certain aspects in this paper for simplicity.

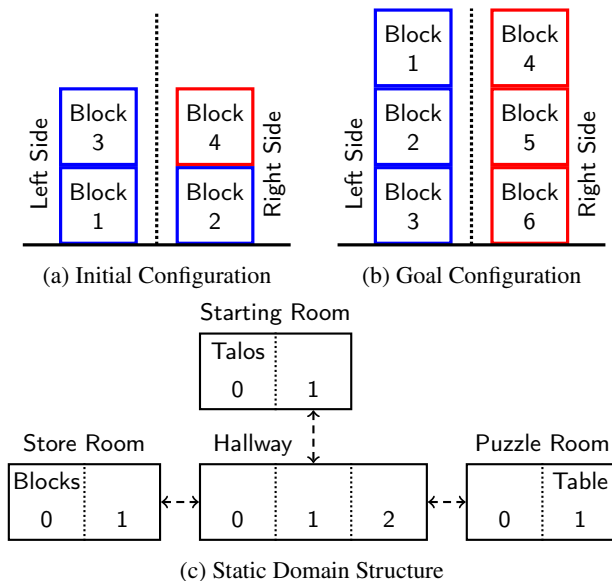


Figure 4: BWP Example Domain

5. Planning Domain Models

A planning domain model defines the dynamic and static laws of a planning domain at a distinct level of abstraction. These laws are encoded as ASP rules, which when grounded, define the valid states and legal transitions of a discrete state transition system. Such models can be used to construct and represent abstraction hierarchies for refinement planning.

When constructing an abstraction hierarchy, the ground model is designed first, and the abstract models are then designed iteratively “from the ground up”, in ascending order. Abstract models are created via various abstractions of another model, called the original. The original model may be the ground model or another abstract model. Abstract models must contain a *state abstraction mapping*, a function between the state spaces of the original model and itself. This mapping propagates the state representation upwards, to allow reasoning about the achievement of subgoal stages, relative to the abstract state, when planning in the original.

In the BWP domain, there are three abstraction levels; the ground model (at level 1), and two abstract models. The first (at level 2) is a condensed model of the ground, and the second (at level 3) is a relaxed model of the condensed.

- *The condensed model* specifies rooms as the union of their constituent cells, the puzzle table as the union of its two constituent sides, and Talos’ manipulator arm assemblies as the union of their parts (limb and hand).
- *The relaxed model* removes all preconditions of locomotive actions, allowing Talos to move instantly to any location, and the precondition requiring Talos to extend his arms or align his hands to grasp blocks.

5.1. The Ground Domain Model

The ground model must be a minimal sufficient description of the planning domain⁹. Plans yielded from the ground model are sent directly to the robot for concrete execution. It must thus be expressive enough to account for the nature of the domain and capabilities of the robot. However, irrelevant detail should always be avoided, to minimise complexity.

5.2. Abstract Domain Models

Abstract models are simplifications of the original, obtained by a removal of problem “details” or by applying simplifying assumptions (e.g. those in [35]). Plans yielded from abstract models only need to be achievable via actions from the original, and are not executed in reality. Instead, abstract plans apply a conformance constraint to refinement planning in the original model, guiding search by restricting the search space and dividing a problem into many partial problems.

Conformance requirement requires that plans generated over an abstraction hierarchy achieve the same effects and remain structurally similar at all levels. The intuition is then, that a plan found in an abstract domain model should have a structure that guides search sufficiently enough, to reduce planning time in the original model (over solving it by classical), by more time than it took to obtain the abstract plan.

⁹The ground model is not concrete. Concrete refers to the continuous time and space models used at run-time by a robot’s low-level controllers.

5.3. Condensed Abstract Domain Models

Condensed models are an intuitive method for creating abstractions. The affect is to reduce (i.e. condense) the state space, by making the state representation less specific, and more granular. The semantics involve abstracting away the detailed constituent *descendants* of entities, combining them into abstract descriptors called *ancestors*, whose class types are the union of those of its descendants. A descendant is one of an ancestor's parts/pieces, or something it is composed of/made from, which give the ancestor certain properties.

The concept is ubiquitous and comes from how humans describe the world at different levels of detail. For example, the BWP expresses that 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 their constituent cells (as done in [51]). In the condensed model, these cells are abstracted away, and Talos only considers locations relative to whole rooms. A condensed navigation plan will thus be a sequence of move actions between such rooms. This plan's refinement will then replace those actions with (sequences of) move actions between the cells of those rooms. Intuitively, such condensed models can significantly reduce the reasoning complexity of planning.

6. Preliminaries

This section introduces the necessary notation.

6.1. ASP Programs

We first need some bespoke notation for ASP programs and their answer sets in the context of the proposed approach.

Definition 6.1 (ASP Program) A Clingo ASP program [20] conforming to the ASP-Core-2 standard [10] is denoted as:

$$\Pi_{j,k}^{pl,tl}(R_0, \dots, R_m) \rightarrow Z_{j,k}^{pl,tl} = \{\zeta_0, \dots, \zeta_n\} : (m, n) \in \mathbb{N}$$

Where R_0, \dots, R_m is a vector of sets of ASP logic rules (the program rules) and the finite set $Z_{j,k}^{pl,tl} = \{\zeta_0, \dots, \zeta_n\}$ are the program's answer sets. The program contains the constants;

- $pl \in \mathbb{N} : 1 \leq pl \leq tl$ the current planning level,
- $sl = \{pl, pl + 1\}$ the state representation levels,
- $tl \in \mathbb{N} : tl \geq 2$ the abstraction hierarchy's top-level,
- $j \in \mathbb{N} : 0 \leq j < k$ the time step to start planning from,
- $k \in \mathbb{N} : k \geq 1$ the current planning horizon.

Atoms of an answer set which have an abstraction level parameter (always the first) will have arguments in the range $l \in [pl, tl]$ and atoms which have a time step parameter (always the last) will have arguments in the range $i \in [j, k]$.

We use this to denote a selective grounding, e.g. the program $\Pi_{j,k}^{pl,tl}(\mathcal{IM}, \mathcal{SM})$ contains the modules \mathcal{IM} and \mathcal{SM} .

The nature of conformance refinement planning requires the state to be simultaneously represented at two adjacent abstraction levels, called the state representation levels.

This is necessary, since the state space at the previous level may not be the same as the current planning level. Thus, to reason about the achievement of conformance constraining subgoal stages, defined by the effects of actions planned at the previous level, requires maintaining the state representation for the previous level, whilst planning at the current.

To achieve this, rules containing special constant sl are expanded disjunctively during program grounding, to obtain a version for the current level pl and previous level $pl + 1$.

6.2. Actions and State

An action is an operation which causes a state transition. A state transition is a state change between successive time steps. A state is a finite set of state literals that hold at a given time step. A state literal is a state variable with an assigned value. A state variable is a function describing a property of the domain, that can be fluent (dynamic and can be changed) or static (fixed and cannot change) in time, e.g. the locations of objects *in* : $object \rightarrow location$ or connectedness of locations *con* : $location_1 \times location_2 \rightarrow bool$, respectively.

- Let $v^l : \bar{\kappa}_o \rightarrow \kappa_o$ be a state variable at abstraction level $l \in \mathbb{N}$, a function whose name is v , domain is a vector of class types $\kappa_i : i \in [0, o)$ and range a class type κ_o .
 - A fluent variable is denoted $f^l : \bar{\kappa}_o \rightarrow \kappa_o$.
 - A static¹⁰ variable is denoted $c^l : \bar{\kappa}_o \rightarrow bool$.
- Let $v_i^l(\bar{t}_o) = v$ and $v_i^l(\bar{t}_o) \neq v$ be positive and negative state literals at discrete time step $i \in \mathbb{N}$, a function literal whose terms are $t_i : i \in [0, o)$ and value is v , representing the temporal assignment of state variables to constants of the class types of their domain and range.
 - The ASP encoding of a fluent literal is an atom:

$$\epsilon(f_i^l(\bar{t}_o) = v) = hld(l, f(\bar{t}_o), v, i)$$

$$\epsilon(f_i^l(\bar{t}_o) \neq v) = not\ hld(l, f(\bar{t}_o), v, i)$$
 - The ASP encoding of a static literal is an atom:

$$\epsilon(c^l(\bar{t}_o) = true) = is(l, c(\bar{t}_o))$$

$$\epsilon(c^l(\bar{t}_o) = false) = not\ is(l, c(\bar{t}_o))$$
- Let $\sigma_i^l = \{f_i^l(\bar{t}_o) = v, \dots, c^l(\bar{t}_o) = true, \dots\}$ be a state, a finite set of positive fluent and static state literals at a distinct abstraction level l and discrete time step i .
 - The ASP encoding is a finite set of atoms:

$$\epsilon(\sigma_i^l) = \{\epsilon(v_i^l(\bar{t}_o) = v) \mid (v_i^l(\bar{t}_o) = v) \in \sigma_i^l\}$$
- Let $\delta_i^l = \langle \sigma_i^l, \sigma_i^{l+1} \rangle$ be a state pair, a tuple¹¹ of states at an adjacent pair of abstraction levels l and $l + 1$.
 - The ASP encoding is a finite set of atoms:

$$\epsilon(\delta_i^l) = \epsilon(\sigma_i^l) \cup \epsilon(\sigma_i^{l+1})$$
- Let $a^l : \bar{\kappa}_o \rightarrow bool$ be an action available to plan (by any robot) at abstraction level l , a Boolean function.

¹⁰For simplicity, static state variables may only have Boolean ranges.

¹¹The notation $e(t)$ is the accessor function for the element e of tuple t .

- Let $\alpha_i^l = \langle r, a_i^l(\bar{t}_o) = \text{true} \rangle$ and $\neg\alpha_i^l = \langle r, a_i^l(\bar{t}_o) = \text{false} \rangle$ be positive and negative action literals¹², representing action planned by robot r to at time step i .
 - The ASP encoding is an atom:

$$\epsilon(\alpha_i^l) = \text{occ}(l, r, a(\bar{t}_o), i)$$

$$\epsilon(\neg\alpha_i^l) = \text{not occ}(l, r, a(\bar{t}_o), i)$$
- Let $e_i^l = \langle r, a_i^l(\bar{t}_o), f_i^l(\bar{t}_o) = v \rangle$ be an action effect, representing that if robot r plans action $a_i^l(\bar{t}_o)$ it causes fluent literal $f_i^l(\bar{t}_o) = v$ to hold in the current state σ_i^l .
 - The ASP encoding is an atom:

$$\epsilon(e_i^l) = ef\ f(l, r, a(\bar{t}_o), f(\bar{t}_o), v, i)$$
- Let $\tau_i^l = \langle \delta_i^l, \alpha_i^l, \delta_{i+1}^l \rangle$ be the state transition occurring from the execution of the action α_i^l in the state $\sigma_i^l(\delta_i^l)$.
 - The ASP encoding is a finite set of atoms:

$$\epsilon(\tau_i^l) = \epsilon(\delta_i^l) \cup \epsilon(a_{i+1}^l) \cup \epsilon(\delta_{i+1}^l)$$
- Let $\pi_{j,k}^l = \{\tau_i^l \mid i \in [j, k]\}$ be a monolevel plan, a finite sequence of state transitions over steps $j \leq i < k$.
 - The ASP encoding is a finite set of atoms:

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

6.3. Final Goals and Subgoals

A final goal is a logical conjunction of fluent state literals to be satisfied on the terminal state of a plan. A subgoal stage is a “partial” state, equivalent to a final goal containing only positive fluent state literals, marked with a *sequence index*.

- Let $g^l(\bar{t}) = v$ and $g^l(\bar{t}) \neq v$ be positive and negative goal literals respectively (state literals with no time).
 - The ASP encoding is an atom:

$$\epsilon(g^l(\bar{t}) = v) = fgo(l, g(\bar{t}), v, \text{true})$$

$$\epsilon(g^l(\bar{t}) \neq v) = fgo(l, g(\bar{t}), v, \text{false})$$
- Let $\omega^l = \{g^l(\bar{t}) = v, \dots, g^l(\bar{t}) \neq v, \dots\}$ be the final goal, a finite set of positive and negative goal literals.
 - The ASP encoding of ω^l is a finite set of atoms:

$$\epsilon(\omega^l) = \{\epsilon(g^l(\bar{t}) = v) \mid (g^l(\bar{t}) = v) \in \omega^l\} \cup \{\epsilon(g^l(\bar{t}) \neq v) \mid (g^l(\bar{t}) \neq v) \in \omega^l\}$$
- Let $\rho_i^{l+1} = \{f_i^l(\bar{t}) = v, \dots\}$ be a subgoal stage at sequence index i , a finite set of positive fluent literals.
 - The ASP encoding is a finite set of atoms:

$$\epsilon(\rho_i^{l+1}) = \{sgo(l, f(\bar{t}), v, i) \mid (f_i^l(\bar{t}) = v) \in \rho_i^{l+1}\}$$
- Let $\rho_{j,k}^{l+1} = \{\rho_i^{l+1} \mid i \in [j, k]\}$ be a finite sequence of subgoal stages between sequence indices $j \leq i < k$.
 - The ASP encoding is a finite set of atoms:

$$\epsilon(\rho_{j,k}^{l+1}) = \bigcup_{i=j}^k \epsilon(\rho_i^{l+1})$$

¹²ASH can represent multiple heterogeneous robot problems, but we do not explore such problems in this paper, due to time and space constraints.

7. Hierarchical Planning Domain Definitions

A hierarchical planning domain definition contains non-ground domain-dependant ASP rules given over an abstraction hierarchy. It is a descriptive knowledge base, allowing reasoning about the dynamic system that governs a given domain. This knowledge must be provided by a human expert.

Definition 7.1 (Hierarchical Planning Domain Definition) A hierarchical planning domain definition over the range of levels¹³ $l \in ([1, tl] \cup \mathbb{N}) : tl \in \mathbb{N}$ is defined by the tuple¹⁴:

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

Where each element represents a hierarchy of sets of rules. The *class inheritance hierarchy* $H^{1,tl}$ is a typing system, the *domain sorts* $S^{1,tl}$ declare the actions and state variables, the *domain rules* $R^{1,tl}$ define the domain’s dynamic and static system laws at each level, and the *state abstraction mappings* $M^{1,tl}$ map the state space between adjacent pairs of levels.

The abstraction hierarchy may be arbitrarily large. Each level is a unique domain model, defined formally as follows.

Definition 7.2 (Domain Model) A domain model specifies the dynamics of a domain at a distinct abstraction level:

$$DM^l = \begin{cases} \langle H^{1,tl}, S^l, R^l, \emptyset \rangle & l = 1 \\ \langle H^{l,tl}, S^{l-1,l}, R^l, M^{l-1} \rangle & l > 1 \end{cases}$$

If $l = 1$ then the model is *ground*, otherwise it is *abstract*. Abstract models include the state abstraction mapping M^{l-1} between the previous adjacent model DM^{l-1} and itself.

The following subsections define the semantics of each element of a domain and syntax of their particular rules. For simplicity, we define each element for a given planning level pl . The hierarchy is then set of those for all $pl \in [1, tl]$.

7.1. Class Inheritance Hierarchy

The class inheritance hierarchy classifies the entities that exist in a domain, and constrains class types over state variables and actions. It is unusual as the inheritance hierarchy is declared over the abstraction hierarchy. Further, it intrinsically supports condensed abstract domain models, obtained by specifying the inheritance of *ancestor classes*, from the union of the *override classes*, of its *descendant classes*.

The class hierarchy is used by the \mathcal{IM} to create *instance relations*. These are special predicates used to constrain what entity constants replace variables in rules. They are critical, as they define the domain and range of functions representing actions and state variables. An instance relation is either;

- a type constraint $inst(l, \kappa, \psi)$ which says that entity ψ is an instance of the class κ at abstraction level l ,
- or a ancestry constraint $desc(l, \psi_1, \psi_2)$ which says that entity ψ_2 is a descendant of ψ_1 at abstraction level l .

¹³The notation $l \in [1, tl]$ henceforth denotes $l \in ([1, tl] \cup \mathbb{N}) : tl \in \mathbb{N}$ and similarly the notation $l \in [1, tl)$ denotes $l \in ([1, tl) \cup \mathbb{N}) : tl \in \mathbb{N}$.

¹⁴The notation $Q^{x,y}$ henceforth denotes a set $Q^{x,y} = \{Q^z \mid z \in [x, y]\}$.

¹⁵Note that the set of entity constants is problem-specific, rather than domain-specific. As such, they are not included in the domain definition.

Definition 7.3 (Class Hierarchy) The class hierarchy used when planning at level $pl \in [1, tl]$ is defined by the tuple:

$$H^{pl} = \langle K^{pl,tl}, I, O \rangle$$

Where each element is a finite set, $(K^l = \{\kappa^l, \dots\}) \in K^{pl,tl}$ are class types with name κ declared at level $l \in [1, tl]$, $(\iota = \langle \kappa_1, \kappa_2 \rangle) \in I$ are inheritance relations indicating that κ_1 is a super-class of κ_2 , and $(o = \langle \kappa_1, \kappa_2, \kappa_3 \rangle) \in O$ are override relations indicating that entities of class κ_1 are ancestors, that have descendants of class κ_2 , from which they derive an inheritance to class κ_3 . These are encoded by facts:

Class type: $\epsilon(\kappa^l) = cls(l, \kappa)$

Inheritance: $\epsilon(\iota) = spr(\kappa_1, \kappa_2)$

Override: $\epsilon(o) = ovr(\kappa_1, \kappa_2, \kappa_3)$

The class hierarchy represents a directed acyclic graph. The nodes of the graph are classes and its directed vertical arcs define the downwards propagation of classes across the abstraction hierarchy. Such that, a class declared as κ^l is added automatically at every lower level $\forall m \in [pl, l]$. $\kappa^m \in K^m$ and is deleted at every higher level $\forall n \in (l, tl]$. $\kappa^n \notin K^n$. Its directed horizontal arcs are inheritance relations between classes at the same level. Its undirected vertical arcs are override relations between inheritance arcs at different levels.

Part of the class hierarchy graph for the BWP is given in Figure 5, as declared by the statements in Listing 1. This describes the composition of Talos' manipulator arms as an assembly of an extensible limb and a hand which can grasp objects. 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. Level 1 is the ground model, the classes from the previous level are propagated down, and two new descendant classes *limb* and *hand* are introduced. The dotted vertical arcs indicate the overrides $\langle arm, limb, extensible \rangle$ and $\langle arm, hand, grasper \rangle$. These break the inheritance relation of the override classes to the ancestor class, transferring them to its descendant classes. *Limb* and *hand* now inherit from *extensible* and *grasper*.

Similar graphs (with only one override arc) describe the composition of rooms as sets of cells $\langle room, cell, location \rangle$ and of the puzzle table as its two sides $\langle table, side, surface \rangle$.

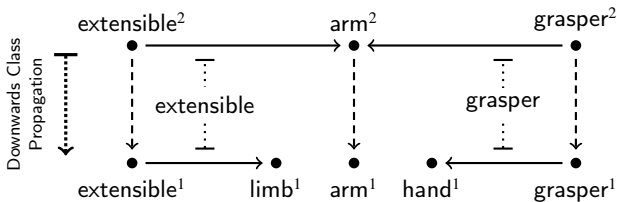


Figure 5: Example Class Hierarchy Graph with Overrides

```
1 %% Arms exist in both the condensed model and the ground model,
2 %% their descendant hands and limbs only exist in the ground model.
3 cls(2, arm). cls(1, limb). cls(1, hand).
4 %% Limbs are extensible, hands are graspers.
5 spr(extensible, limb). spr(grasper, hand).
6 %% Arms inherit extensible and grasper classes from their descendants.
7 ovr(arm, limb, extensible). ovr(arm, hand, grasper).
```

Listing 1: Example Class Hierarchy Declaration with Overrides

7.2. Domain Sorts

The domain sorts declare the actions and state variables used in the domain rules and state abstraction mappings.

Definition 7.4 (Domain Sorts) The domain sorts used when planning at level $pl \in [1, tl]$ is defined by the tuple:

$$S^{pl} = \langle A^{pl}, F^{pl,pl+1}, C^{pl,pl+1} \rangle$$

Where A^{pl} , $F^{pl,pl+1}$ and $C^{pl,pl+1}$ are rule sets specifying *actions*, *fluent* and *static* state variables, respectively. Fluents can be *inertial* or *defined* (as in [24]). Inertial fluents can be affected directly by actions and obey inertia, i.e. their value is preserved in time if left unperturbed. Defined fluents are defined by other fluents. These are declared by rules:

Action: $\epsilon(a^{pl} : \bar{\kappa}_o \rightarrow bool) = act(pl, R, a(\bar{T}_o)) \leftarrow body^{pl}$

Fluent: $\epsilon(f^{sl} : \bar{\kappa}_o \rightarrow \kappa_o) = flu(sl, B, f(\bar{T}_o), Y) \leftarrow body^{sl}$

Static: $\epsilon(c^{sl} : \bar{\kappa}_o \rightarrow bool) = sta(sl, c(\bar{T}_o)) \leftarrow body^{sl}$

Where $B \in \{in, de\}$ define an inertial or defined fluent respectively. All variables occurring in the head of a domain sort declaration must occur in a type constraint of its body, such that $body = inst(l, \kappa_i, T_i), \dots, inst(l, \kappa_o, Y)$, this defines the domain (and range for fluent state variables) of the sort. The predicate in the head of a sort is a *sort constraint*.

An example sorts declaration for the BWP is given in Listing 2. This declares sorts related to the existence of locations which objects can occupy and that a robot can move between such locations. Consider line 5, which declares the inertial fluent $in : object \rightarrow location$. The domain of its single parameter is given by type constraint $inst(sl, object, O)$ and its range by $inst(sl, location, P)$. Since *location* is an override class defined by relation $\langle room, cell, location \rangle$, this fluent is a *condenser state variable*. It condenses the state space by combining many descendant cells (from the ground model) into fewer ancestor rooms (in the condensed model). Our definition ensures that during program grounding, variables are replaced with entity constants of the class corresponding to the current model, to automatically obtain a version of this state variable for the condensed and ground models, from a single declaration. In the condensed model it expands to $in : object \rightarrow room$ and in the ground model to $in : object \rightarrow cell$. This occurs similarly for the other sorts.

This exemplifies the simplicity of condensed model definitions. The only additional knowledge we now need is; the domain dependant mapping from the original to the condensed versions of these sorts, and the problem specific entity ancestry relations between the override type entities.

```
1 #program domain_sorts(pl, sl, tl).
2 %% Robots can move between locations
3 act(pl, R, move(P)) :- inst(pl, robot, R),
4                         inst(pl, location, P).
5 %% Objects must occupy a location
6 flu(sl, in, in(O), P) :- inst(sl, object, O),
7                         inst(sl, location, P).
8 %% Different locations can be connected
9 sta(sl, con(P1, P2)) :- inst(pl, location, P1),
10                        inst(pl, location, P2), P1 != P2.
```

Listing 2: BWP Example Domain Sorts

7.3. Domain Rules

The domain rules encode the axiomatic laws of the dynamic system governing each domain model in the hierarchy. These laws do not tell a robot what to do or when to do it, but rather what it can do. A robot can then reason for itself about how to transition such a system and make a plan.

Definition 7.5 (Domain Rules) The domain rules used when planning at level $pl \in [1, tl]$ are defined by the tuple:

$$R^{pl} = \langle E^{pl}, P^{pl}, U^{pl, pl+1} \rangle$$

Where E^{pl} are *effects* specifying direct deterministic Markovian instantaneous effects of actions¹⁶, P^{pl} are *preconditions* (preconds) specifying conditions that prohibit planning actions, and U^{pl} are *relations* specifying dependencies between state variables. These are declared by the rules:

$$\begin{aligned} \text{Effect:} \quad & \epsilon(e_k^{pl} = \langle R, a_k^{pl}(\bar{T}_o), f_k^{pl}(\bar{T}_o) = Y \rangle) \leftarrow body_{k-1}^{pl} \\ \text{Precond:} \quad & \epsilon(\neg \alpha_k^{pl} = \langle R, a_k^{pl}(\bar{T}_o) = false \rangle) \leftarrow body_{k-1}^{pl} \\ \text{Relation:} \quad & (not) \epsilon(v_k^{sl}(\bar{T}_o) = Y) \leftarrow body_k^{sl} \end{aligned}$$

Where (*not*) is an optional *naf*. The bodies of all rules are a conjunction of state literal encodings, sort constraints, and instance relations. The body of an effect or precondition contains fluents at step $k - 1$, and relations at k . All variables occurring in the head or body of a domain rule must also occur in a sort constraint of its body. Effects and preconds are included for the planning level pl , whereas relations are kept for both state representation levels sl to ensure an invalid state relative to previous level $pl + 1$ is not reached.

Effect rules define state transitions, by deductively inferring how actions affect the current state, conditional on the previous state. An effect is applied to the given action if it is planned from a state which satisfies its body. Preconds prohibit an action from being planned from a state which satisfies its body. Relations force a particular positive or negative fluent literal to hold in a state that satisfies its body.

An example domain rules declaration for the BWP is given in Listing 3. These rules define Talos' ability to move between locations, based on one of each type of rule (given in order). As with the domain sorts declaration, during program grounding these rules are expanded automatically, to obtain versions related to rooms in the condensed model and their descendant cells in the ground model. The constant $relax = 3$ denotes the relaxed model level, used to delete the precondition allowing travel between only connected rooms.

```
1 #program domain_rules(pl, sl, tl, k). #const relax = 3.
2 %% When a robot moves its location changes
3 eff(pl, R, move(L), in(R), L, k) :- act(pl, R, move(L)),
4     flu(pl, _, in(R), L).
5 %% A robot cannot move between locations that are not connected
6 not occ(pl, R, move(L1), k) :- hld(pl, in(R), L2, k - 1),
7     not is(pl, connected(L1, L2)),
8     act(pl, R, move(L2)), flu(pl, _, in(R), L2),
9     sta(pl, connected(L1, L2), pl < relax.
10 %% Reversibility of connections between locations
11 is(sl, con(L1, L2)) :- is(sl, con(L2, L1)), st(sl, con(L1, L2)),
12     sta(sl, con(L2, L1)).
```

Listing 3: BWP Example Domain Rules

7.4. State Abstraction Mappings

The state abstraction mappings link the state spaces of each original domain models to its abstraction. During refinement planning, they propagate the state representation in an upwards manner, from the current planning level, to the previous adjacent level. This allows reasoning about the achievement of subgoals, relative to the abstract state space.

Informally, a state or state literal of an original model, conforms with one of the abstract model, if when the prior is mapped to the abstract it is equivalent to the latter. Both fluent and static mappings are needed. Fluent mappings are used to enforce plan conformance. Static mappings ensure that static laws remain invariant over the hierarchy.

Definition 7.6 (State Abstraction Mappings) The state abstraction mappings used when planning at level $pl \in [1, tl]$ are defined by the tuple:

$$M^{pl} = \langle FM^{pl, pl+1}, CM^{pl, pl+1} \rangle$$

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. Both are declared by the rule:

$$\text{Mapping:} \quad \epsilon(v_k^{pl+1}(\bar{T}_o) = Y^{pl+1}) \leftarrow \epsilon(v_k^{pl}(\bar{T}_o) = Y^{pl}), body^{pl}$$

Where the body is a conjunction containing a single state literal encoding (of the given type) and sort constraints at abstraction level pl . Every state variable should have an abstraction mapping, to ensure the mapping is exhaustive.

The nature of these mappings are specific to the type of abstract model being created. If the abstract state space relative to the original is unchanged, as in relaxed models, the mapping is one-to-one, i.e. all state variables map to themselves. Whereas, if the abstract state space is reduced, as in condensed models, the mappings are many-to-one.

For condensed models, such mappings are very simple to encode. Informally, all condenser variables map from their descendants to their ancestors, and all other variables map to themselves. Examples of such mappings for the condenser state variables related to locations are given in Listing 4. The general method to which these are written is as follows.

For all condenser state variables, and for all elements κ_i of their domain $\bar{\kappa}_o$ defined by an override class, add a rule:

$$\epsilon(f_k^{pl+1}(\bar{t}_o) = v) \leftarrow \epsilon(f_k^{pl}(\bar{t}_o) = v), desc(pl, t_i^{pl+1}, t_i^{pl})$$

If it is fluent and its range κ_v is defined by an override add:

$$\epsilon(f_k^{pl+1}(\bar{t}_o) = v) \leftarrow \epsilon(f_k^{pl}(\bar{t}_o) = v), desc(pl, v^{pl+1}, v^{pl})$$

```
1 #program abstraction_mappings(pl, sl, tl, k).
2 %% If an object is in a location that is a descendant of some ancestor
3 %% location, then that object is also located in the ancestor location
4 hld(pl + 1, in(O), L1, k) :- hld(pl, in(O), L2, k), desc(pl, L1, L2),
5     flu(pl + 1, i, in(O), L1),
6     flu(pl, i, in(O), L2).
7 %% If descendant locations are connected, then so are their ancestors
8 is(pl + 1, con(L1, L2)) :- is(pl, con(L3, L4)),
9     desc(pl, L1, L3), desc(pl, L2, L4),
10     sta(pl + 1, con(L1, L2)),
11     sta(pl, con(L3, L4)).
```

Listing 4: BWP Example Abstraction Mappings

8. Hierarchical Planning Problems

Conformance refinement requires planning over the abstraction hierarchy. For each distinct abstraction level, there exists a unique *classical* or *refinement* type *monolevel planning problem*, defined by an adjacent pair of domain models.

Solving a hierarchical planning problem requires iteratively solving the monolevel problems, from top-to-bottom, in descending order. Only at the top-level is a classical problem solved. At every lower level, a conformance refinement problem is solved, refining the plan from the previous level.

A monolevel conformance refinement problem may be *complete* or *partial*. A complete problem entirely solves the given level, achieving all subgoal stages passed from the previous level, and satisfying the final goal. A partial problem is a division of a complete problem which solves it only partly, achieving a contiguous subsequence of subgoal stages, and does not necessarily start in the initial state (of the complete problem) or satisfy the final goal. Multiple contiguous partial plans can then be concatenated to obtain a complete plan.

A hierarchical problem can be solved *offline* or *online*. Offline planning uses one top-to-bottom iteration, all refinement problems are complete and solved prior to execution. Online planning permits refinement problems to be partial, only a partial ground plan is found prior to execution, and is extended by a nested left-to-right iteration during execution.

To define a planning problem, we need the following;

1. A *hierarchical planning domain definition*, providing the non-ground domain-dependent system rules.
2. *Entity constant* and *ancestry relation* definitions, used to ground these rules and obtain a program representing the problem-specific state transition system.
3. The *initial* state and *final goal* to find a plan between.

Definition 8.1 (Hierarchical Planning Problem) A hierarchical planning problem $HP^{1,tl}$ in domain $DD^{1,tl}$ defined over abstraction levels $l \in [1, tl]$ is defined by the tuple:

$$HP^{1,tl} = \langle DD^{1,tl}, \Psi, \Phi, \zeta^{1,tl}, \Omega^{1,tl} \rangle$$

$$\zeta^{1,tl} = \{ \delta_0^1, \dots, \delta_0^{tl-1}, \sigma_0^{tl} \}$$

$$\Omega^{1,tl} = \{ \omega^1, \dots, \omega^{tl} \}$$

Where Ψ and Φ are the entity constant and ancestry relation definitions respectively, $\zeta^{1,tl}$ initial states¹⁷, and $\Omega^{1,tl}$ goals.

We now need some additional notation for the entity constant definitions and entity ancestry relation definitions.

- Let $\Psi = \{ \text{entity}(\kappa, \psi), \dots \}$ be the entity constant definitions, a finite set of facts defining entities whose names are ψ and whose most base classes are κ .
- Let $\Phi = \{ \text{ances_rel}(\psi_1, \psi_2), \dots \}$ be the ancestry relation definitions, a finite set of facts defining that the entity ψ_1 is an ancestor of entity ψ_2 .

With a planning domain definition already given, defining the remaining problem specific elements is very simple.

¹⁷At the top-level tl only a single initial state σ_0^{tl} is needed to solve the only classical planning problem. Whereas, at all lower levels $l \in [1, tl)$ an initial state pair δ_0^l is needed to solve the conformance refinement problems.

```

1 %% The entity constants for Talos and his descendant parts
2 ent(robot, talos). ent(arm, left_arm). ent(arm, right_arm).
3 ent(limb, left_limb). ent(hand, left_hand).
4 ent(limb, right_limb). ent(hand, right_hand).
5 %% Talos has a left and right arm, each composed of a limb and hand
6 ances_rel(talos, left_arm). ances_rel(talos, right_arm).
7 ances_rel(left_arm, left_limb). ances_rel(left_arm, left_hand).
8 ances_rel(right_arm, right_limb). ances_rel(right_arm, right_hand).
    
```

Listing 5: The Problem-specific Definitions for Talos' Arms

```

1 %% Block 5 and 6 start in cell 0 of the store room
2 %% The other four blocks start stacked on the table
3 hld(1, in((block5 ; block6)), (store_room, 0), 0).
4 hld(s1, in((block5 ; block6)), store_room, 0) :- s1 > 1.
5 hld(1, on(block1), table_left, 0). hld(1, on(block2), table_right, 0).
6 hld(s1, on((block1 ; block2)), table, 0) :- s1 > 1.
7 hld(s1, on(block3), block1, 0). hld(s1, on(block4), block2, 0).
8 %% The goal is to stack the blocks in descending order towers of
9 %% unique colour with block 6 on the right and block 3 on the left
10 fgo(1, on(block3), table_left, true).
11 fgo(1, on(block6), table_right, true).
12 fgo(pl, on((block5 ; block6)), table, true) :- pl > 1.
13 fgo(pl, comp_tower(C), true, true) :- inst(pl, colour, C).
    
```

Listing 6: Initial state and Final Goal declarations for the BWP

The example encoding in Listing 5, declares the entity constant and ancestry relation definitions for Talos' manipulator arms, and their composition as a limb and hand.

The example encoding in Listing 6 contains a partial encoding of the some critical aspects of the initial state and final goals of the BWP problems. These give the initial and desired goal position of the blocks as depicted in Figures 4b and 4b, where the fluent $\text{comp_tower} : \text{colour} \rightarrow \text{bool}$ set to is true when the blocks are stacked in descending order towers of the corresponding colour. In particular, the initial state and final goal must be given at all levels in the abstraction hierarchy, and should conform with each other such that the monolevel plan at each level starts and ends in a conforming state¹⁸. E.g. lines 3 and 4 define the start position of blocks 5 and 6 in the ground and condensed models respectively.

8.1. Monolevel Planning Problems

We first need to define the nature of the various types of monolevel problems that constitute hierarchical problems.

Definition 8.2 (Complete Monolevel Planning Problem) The complete classical or conformance refinement monolevel problem at current planning level $pl \in [1, tl]$ is defined by¹⁹:

$$CP^{pl} = \begin{cases} \langle DD^{pl}, \Psi, \Phi, \sigma_0^{pl}, \omega^{pl}, \emptyset \rangle & pl = tl \\ \langle DD^{pl}, \Psi, \Phi, \delta_0^{pl}, \omega^{pl}, \rho_{1,\varpi}^{pl+1} \rangle & pl < tl \end{cases}$$

If $pl = tl$ then it is the top-level classical problem containing no subgoal stages, otherwise it is a conformance refinement problem and additionally includes the subgoal stages from the previous adjacent level $\rho_{1,\varpi}^{pl+1}$. Where ϖ is the length of the complete monolevel plan from the previous level, (denoted ϖ^{pl+1}), defining the number of subgoal stages to be achieved to obtain a complete refinement plan at the current.

¹⁸We leave the problem of generating an initial state and final goal that is guaranteed to conform across the abstraction hierarchy to our future work.

¹⁹Elements of a monolevel problem are derived from the hierarchical.

Complete refinement problems can be divided into a finite sequence of partial problems, where each partial problem is involved in a single increment of online planning.

Definition 8.3 (Partial Monolevel Planning Problem) The partial κ^{th} in sequence conformance refinement monolevel problem at current planning level $pl \in [1, tl]$ is defined by:

$$PP_j^{pl}(\kappa) = \begin{cases} \langle DD^{pl}, \Psi, \Phi, \delta_j^{pl}, \omega^{pl}, \rho_{\varphi, \vartheta}^{pl+1} \rangle & \vartheta = \varpi^{pl+1} \\ \langle DD^{pl}, \Psi, \Phi, \delta_j^{pl}, \emptyset, \rho_{\varphi, \vartheta}^{pl+1} \rangle & \vartheta < \varpi^{pl+1} \end{cases}$$

Where κ is called the *division number* of the partial problem, δ_j^{pl} is the state pair to start planning from and $\rho_{\varphi, \vartheta}^{pl+1} \in \rho_{1, \varpi}^{pl+1}$ is the contiguous subsequence of subgoal stages achieved by the problem. If start step $j = 0$ then both $(\kappa, \varphi) = 1$ and the problem is *initial*, i.e. it is the first partial problem, and thus includes the first subgoal stage. Otherwise, if $\kappa > 1$ then $(\kappa, \varphi) \in (1, \varpi^{pl+1}]$ and it is an extension of the previous partial problem at $\kappa - 1$. If $\vartheta = \varpi^{pl+1}$ the problem is *terminal*, i.e. it is the last partial problem, and thus includes both the last subgoal stage and final goal (completing level pl).

The starting state of an initial partial problem is the initial state of the respective complete problem. Otherwise, the starting state is the terminal state of the partial plan which solved the previous problem. Importantly, the previous problem must be solved to know this state. It is only guaranteed to satisfy the final subgoal stage of the previous problem.

If a complete problem CP^{pl} is divided into $n \leq \varpi^{pl+1}$ partial problems, we obtain a sequence of partial problems denoted by $div(CP^{pl}, n) = \{PP_0^{pl}(1), PP_1^{pl}(\kappa), \dots, PP_j^{pl}(n)\}$. The range of their subgoal stages is a function of division number $(\varphi, \vartheta)^{\mathbb{N}} : \mathbb{N} \rightarrow (\mathbb{N}, \mathbb{N})$ giving $\kappa^l \mapsto (\varphi, \vartheta)(PP_j^{pl}(\kappa))$. Where the subgoal stages of each follow contiguously such that $\varphi^{pl}(1) = 1$, $\vartheta^{pl}(x) = \varphi^{pl}(x + 1)$ and $\vartheta(n) = \varpi^{pl+1}$.

8.2. Problem Solutions

Solutions to planning problems are found by solving ASP programs (Section 6.1) containing the modules of ASH (Section 3.2). The modules contain the domain-independent rules used for finding plans, these are detailed in Appendix B.

Definition 8.4 (Solving Planning Problems) Solving any monolevel planning problem $MP_j^{pl} \in \{CP^{pl}, PP_j^{pl}(\kappa)\}$ (i.e. complete or partial) requires solving the ASP program:

$$\begin{aligned} \Pi_{j=0, k}^{pl, tl}(MP_{j=0}^{pl}, IR, SM, PM) &\rightarrow Z_{j=0, k}^{pl, tl} & pl = tl \\ \Pi_{j, k}^{pl, tl}(MP_j^{pl}, IR, SM, PM, CM) &\rightarrow Z_{j, k}^{pl, tl} & pl < tl \end{aligned}$$

Where conformance module CM is included for refinement only. The answer sets $Z_{j, k}^{pl, tl}$ contain all plans $\pi_{j, k}^l$ such that:

$$\Xi(MP_j^{pl}, k) = \{\pi_{0, k}^{pl} \mid \epsilon(\pi_{0, k}^{pl}) \in \zeta, \zeta \in Z_{0, k}^{pl, tl}\}$$

Where $\Xi(MP^{pl}, k)$ is the problem's k length solution space, such that level pl is *solvable* iff $\exists k \in \mathbb{N}. \Xi(CP^{pl}, k) \neq \emptyset$. The minimum k length $\Xi(CP^{pl}, k)$ is called the *global refinement optimum* of the complete problem CP^{pl} and $\Xi(PP_j^{pl}, k)$ the *local refinement optimum* of the partial problem PP_j^{pl} .

8.2.1. Classical Problem Solutions

We begin by defining solutions for the top level classical problem. This is a monolevel plan, that contains a path between the initial state and a state that satisfies the final goal.

Definition 8.5 (Final Goal Satisfaction) A state σ_k^{pl} satisfies final goal ω^{pl} iff all goal literals are satisfied by the state:

$$\begin{aligned} sat(\omega^{pl}, \sigma_k^{pl}) &\Leftrightarrow \forall (f^{pl}(\bar{t}_o) = v) \in \omega^{pl}. (f_k^{pl}(\bar{t}_o) = v) \in \sigma_k^{pl} \wedge \\ &\quad \forall (f^{pl}(\bar{t}_o) \neq v) \in \omega^{pl}. (f_k^{pl}(\bar{t}_o) = v) \notin \sigma_k^{pl} \end{aligned}$$

A classical monolevel plan $\pi_{0, k}^{pl}$ achieves final goal ω^{pl} iff its final state transition ends in a state that satisfies the goal:

$$ach(\omega^{pl}, \pi_{j, k}^{pl}) \Leftrightarrow sat(\omega^{pl}, \sigma_k^{pl}(\delta_k^{pl}(\tau_{k-1}^{pl} \in \pi_{j, k}^{pl})))$$

Definition 8.6 (Complete Classical Plan) A monolevel plan $\pi_{0, k}^{tl}$ completes a classical planning problem CP_0^{tl} iff it starts in the problem's initial state and achieves its final goal:

$$\begin{aligned} comp(CP^{tl}, \pi_{0, k}^{tl}) &\Leftrightarrow ach(\omega^{tl} \in \Omega^{1, tl}(CP^{tl}), \pi_{0, k}^{tl}) \wedge \\ &\quad \sigma_0^{tl}(\delta_0^{tl}(\tau_0^{tl} \in \pi_{0, k}^{tl})) \in \zeta^{1, tl}(CP^{tl}) \end{aligned}$$

8.2.2. Conformance Refinement Problem Solutions

We can now define the solutions for the conformance refinement planning problems. This is a monolevel plan, that is both classical complete, and contains a path which passes through a sequence of subgoal stages in the correct order.

The diagram in Figure 6 depicts the conformance refinement process. The blue squares are states, red circles actions, and green diamonds subgoal stages. The upwards dotted arrows indicate states and subgoal stages which conform with (map to) each other. The downwards dashed arrows indicate the creation of subgoal stages from abstract state transitions.

The top-level is a plan generated in an abstract domain model and the bottom-level is the refinement of that plan generated in the original. Each abstract state transition creates one subgoal stage, forming the conformance constraint for the next level. The plan at the next level must both satisfy any reintroduced planning constraints of the original model that were removed in the abstract, and achieve these subgoal stages by passing through states that satisfy them in the same order, thus satisfying the conformance constraint. This restricts the search space, by allowing only original level plans that conform with the abstract, thus making it easier to find a plan. Each subgoal stage can be seen as being achieved by a distinct partial plan. The start state of each non-initial partial plan is known only to satisfy the previous subgoal stage, and is otherwise unknown until that stage has been achieved by the previous partial plan. The initial state and final goal at each level is fixed and must conform with each other.

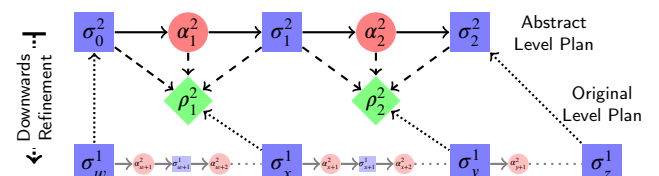


Figure 6: Conformance Refinement Planning Process

Conformance constraints are effective yet flexible way to guide search. Since only an ordering constraint is applied over the time of achievement of subgoal stages, mutually dependent subgoal stages included in the same problem are allowed to *interact*. This property allows the planner to delay the achievement of earlier subgoal stages to better account (in the satisfying state) for things it needs to achieve later subgoal stages, if doing so would decrease the plan length. This is critical, as it eliminates the dependency problem in complete planning and alleviates it in partial planning. Unfortunately, this is why the start state of a non-initial partial problems cannot be fixed, else this flexibility is lost.

Complete refinement planning gains its speed only from the restriction on the search space provided by conformance constraints, but eliminates the dependency problem by considering the global problem, inclusive of all subgoals, all of which can interact. Solutions to complete refinement problems do achieve the global refinement optimum, but are not guaranteed to be optimal relative to the respective classical problem. This is because the requirement for the plan to pass through the subgoal stages may lead the planner away from the classical optimum, i.e. the need for all decisions made at the abstract levels to be maintained during the refinement.

Partial planning gains additional speed by reducing plan length linearly, and thus search complexity exponentially, but the dependency problem can be promoted as only the local problem is considered, exclusive of some subgoals, and only mutually dependent subgoals in the same subsequence can interact. Solutions to partial refinement problems are not guaranteed to be globally optimal relative to the respective complete refinement problem, because the requirement to divide a problem based on a subsequence of subgoal stages, means that the planner does not consider any later subgoal stages, thus the planner falls into local optimums that are good in the short term but bad in the long term (respective of the global problem inclusive of all subgoal stages).

Definition 8.7 (Conformance Constraint Generation) The conformance constraints generated from plan $\pi_{j,k}^{pl+1}$ are a sequence of subgoal stages, one for each of its state transitions:

$$\rho(\pi_{j,k}^{pl+1}) = \{\rho_{i+1}^{pl+1}(\tau_i^{pl+1}) \mid \tau_i^{pl+1} \in \pi_{j,k}^{pl+1}\}$$

These are generated as follows. Let X be an ground action effect domain rule (see Definition 7.5 and Equation 1), then:

- Let $hd(X) = e_k^{pl+1} = \langle \alpha_i^{pl}, f_i^{pl}(\bar{t}_o) = v \rangle$ be its head.
- Let $ls^+(X) = \{v^{pl}(\bar{t}) = v, \dots\}$ and $ls^-(X) = \{v^{pl}(\bar{t}) \neq v, \dots\}$ be the positive and negative body state literals.

Its head is satisfied in a state containing all its state literals:

$$sat(X, \sigma_i^{pl}) \Leftrightarrow ls^+(X) \subseteq \epsilon(\sigma_i^{pl}) \wedge ls^-(X) \cup \epsilon(\sigma_i^{pl}) = \emptyset$$

The subgoal stage $\rho_{i+1}^{pl+1}(\tau_i^{pl+1})$ is that generated from the effects of the state transition's action $\alpha_{i+1}^{pl+1}(\tau_i^{pl+1})$ defined as:

$$\begin{aligned} \rho_{i+1}^{pl+1}(\tau_i^{pl+1}) &= \{f_{i+1}^{pl+1}(\bar{t}_o) = v \mid X \in E^{pl+1}(DD^{pl+1}), \\ &\quad sat(X, \sigma_i^{pl+1}(\tau_i^{pl+1}))\} \\ &: hd(X) = e_{i+1}^{pl+1}, \alpha_{i+1}^{pl+1}(\tau_i^{pl+1}) = \alpha_{i+1}^{pl+1}(e_{i+1}^{pl+1}) \end{aligned}$$

Definition 8.8 (Subgoal Stage Satisfaction) A subgoal stage ρ_i^{pl+1} is satisfied by a state σ_k^{pl} iff all subgoals are in the state:

$$sat(\rho_i^{pl+1}, \sigma_k^{pl}) \Leftrightarrow \rho_i^{pl+1} \subseteq \sigma_k^{pl}$$

A monolevel plan $\pi_{j,k}^{pl}$ achieves subgoal stage ρ_i^{pl+1} iff its final state transition ends in a state that satisfies the stage:

$$ach(\rho_i^{pl+1}, \pi_{j,k}^{pl}) \Leftrightarrow sat(\rho_i^{pl+1}, \sigma_{k+1}^{pl+1}(\delta_{k+1}^l(\tau_k^{pl} \in \pi_{j,k}^{pl})))$$

If $\rho_i^{pl+1} = \rho(\tau_i^{pl+1})$ then $ach(\rho(\tau_i^{pl+1}), \pi_{j,k}^{pl})$ says that the conformance refinement of the state transition from the previous level τ_i^{pl+1} is the partial plan $\pi_{j,k}^{pl}$ at the current. We say that $\pi_{j,k}^{pl}$ *uniquely achieves* subgoal stage ρ_i^{pl+1} iff it contains within it no other partial plan that achieves the stage:

$$\begin{aligned} uach(\rho_i^{pl+1}, \pi_{j,k}^{pl}) &\Leftrightarrow ach(\rho(\tau_i^{pl+1}), \pi_{j,k}^{pl}) \wedge \\ &\quad \nexists \pi_{j,x}^{pl} \in \pi_{j,k}^{pl}. ach(\rho(\tau_i^{pl+1}), \pi_{j,x}^{pl}) \end{aligned}$$

Unique achievement ensures that only the terminal transition of the plan achieves the subgoal stage.

Definition 8.9 (Conforming Plan) The (possibly partial) plan $\pi_{y,z}^{pl}$ is a conformance refinement of subgoal stages $\rho_{w,x}^{pl+1}$ iff the plan uniquely achieves all subgoal stages in order:

$$\begin{aligned} con(\rho_{w,x}^{pl+1}, \pi_{y,z}^{pl}) &\Leftrightarrow \forall \rho_h^{pl+1} \in \rho_{w,x-1}^{pl+1}. \exists! (\pi_{i,j}^{pl}, \pi_{j,k}^{pl}) \subseteq \pi_{y,z}^{pl} \\ &: uach(\rho_h^{pl+1}, \pi_{i,j}^{pl}) \wedge uach(\rho_{h+1}^{pl+1}, \pi_{j,k}^{pl}) \end{aligned}$$

This ensures that, for all subgoal stages at indices $h \in [w, x-1]$, there is exactly one pair of partial plans $(\pi_{i,j}^{pl}, \pi_{j,k}^{pl}) \subseteq \pi_{y,z}^{pl}$, where the prior partial plan uniquely achieves subgoal stage ρ_x^{pl+1} before the latter uniquely achieves next stage ρ_{x+1}^{pl+1} .

Definition 8.10 (Complete Conformance Refinement Plan)

A monolevel plan $\pi_{0,\varpi}^{pl}$ completes a conformance refinement planning problem CP^{pl} iff it is a complete classical plan and is a conformance refinement of subgoal stages $\rho_{0,\varpi}^{pl+1}(CP^{pl})$:

$$crc(CP^{pl}, \pi_{0,\varpi}^{pl}) \Leftrightarrow clc(CP^{pl}, \pi_{0,\varpi}^{pl}) \wedge con(\rho_{0,\varpi}^{pl+1}, \pi_{0,\varpi}^{pl})$$

This also holds if $\pi_{0,\varpi}^{pl}$ is the concatenation (union) of a contiguous sequence of partial plans $\{\pi_{0,i}^{pl}, \pi_{i,j}^{pl}, \dots, \pi_{k,\varpi}^{pl}\}$ which solve the partial problem sequence $div(CP^{pl}, n)$ such that each partial plan conforms with its subgoal subsequence:

$$\forall x \in [1, n]. conf(\rho_{\varphi(x), \theta(x)}^{pl+1}, \pi_{j,k}^{pl} \in \pi_{0,\varpi}^{pl})$$

If the partial plan that achieves the last subgoal stage $uach(\rho_{\varpi}^{pl+1}, \pi_{j,k}^{pl})$ does not achieve the goal $\neg ach(\omega^{pl}, \pi_{j,k}^{pl})$ (i.e. $k < \varpi^{pl}$), then we call $\pi_{k,\varpi}^{pl}$ the trailing part of $\pi_{0,\varpi}^{pl}$, which does not achieve any stage from the previous level. This occurs when refining from the condensed model to the ground, because the ground state space is more detailed. This means that, achieving the last subgoal stage relative to the move granular condensed state space, might not achieve the ground level final goal, since many ground states may achieve the last subgoal stage but not necessarily the final goal.

8.3. Refinement Trees and Hierarchical Plans

Refinement trees are graphical models, which depict how conformance refinement ensures plans remain structurally similar at different abstraction levels. They map abstract actions, to the sequence of more concrete actions which refine them and show how planning problems can be divided.

Their head is a state transition of an abstract model, defining a subgoal stage, which is achieved by its children, an ordered sequence of refinement trees of the original model, i.e. the children are a refinement of the head. The plan defined by a tree's children achieve the same effects as its head transition, relative to the abstract model's state representation.

Definition 8.11 (Refinement Tree) A refinement tree whose head is at level l and leafs at $b \leq l$ is defined by the tuple:

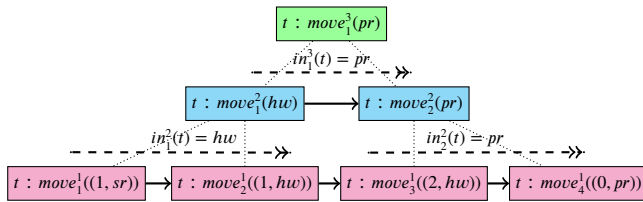
$$\lambda_i^{b,l} = \langle \tau_i^l, \rho_{i+1}^l, \Lambda_{j,k}^{b,l-1} \rangle : j \geq i, k > j$$

Where τ_i^l is the tree's head transition, ρ_{i+1}^l its subgoal stage, and $\Lambda_{j,k}^{b,l-1}$ its children. The children define the partial plan $\pi_{j,k}^{l-1}$ uniquely achieving ρ_{i+1}^l to form a descending sequence of trees for each transition $\{\lambda_h^{b,l-1} \mid \tau_h^{l-1} \in \pi_{j,k}^{l-1}\}$. A tree is:

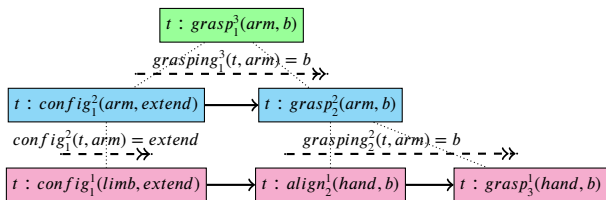
- *empty* if it has an empty set of children,
- *achieved* if its children achieve its subgoal stage,
- *absolute* if either; its head is at the ground level i.e. $l = 1$, or achieved and all its children are absolute.

In other words, an absolute tree is one where itself and all of its children are achieved, all the way down to ground level.

A hierarchical plan is then simply a top-level sequence of refinement trees $\Lambda_{0,\varpi}^{b,tl}$. The hierarchical plan is said to be *final* on the hierarchical planning problem $HP^{1,tl}$ iff all its top-level trees are absolute, i.e. $b = 1$, and is complete at all levels $l \in [1, tl]$. If $\pi_{0,\varpi}^l : l \in [1, tl]$ is complete, but has a trailing part, then that part does not descend from any tree at the previous level $l + 1$. To simplify the notation, the last refinement tree at level l simply “adopts” this trailing plan.



(a) Navigation Refinement Tree



(b) Manipulation Refinement Tree

Figure 7: Absolute Refinement Tree Examples for the BWP

Two example absolute refinement trees for the BWP are given in Figure 7. Where the top-level green nodes are actions planned in the relaxed model, the middle-level blue nodes the condensed model, and the bottom-level pink nodes the ground model. The horizontal dashed arrows are subgoal stages (here just one subgoal) generated from the ascending node (a tree's head). The dotted lines they cross define the descending children (a partial plan) which achieve that stage.

Sub-Figure 7a is a tree for a navigation plan for Talos (denoted by t) moving to puzzle room pr starting from starting room sr . In the relaxed model, Talos can move instantly to any location, and reaches the puzzle room with one action $t : move(pr)$ creating the subgoal $\rho_1^3 = \{in_1^3(t) = pr\}$. In the condensed model, Talos may move only between connected rooms, hence (assuming the puzzle room door is open) he achieves this subgoal with two actions, by moving through the hallway hw , creating two more subgoals. In the ground model, Talos must move between cells of these rooms; firstly reaching a cell of the hallway and then the puzzle room, to achieve the middle-level tree and absolute the top-level tree.

Sub-Figure 7b is a 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 $t : grasp_1^3(arm, b)$ creating $\rho_1^3 = \{grasp_1^3(t, arm) = b\}$. In the condensed model, the arm must be extended prior to grasping the block, requiring an additional action to achieve the top-level subgoal, again creating two more subgoals. 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 with its descendent hand, however this requires one additional alignment action to enable it.

The abstract example hierarchical plan in Figure 8 depicts a sequence of refinement trees. Where the notation $l:i$ is the state transition τ_i^l and the vertical red lines are all possible plan divisions. Consider generating this plan online, where all divisions are made. The contours indicate the increments of online planning, containing a hierarchy of partial plans, one for each division number. Each increment starts from the deepest unachieved refinement tree and refines its children down to the ground level. The intuition is to obtain as much of the ground plan as possible, as quickly as possible. Importantly, not all divisions are balanced, since the lengths of the ground-level partial plans (number of child nodes) that achieves each tree are not equal. Because the length of the partial plan which achieves tree (or a subsequence of trees) is unknown until it is solved, there is no way to direct measure their complexity prior to solving.

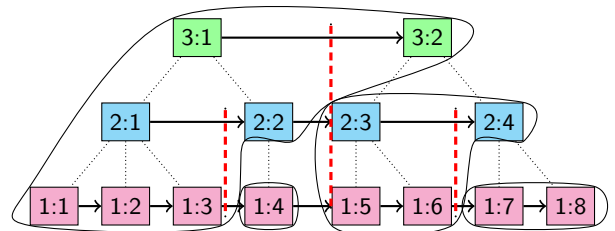


Figure 8: An Abstract Unbalanced Hierarchical Plan

8.4. Finding Effective Problem Divisions

The nature of conformance refinement means that not all possible problem divisions must be made in online planning. The challenge is in deciding how many divisions to make, and where to make them. This is the *problem division strategy*. The desire is to maximise the number of divisions, but to divide only between independent partial problems, since this will minimise the complexity of the overall problem.

A sequence of partial problems are independent *iff* the sum of the local refinement optimums is equal to the global refinement optimum, otherwise they are dependent. To know this requires generating the minimal length solution spaces. Thus, the dependency of partial problems is unknown until solved. It is hard even to predict, since the start state of each partial problem is unknown until the previous is solved.

Since a large number of divisions increases the likelihood of dividing between dependent partial problems, an effective strategy is likely to be one that balances the estimated complexity of the partial problems, against this likelihood. We propose the design of division strategies as an open research question, which we will address in our future work.

9. Planning Algorithms

This section provides the pseudo-code algorithm used to generate hierarchical plans by conformance refinement. It contains inner functions used for; solving monolevel problems, performing the top-to-bottom iteration downwards across the abstraction hierarchy, and the left-to-right iteration over a sequence of partial problems.

The algorithm first initialises and solves the top-level classical planning problem. It then initialises a hierarchical plan, which it finalises in either offline or online mode.

The function *SOLVE* simply solves the corresponding problem incrementally. For all steps $k > j$ it solves the program given by Definition 9, checks if the k length solution space is non-empty, and returns a monolevel plan if it is.

The function *TOP-BOT* performs the top-to-bottom iteration. It takes as input; a hierarchical planning problem, a hierarchical plan extended by the function call, a level ml to plan downwards from, and the division strategy ρ and division number κ . The algorithm iterates in descending order, initialising a complete or partial monolevel plan at each level, with starting state equal to either the complete problems initial state or the terminal state of the currently stored partial plan i.e. $\delta_j^l \in \pi_{0,j}^l(\Delta_{0,\varpi}^{b,tl})$. If the planner is in offline mode, the division strategy is irrelevant, and subgoal stage range simply includes all those from the previous level. If the planner is in online mode, the division strategy is used to decide the range for the current division number based on the existing hierarchical plan. It then calls the monolevel planning function, if some k length plan is returned by the monolevel planning call, the hierarchical plan is extended and returned when the ground level is reached.

The function *LEFT-RIGHT* performs the left-right iteration. It generates hierarchical plans online, taking as input only a hierarchical planning problem, hierarchical plan, and

a division strategy ρ . Whilst the ground level is incomplete, the algorithm iterates upwards in ascending order, searching for the shallowest unachieved subgoal stage, i.e. refinement tree. It then calls the top-to-bottom iteration function, performing the online planning increment for division number κ . The returned hierarchical plan is an extension of that which was passed as input to the call. This implicitly performs the left-to-right iteration, since the initial state of the partial problem is set to the final state of the currently held partial plan.

Algorithm 1 Hierarchical Conformance Refinement Plan

```

1: procedure HIERARCHICAL PLAN( $HP^{1,tl}$ ,  $mode$ ,  $\rho$ )
2:   initialise  $CP^{tl}$  from  $HP^{1,tl}$   $\triangleright$  Classical problem
3:    $\pi_{0,\varpi}^{tl} \leftarrow \text{SOLVE}(CP^{tl})$ 
4:   initialise  $\Delta_{0,\varpi}^{b,tl}$  from  $\pi_{0,\varpi}^{tl}$   $\triangleright$  Top-level trees
5:   if  $mode = \text{"offline"}$  then
6:     return TOP-BOT( $HP^{1,tl}$ ,  $\Delta_{0,\varpi}^{b,tl}$ ,  $tl - 1$ ,  $\rho$ , 0)
7:   else if  $mode = \text{"online"}$  then
8:     return LEFT-RIGHT( $HP^{1,tl}$ ,  $\Delta_{0,\varpi}^{b,tl}$ ,  $\rho$ )
    
```

Sub-algorithm 1 - Solve Monolevel problem

```

9: function SOLVE( $MP_j^{pl}$ )
10:  for all  $k > j$  do  $\triangleright$  Solve incrementally
11:    if  $pl = tl$  then  $\triangleright$  Solve classical
12:       $\Pi_{j=0,k}^{pl,tl}(MP_{j=0}^{pl}, IR, SM, PM) \rightarrow Z_{j=0,k}^{pl,tl}$ 
13:    else  $\triangleright$  Solve refinement
14:       $\Pi_{j,k}^{pl,tl}(MP_j^{pl}, IR, SM, PM, CM) \rightarrow Z_{j,k}^{pl,tl}$ 
15:    if  $Z_{0,k}^{pl,tl} \neq \emptyset$  then
16:      extract  $\Xi(MP_j^{pl}, k)$  from  $Z_{0,k}^{pl,tl}$ 
17:      return some  $\pi_{j,k}^l$  from  $\Xi(MP_j^{pl}, k)$ 
    
```

Sub-algorithm 2 - Top-Bottom Hierarchical Iterate

```

18: function TOP-BOT( $HP^{1,tl}$ ,  $\Delta_{0,\varpi}^{b,tl}$ ,  $ml$ ,  $\rho$ ,  $\kappa$ )
19:  decide  $(\varphi, \vartheta)(\kappa)$  using  $\rho$  and  $\Delta_{0,\varpi}^{b,tl}$ 
20:  for all  $l \in [ml, tl]$  do
21:    initialise  $MP_j^l$  from  $HP^{1,tl}$  using  $(\varphi, \vartheta)(\kappa)$ 
22:     $\pi_{j,k}^l \leftarrow \text{SOLVE}(MP_j^l)$ 
23:    extend  $\Delta_{0,\varpi}^{b,tl}$  with  $\pi_{j,k}^l$ 
24:  return  $\Delta_{0,\varpi}^{b,tl}$ 
    
```

Sub-algorithm 3 - Left-Right Online Iterate

```

25: function LEFT-RIGHT( $HP^{1,tl}$ ,  $\Delta_{0,\varpi}^{b,tl}$ ,  $\rho$ )
26:   $\kappa \leftarrow 1$ 
27:  while  $\neg \text{crc}(CP^1, \pi_{0,\varpi}^1)$  do
28:    for all  $l \in [1, tl]$  do
29:      if Some unachieved subgoal exists at  $l$  then
30:         $\Delta_{0,\varpi}^{b,tl} \leftarrow \text{TOP-BOT}(HP^{1,tl}, \Delta_{0,\varpi}^{b,tl}, l -$ 
31:           $1, \rho, \kappa)$ 
32:         $\kappa \leftarrow \kappa + 1$ 
33:  return  $\Delta_{0,\varpi}^{b,tl}$ 
    
```

10. Experimental Trials

This section presents the results from simulated experimental trials of the proposed conformance refinement planning approach on the BWP domain presented in Section 4.

10.1. Setup

Three different problem instances in the BWP were run. In the first, all doors between rooms start open, the optimal plan length is 39 steps. In the second, the doors between the hallway, and both the store and puzzle rooms, start closed, increasing the optimal plan length (by 14 steps) to 53 steps. The third extends the second, with block 4 starting in cell 0 of the store room (instead of on the puzzle table), increasing the optimal plan length (by 14 steps) to 67 steps. The same entity constants were used throughout all problems to ensure the state space remained unchanged over all experiments.

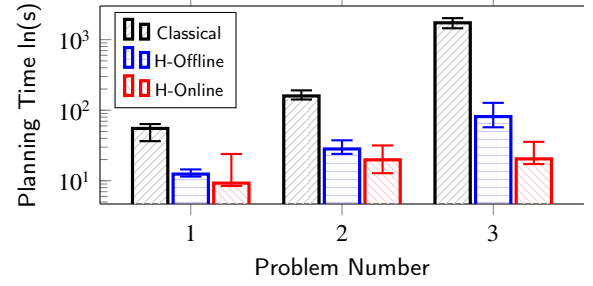
Experiments were run for each of; classical, hierarchical offline (h-offline), and hierarchical online (h-online) planning. All combinations of problems and planning types were run 20 times to ensure a reasonable range for the possible planning times and plan lengths. For h-online, the problem division strategy was set to dynamically divide the problems in half at the top-level (level 3), and again at the middle-level (level 2). Such that the number of subgoal stages passed was equal to 50% of the unachieved subgoals from the previous level. This ensured exactly four ground level (level 1) increments of the online planning algorithm for all problems.

Clingo was invoked with 8 competing threads. All experiments were run on a desktop computer running Microsoft Windows 10 operating system, Intel Core i7-6700K CPU @ 4.00GHz (8 Logical Cores), 16384MB 2633Mhz RAM, and ASUSTeK Z170 Pro Gaming motherboard. Clingo held at least 97% CPU usage throughout all experimental trials.

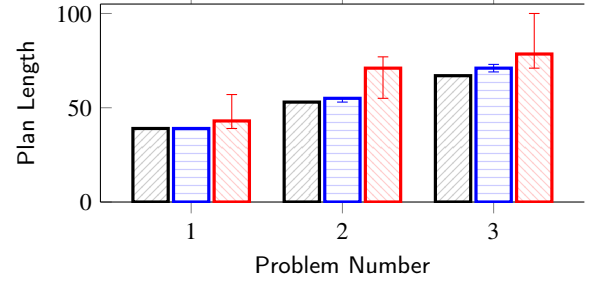
10.2. Results

The primary results for planning times and plan lengths, for all problems and planning types, are given in Figure 9. The bar charts in Sub-Figures 9a and 9b, display median total planning time and plan lengths, respectively. The error bars of these charts indicate the data range around the median, the lower bar indicates the minimum (best case) value and the higher bar indicates the maximum (worst case) value. Table 9c contains the median grounding time (GT), solving time (ST), total planning time (TT) and plan length (PL) for hierarchical planning, as a percentage of the respective value for classical planning. Table 9d breaks down h-online planning time and plan lengths, to each planning increment, similarly as a percentage of classical. This indicates; the time spent obtaining, and how long, the partial plan was, which solved each of the four ground level partial problems. Particularly, the planning time for the first increment is the execution latency, i.e. the time taken to yield the first partial plan.

The grounding time is the time taken to obtain the ground ASP program. This indicates 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. This indicates the complexity (size) of the problem search space.



(a) Total Planning Time for each Problem



(b) Total Plan Length for each Problem

	H-Offline				H-Online			
	GT	ST	TT	PL	GT	ST	TT	PL
P1	187.7	9.9	24.3	100	186.8	9.0	23.0	135.9
P2	164.1	6.7	16.6	103.8	107.8	7.4	13.8	134.0
P3	177.5	4.0	5.8	109.0	76.0	0.5	1.3	120.9

(c) Hierarchical Planning Results as Percentage of Classical

	Inc 1		Inc 2		Inc 3		Inc 4	
	TT	PL	TT	PL	TT	PL	TT	PL
P1	2.4	33.3	2.9	69.2	1.4	82.1	9.7	135.9
P2	1.6	41.5	1.6	81.1	0.6	90.6	6.9	134.0
P3	0.2	37.3	0.2	65.7	0.1	87.3	0.3	120.9

(d) Online Incremental Results as Percentage of Classical

Figure 9: Planning times and plan lengths

Crucially, Charts 9a and 9b, and Tables 9c and 9d, indicate performance of hierarchical conformance refinement relative to classical planning, i.e. reduction in planning time and increase in plan length of hierarchical over classical.

The results for solution checking times respective of plan length for all problems and planning types are given by the graphs in Figure 10. The solution checking time is the time taken to exhaust the ground level search space and determine whether a plan exists, for each step up to the given plan length. It is thus a measure of the complexity of the search space, relative to plan length. For classical and h-offline, exponential trend lines are fitted to the available data. Whereas, for h-online the data is plotted directly, since no clear global exponential trend exists. The vertical dashed green lines indicate the average time step upon which a partial planning increment of h-online completed and moved to the next, i.e. the position where the planning problem has been divided. The vertical dashed orange line indicates the average position where the goal ignorance problem occurred (discussed in Section 10.5). This line indicates the median plan lengths when the goal ignorance problem did and did not occur.

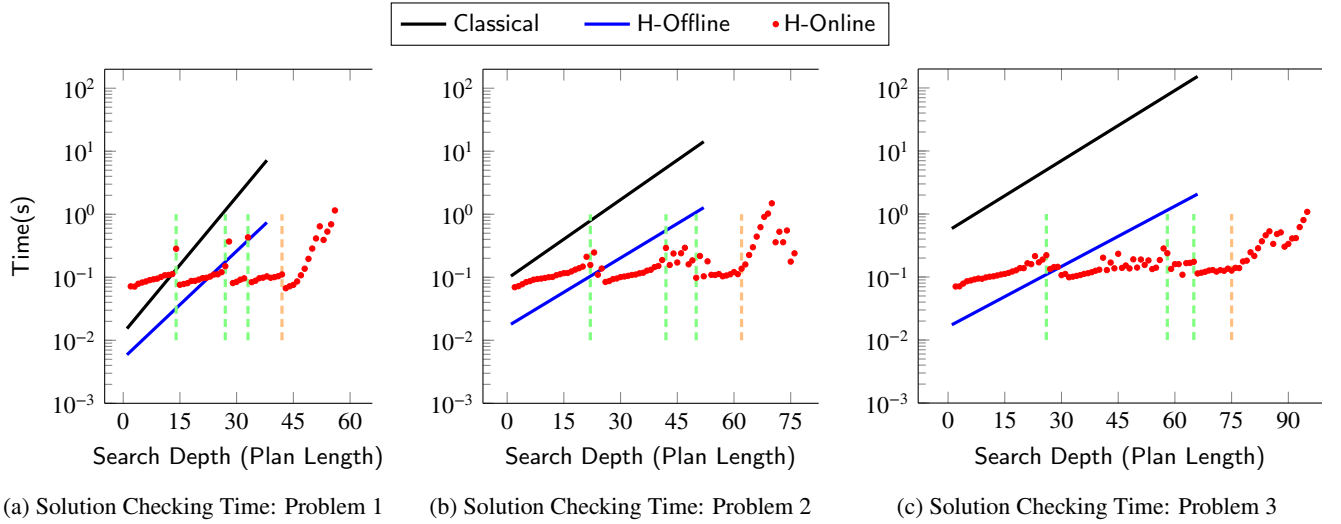


Figure 10: Solution Checking Time respective of Plan Length and Execution Latency

10.3. Evaluation

Our results indicate that hierarchical conformance refinement planning always obtains a ground level plan faster than classical planning. The trade-off is clear, conformance refinement does not always find the classical optimal plan.

Total planning time for any type of hierarchical planning was on average between 24.3% and 1.3% of classical. Whilst hierarchical has greater grounding times than classical (except for h-online P3), solving times for hierarchical were on average between 10.0% and 0.5% of classical, which significantly outweighs the increased grounding time. We expect increased grounding due to the extra conformance constraining rules used in refinement, although the reduced grounding time for h-online P3 is unexpected, and may indicate a link between program and problem complexity. Reduced solving time confirms our expectations that conformance constraints restrict the search space and reduce time taken to exhaust it. Further, execution latency for h-online was on average between 2.4% and 0.2%, of classical. This confirms our expectations that partial planning reduces execution latency.

The solution checking time plots give further support for these claims. The divergent trend lines indicate a consistent exponential reduction in planning time respective of plan length for h-offline over classical. The marks for h-online indicate that problem division appears to linearise planning time respective of length, by “resetting” the complexity of the search space at the point of each division. However, the unequally spaced division lines indicate the hierarchical plan contains unbalanced refinement trees, i.e. the complexity of the partial problems are non-equal (see Section ??).

H-offline shows very little increase and variation in plan lengths for all problems. Whereas, h-online shows far greater increase and variation in plan lengths over h-offline. In the best case, plan lengths for h-online were less than or equal to the average case for h-offline. In the average and worst case, plan lengths for h-online were always greater than h-offline.

As the complexity of the planning problem (minimal plan length) increased, as did the benefits and potential trade-offs of conformance refinement. The most complex P3 shows the

greatest reduction in planning time for h-offline, but the most increase in plan length, over classical. Whereas, the simplest P1 shows the least reduction in planning time with no change in plan length. Conversely, P3 shows the best overall benefits for h-online, with greatest reduction in planning time and execution latency, and least increase in plan length.

10.4. Specific Issues

The following identifies specific issues caused hierarchical conformance refinement to miss the optimal plan length.

H-offline P1 always found the optimal plan length in on average 24.3% of the time taken by classical. This indicates that for simple problems, conformance refinement can find the optimal solution much faster than classical planning.

H-offline P2 finds either the optimal plan or a 3.8% longer sub-optimal plan, in average 16.6% of the time taken by classical. For this problem, the only issue causing non-optimal plans, is due to one decision made in the condensed model. 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 optimal to open the puzzle room door first, and then the store room door, before proceeding to collect the blocks, this is indicated by the red navigation path on Figure 11. 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 thus 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 additional actions to refine to the ground level, as indicated by the longer green path on Figure 11.

H-offline P3 always found plans 3 – 9% longer than the optimal, in on average 5.8% of the time taken by classical. For this problem, the optimal abstract plan could not be refined to the optimal ground plan. This occurred 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, then open the puzzle room door with his spare hand, and place the block on the table. However, as aforementioned, 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). This decision must be maintained in the refinement and means the optimal ground plan can never be found. A further problem occurs if Talos takes only one block first in the relaxed model. When he reaches the puzzle table, it is possible for him to then unnecessarily use his spare arm to arbitrarily move blocks on the table, since grasping preconditions are removed. However, in the ground model, these preconditions are re-imposed on Talos, demanding extra redundant actions be inserted during the refinement to extend and retract this “spare” arm, which would be unnecessary if those blocks were moved later.

H-online experienced the same problems as h-offline, but was also prone to additional problems, occurring from the division of planning problems into partial problems. The most prolific example of this was when a problem division fell between Talos extending his arm and grasping a block. In Figure 11, the green mark indicates the position where this occurred most often. The plan Talos generates in the condensed model involves moving to the store room, extending his arms and then grasping the blocks, this is because Talos can grasp the blocks from anywhere in the store room in the condensed model. In the ground model however, Talos needs one additional action to move from cell 1 to cell 0 of the store room before he can grasp the blocks. If a division is made between the extend and grasp actions from the condensed model, Talos is not able to insert this additional action in the first partial problem. As a result, Talos must retract, move, and then re-extend his arms in the next problem, because he is unable to move with his arms extended.

The most major factor impacting online planning is that the final goal is only considered when solving the final partial problem. This causes Talos to make decisions during early partial problems that are poor with respect to the final goal. This occurred because the blue tower must be stacked on the left and the red on the right of the puzzle table. However, since in the condensed model, the sides of the table are abstracted away, the subgoals from the condensed plan tell the planner nothing about which side to stack the towers on, and are initially chosen arbitrarily. This means that on the final partial problem, whole towers may be moved from one side to the other, increasing plan lengths.

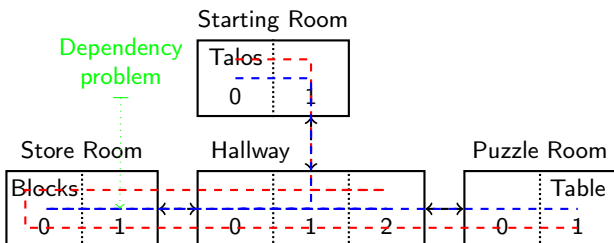


Figure 11: Optimal Navigation Plan for the BWP

10.5. Open Problems in Conformance Refinement

We now identify three general problems for hierarchical conformance refinement planning and pose them as open research questions which we aim to tackle in our future work. The first two of which were identified from existing problem division refinement planners in Section 2.1.2 and have been found to also occur in conformance refinement, but in a somewhat dissimilar manner.

1. **The ignorance problem**, the most intrinsic problem for conformance refinement planning, occurring from the lost knowledge in the abstract models. This leads to ASH making decisions in abstract problems, which are poor with respect to the original ground problem. There exist two causes of this, derived from; the loss and generalisation of action enabling constraints²⁰ in the relaxed model and condensed models respectively.
 - Loss of enabling constraints, can allow a contiguous subsequence of actions that are unconstrained in the abstract model, to be planned in an arbitrary order. If these actions become constrained at the next level, achieving the effects of these actions in the same order, may require inserting extra actions to satisfy in the refinement.
 - Generalisation of enabling constraints, can allow a subset of actions to be planned, that are unconstrained at the abstract model, but constrained in many states of the original that map to them. When such a plan is refined, additional redundant actions may be needed to achieve the effect of the abstract actions from a constrained state.

The open question is whether the properties of abstract models that promote these problems can be identified and minimised, whilst still getting the speed benefits.

2. **The dependency problem**, occurs only in online planning, from the division of problems into partial problems. It occurs because partial plans are only locally optimal. Thus, if the planner does not look sufficiently far enough into the future, and makes decisions that are good in the short term, but poor in the long term. The open question is whether we can determine or predict before hand how to divide the problem such that interaction between partial problems is minimised.
3. **The unconsidered goal problem**, occurs only in online planning, because the final goal is only considered in the final partial problem. It occurs if a condensed plan being refined fails to give the planner sufficient information to account for the more detailed final goal required in the ground model. Thus, the planner may make decisions during early partial problems, that are poor with respect to the final goal, and the planner may then have to undo its previous progress during later partial problems. The open question is whether search can be guided towards making better decisions respective of the goal during early partial problems.

²⁰An action enabling constraint may be a precondition or a state constraint that prohibits an action whose effects lead to an invalid state.

11. Conclusions

We have proposed and experimentally evaluated a novel approach to hierarchical abstraction planning called *conformance refinement*. The method successively refines a highly abstract classical plan, downwards across an abstraction hierarchy, under a constraint requiring that plans achieve the same effects and remain structurally similar at all levels. The approach generalises past refinement planners to support arbitrarily large abstraction hierarchies, containing any abstract model of a planning domain with a state space mapping from the original. Including our ubiquitous condensed models, for which we give a general way to create a model and mapping.

Our implementation, called ASH, supports both offline complete planning and online partial planning. Our results indicate that; offline refinement planning reduces total planning time exponentially over classical planning, and online refinement planning yields partial plans exponentially faster than complete planning. Fundamentally, this allows a robot to begin execution in a fraction of the time taken to generate a complete plan via classical planning. The benefit is made stark in consideration that classical takes on average 1692.6s to solve our most complex test problem, whereas offline refinement takes 98.8s, and online refinement just 21.3s to generate the complete plan and 5.56s to generate the first partial plan, resulting in a 99.68% reduction in execution latency.

ASH gains this speed from two properties of the conformance refinement method to planning; automatic restriction of the low-level search spaces achieved by abstract *conformance constraints* obtained in a general manner from high-level plans, the optional recursive division and sub-division of complete planning problems into a sequence of smaller partial problems. This does away with the need for classic heuristic search, and makes conformance refinement suited to complex problems, where finding admissible heuristics is very difficult. The only extra knowledge required to enable refinement is to define the abstraction hierarchy itself.

The trade-off is that plans are not guaranteed to be optimal in the classical sense. The increased flexibility of conformance refinement solves many issues of past refinement planners. In offline planning, the dependency problem (which was prolific in past work) is eliminated entirely. In online planning, the dependency problem is alleviated by allowing a contiguous subsequence of actions from the previous level to be refined (allowing them to interact), rather than just one action. However, the approach still has important facets, which we have categorised and summarise below:

1. In both offline and online planning, ASH can make decisions which are optimal in the abstract problems, but are poor with respect to the original ground problem. During refinement, these decisions must be maintained by conformance, and might lead to sub-optimality.
2. In online planning, ASH can make decisions during earlier partial problems which are locally optimal, but not globally optimal with respect to the complete problem, and in particular the final goal. This very often leads to substantially longer plans in the long-term.

A. Glossary of Terms

B. Operational Modules of ASH

This Section formally defines the nature of the five operational modules of ASH as introduced in Subsection 3.2.

B.1. Instance Relations Module

The instance relations module contains the rules for creating the instance relations and sort constraints. This provides everything necessary to ground the domain rules and obtain a program representing a state transition system used for planning. In particular, it gives a view on the relevant entity constants, their class types at each abstraction level, and available domain sorts at each abstraction level.

Definition B.1 (Instance Relations and Sort Constraints)

The instance relations and sort constraints are generated from the class hierarchy $H^{1..N}$, the domain sorts $S^{1..N}$, and the problem specific entity constants Ψ and ancestry relations Φ . The program containing the instance relations module, as described below, has exactly one answer set such that:

$$\begin{aligned} \Pi_{0,0}^{1..N}(PP^{1..N}, \mathcal{IR}) \rightarrow Z_{0,0}^{1..N} &= \{ \zeta \} \\ \mathfrak{I}^{1..N}(PP^{1..N}) &= (\{ inst(l, \kappa, \psi) \in \zeta \} \cup \\ &\quad \{ desc(l, \psi_1, \psi_2) \in \zeta \}) \\ \mathfrak{C}^{1..N}(PP^{1..N}) &= \{ act(l, r, a(\bar{t}_o)) \in \zeta \} \cup \\ &\quad \{ flu(l, B, f(\bar{t}_o), v) \in \zeta \} \end{aligned}$$

If ψ occurs in at least one type constraint then ψ exists at level l with the classes $\{ \kappa^l \mid inst(l, \kappa, \psi) \in \mathfrak{I}^{1..N}(PP^{1..N}) \}$.

The rules are the *instance relations module* are given in Listing 7 and function as follows; (6-11) 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 adjacent level which are not overridden by one of their descendants, (12-18) an entity which is an instance of an ancestor class at a given level, loses its inheritance to its override class at the next level, if there is an instance of its descendant class at that next level, (19-21) an entity is the descendant of an ancestor entity at a given level, if there is an ancestry relation between them and they both exist at that level.

```

1 #program instance_module(plan_at, abs_max).
2 inst(A, C, E) :- ent(C, E), cls(A, C), al(A).
3 inst(A3, C1, E) :- inst(A2, C2, E), spr(C1, C2),
4     ent(C3, E), cls(A3, C3), cls(A1, C1),
5     A1 >= A2, al(A1), al(A2), al(A3).
6 inst(A-1, C, E) :- inst(A, C, E), not ovr(A-1, C, E),
7     cls(_, C), al(A), al(A-1).
8 ovr(A-1, C, E1) :- ovr(A-1, C, E1, E2),
9     inst(A, C, E1), inst(A-1, C, E2),
10    cls(_, C), al(A), al(A-1).
11 ovr(A-1, C3, E1, E2) :- ovr(C1, C2, C3), desc(A-1, E1, E2),
12    inst(A, C1, E1), inst(A-1, C2, E2),
13    inst(A, C3, E1), inst(A-1, C3, E2),
14    cls(_, C1), cls(_, C2), cls(_, C3), al(A-1).
15 desc(A, E, D) :- ances_rel(E, D), inst(A, _, E), inst(A, _, D), al(A).
16 desc(A, E, D1) :- desc(A, D2, D1), desc(A, E, D2), inst(A, _, E),
17    inst(A, _, D1), inst(A, _, D2), al(A).
    
```

Listing 7: Rules of the Instance Relations Module

B.2. State and Planning Modules

A state transition system is defined by the ground rules of an adjacent pair of domain models. Recall that when planning at a distinct abstraction level, the state is represented at the current and previous abstraction level. Planning only occurs at the current level, but the state must be propagated to the previous level, in order to reason about the achievement of subgoals, which are state literals from the previous level.

To define such a transition system in ASP, we need to ensure that; its states are valid, and its transitions are legal.

Definition B.2 (Transition System) A transition system is denoted by $\mu^l(DD^{1..N}, \Psi, \Phi) : l \in [1, N]$, is abbreviated to $\mu(PP^l)$ (the system involved in problem PP_j^l whose initial and goal states are ignored), and is defined by the tuple:

$$\mu(PP^l) = \langle \Sigma^l, \Delta^l, T^l, B^l \rangle \quad (2)$$

Where each element is a finite set, $\sigma_i^l \in \Sigma^l$ are the system's valid states, $\delta_i^l \in \Delta^l$ are its valid state pairs, $\tau_i^l \in T^l$ are its legal state transitions, and B^l its executable actions. The set of state pairs Δ^l define the mapping from states of $\Sigma^l(\mu(PP^l))$ to those of $\Sigma^{l+1}(\mu(PP^{l+1}))$.

A state transition system can be viewed as a directed graph, whose nodes are states and arcs are transitions. Planning can be seen as finding a path of its arcs. The example graph in Figure 12 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 rooms²¹. The solid arrows indicate transitions that occur from the execution of a given action in any the four possible states of this system. The dotted connections on the lower states indicate that moving between a and b is not possible when d is closed. Informally, action effects create arcs of the graph, action preconditions delete a subset of those arcs, and fluent constraints, fluent definitions and fluent mappings delete states and all arcs connected to those states.

We now give some formal definitions for states, transitions, and systems.

Definition B.3 (Complete State) A state σ_i^l is complete on a planning problem PP_j^l iff there is exactly one state literal for each state variable relevant to the problem:

$$\forall e(v^l(\bar{t})) \in \mathfrak{C}^{l..N}(DD^{l..N}, \Psi, \Phi). \exists! v_i^l(\bar{t}) = v \in \sigma_i^l : i \geq j$$

²¹ In the BWP, Talos cannot arbitrarily open a door with one action, he must first grasp one of its handles in the same manner that he grasps blocks.

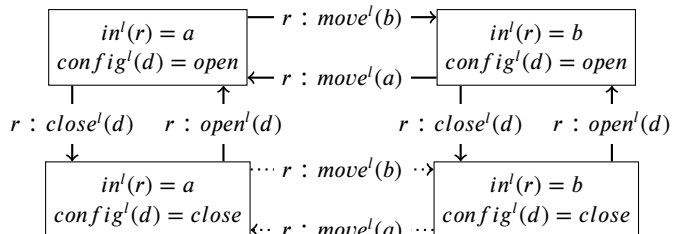


Figure 12: Example State Transition System Graph


```

1 #program state_module(k).
2 { hld(L, F, V, k) :- flu(L, _, F, V), sl(L), k = 0.
3 :- not { hld(L, F, V, k) : flu(L, _, F, V) } = 1,
4   flu(L, _, F, _), sl(L).
5 hld(L, F, V, k) :- hld(L, F, V, k - 1), not not hld(L, F, V, k),
6   f(L, in, F, V), sl(L).
7 hld(L, F, false, k) :- not hld(L, F, true, k), f(L, de, F, _), sl(L).
8 :- not { occ(L, R, A, k) : act(L, R, A) } = 1, pl(L).
9 hld(L, F, V, k) :- occ(L, R, A, k), act(L, R, A),
10   eff(L, R, A, F, V, k),
11   flu(L, in, F, V), pl(L).
    
```

Listing 8: Rules of the State Module

```

1 #program planning_module(j, k).
2 { occ(L, R, A, k) : act(L, R, A) } :- pl(L), k > j.
3 sat(L, F, V, tr, k) :- fgoal(L, F, V, trur), flu(L, _, F, V),
4   hld(L, F, V, k), pl(L).
5 sat(L, F, V, fa, k) :- fgoal(L, F, V, false), flu(L, _, F, V),
6   not hld(L, F, V, k), pl(L).
7 incomplete(k) :- fgoal(L, F, V, B), not sat(L, F, V, B, k),
8   flu(L, _, F, V), pl(L), boolean(B).
9 #external query(k). :- incomplete_plan(k), query(k).
    
```

Listing 9: Rules of the Planning Module

Definition B.4 (Valid State) A valid state is a complete state which satisfies the rules of the planning domain definition. If the program:

$$\Pi_{0,0}^{y,z}(DD^{1,N}, \Psi, \Phi, \text{IM}, \text{SM}, \epsilon(\sigma_0^l)) \rightarrow Z_{0,0}^{y,z}$$

Has an answer set $|Z_{0,0}^{y,z}| = 1$ then σ_0^l is a valid state of $\mu^l(PP^l)$, such that:

$$\epsilon(\delta_0^l) \in \zeta. \sigma_0^l(\delta_0^l) \in \Sigma^l \wedge \delta_0^l \in \Delta^l$$

Definition B.5 (Legal Transition) If the program:

$$\Pi_{0,1}^{y,z}(DD^{1,N}, \Psi, \Phi, \text{IM}, \text{SM}, \text{PM}, \epsilon(\tau_0^l)) \rightarrow Z_{0,1}^{y,z} \quad (3)$$

has an answer set $|Z_{0,1}^{y,z}| = 1$ then τ_0^l is a legal transition of $\mu^l(PP^l)$ such that:

$$\epsilon(\tau_0^l) \in \zeta. \alpha_0^l(\tau_0^l) \in B^l \wedge \tau_0^l \in T^l$$

The state module and plan module contain the domain-independent rules for defining a discrete deterministic state transition system where the state is represented over an adjacent pair of abstraction levels. In a nutshell, the rules first ensure that the initial state is complete, and then ensures that the state remains complete through time as planned actions transition the state sequentially. The plan module also includes the rules defining when goals are satisfied, which are used in the generation of plans through the system, but are not related to the semantics of the transition system itself.

The rules of the *state module* are given in Listing 8 and function as follows; (1) for each time step k , (4) if the state is initial then the *law of awareness* allows fluents to take some value from their range, (2-3) the *law of continuity* ensures that all fluents take exactly one value, (5-7) the *law of inertia* propagates the value of inertial fluents through time unless changed by actions, (8-9) the *closed world assumption for*

defined fluents ensures that if a defined fluent is not deductively true, then it is false. Of particular interest is the law of inertia, a defeasible rule which ensures that if a fluent has not been deductively assigned a value by the effect of an action, then it takes its value from the previous time step, ensuring that the system state does not change unless perturbed.

The rules of the *plan module* are given in Listing 9 and function as follows; (1) for each time step k greater than the starting step j , (2-3) plan actions sequentially (exactly one action per time step), (5-7) apply the direct effects of an action on the same time step it was planned at, (8-9) a positive final goal literal is satisfied if its fluent literal holds, (10-11) a negative final goal literal is satisfied if its fluent literal does not hold, (12-13) a monolevel plan is incomplete if at least one final goal is unsatisfied in the terminal state at step k , (9-10) do not return until a complete plan is found. Of particular interest is the special external atom *query(k)* whose truth value is assignable by external Python script. The atom enforces a constraint ensuring that the program is unsatisfiable until a plan of length k is found.

B.3. Conformance Module

The conformance module is an extension of the planning module, used only in conformance refinement planning.

The rules of the *conformance module* are given in Listing 8 and function as follows; (1) for each time step k greater than the starting step j , enforce subgoal stages at sequence indices greater than or equal to the starting index φ and less than the final index ϑ , (2-3) for each action planned at the previous abstraction level, there is a subgoal for each of its direct effects whose sequence index is equal to the abstract time step T which is was planned at, (4) the current subgoal stage to be achieved from the starting step j is initially the subgoal stage is at the first index φ , (5-7) the current subgoals are propagated to the next time step k if the subgoal index that was current at the previous time step $k - 1$ has not yet been achieved, (8-10) otherwise, if the sequence index that was current at the previous time step is no longer current at k then the next subgoal sequence index $T + 1$ becomes current, (11-12) the subgoal stage sequence index T which is currently being achieved at k is equal to the index value of the current subgoals, (13-15) a subgoal is satisfied on the step its value holds in the state, (16-18) the current subgoal stage is unachieved whilst any its subgoals are not satisfied, (19-20) the plan is incomplete whilst there is at least one current subgoal i.e. all subgoal sequence indices must be exhausted to complete the plan.

CRedit authorship contribution statement

Oliver Michael Kamperis: Writing, theory, software and experiments. **Marco Castellani:** Writing, data curating, and proofing. **Yongjing Wang:** Writing and proofing.

References

- [1] Aker, E., Patoglu, V., Erdem, E., 2012. Answer set programming for reasoning with semantic knowledge in collaborative housekeeping

- robotics. IFAC Proceedings Volumes 45, 77–83.
- [2] Andres, B., Obermeier, P., Sabuncu, O., Schaub, T., Rajaratnam, D., 2013. Rosoclingo: A ros package for asp-based robot control. CoRR abs/1307.7398.
- [3] Andres, B., Rajaratnam, D., Sabuncu, O., Schaub, T., 2015. Integrating asp into ros for reasoning in robots, in: International Conference on Logic Programming and Nonmonotonic Reasoning, Springer. pp. 69–82.
- [4] Babb, J., 2015. Action language bc +.
- [5] Bacchus, F., Yang, Q., 1994. Downward refinement and the efficiency of hierarchical problem solving. Artificial Intelligence 71, 43–100.
- [6] Balduccini, M., Regli, W.C., Nguyen, D.N., 2014. An asp-based architecture for autonomous uavs in dynamic environments: Progress report. CoRR abs/1405.1124.
- [7] Baral, C., 2003. Knowledge Representation, Reasoning and Declarative Problem Solving with Answer Sets. Cambridge University Press. doi:10.1017/CB09780511543357.
- [8] Bercher, P., Alford, R., Höller, D., 2019. A survey on hierarchical planning-one abstract idea, many concrete realizations., in: IJCAI, pp. 6267–6275.
- [9] Bylander, T., 1994. The computational complexity of propositional strips planning. Artificial Intelligence 69, 165–204.
- [10] Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F., Schaub, T., 2012. Asp-core-2: Input language format. ASP Standardization Working Group.
- [11] Chen, Y., Hsu, C.W., Wah, B.W., 2004. Sgplan: Subgoal partitioning and resolution in planning. Edelkamp et al.(Edelkamp, Hoffmann, Littman, & Younes, 2004).
- [12] Chinchalkar, S., 1996. An upper bound for the number of reachable positions. ICGA Journal 19, 181–183.
- [13] Clark, K.L., 1978. Negation as failure, in: Logic and data bases. Springer, pp. 293–322.
- [14] Dimopoulos, Y., Gebser, M., Lühne, P., Romero, J., Schaub, T., 2017. plasp 3: Towards effective asp planning, in: International Conference on Logic Programming and Nonmonotonic Reasoning, Springer. pp. 286–300.
- [15] Erdem, E., Patoglu, V., 2018. Applications of asp in robotics. KI - Künstliche Intelligenz 32, 143–149.
- [16] Erol, K., Hendler, J., Nau, D.S., 1994. Htn planning: Complexity and expressivity, in: AAAI, pp. 1123–1128.
- [17] Erol, K., Hendler, J.A., Nau, D.S., 1995. Semantics for hierarchical task-network planning. Technical Report. MARYLAND UNIV COLLEGE PARK INST FOR SYSTEMS RESEARCH.
- [18] Esra Erdem, Michael Gelfond, N.L., 2016. Applications of answer set programming.
- [19] Fikes, R., Nilsson, N.J., 1971. Strips: A new approach to the application of theorem proving to problem solving. Artificial Intelligence 2, 189–208. doi:10.1016/0004-3702(71)90010-5.
- [20] Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T., Thiele, S., Wanko, P., 2019a. Potassco User Guide. 2nd ed. University of Potsdam. Am Neuen Palais 10, 14469 Potsdam, Germany. Available at: <https://github.com/potassco/guide/releases>, Accessed: 18/01/2021.
- [21] Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., 2014. Clingo = asp + control: Extended report. CoRR abs/1405.3694.
- [22] Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., 2019b. Multi-shot asp solving with clingo. TPLP 19, 27–82.
- [23] Gebser, M., Maratea, M., Ricca, F., 2019c. The seventh answer set programming competition: Design and results. CoRR abs/1904.09134.
- [24] Gelfond, M., Inclezan, D., 2013. Some properties of system descriptions of ald. Journal of Applied Non-Classical Logics 23, 105–120.
- [25] Gelfond, M., Kahl, Y., 2014. Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach. Cambridge University Press, New York, NY, USA.
- [26] Gelfond, M., Lifschitz, V., 1988. The stable model semantics for logic programming, in: Kowalski, R., Bowen, Kenneth (Eds.), Proceedings of International Logic Programming Conference and Symposium, MIT Press. pp. 1070–1080.
- [27] Gelfond, M., Lifschitz, V., 1991. Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385. doi:10.1007/BF03037169.
- [28] Gelfond, M., Lifschitz, V., 1998. Action languages. Electronic Transactions on Artificial Intelligence 3, 195–210.
- [29] Georgievski, I., Aiello, M., 2015. Htn planning: Overview, comparison, and beyond. Artificial Intelligence 222, 124–156.
- [30] Ghallab, M., Knoblock, C., Wilkins, D., Barrett, A., Christianson, D., Friedman, M., Kwok, C., Golden, K., Penberthy, S., Smith, D., Sun, Y., Weld, D., 1998. Pddl - the planning domain definition language.
- [31] Giunchiglia, F., Walsh, T., 1992. A theory of abstraction. Artificial intelligence 57, 323–389.
- [32] Gupta, N., Nau, D.S., 1992. On the complexity of blocks-world planning. Artificial Intelligence 56, 223–254.
- [33] Hart, P.E., Nilsson, N.J., Raphael, B., 1968. A formal basis for the heuristic determination of minimum cost paths. IEEE transactions on Systems Science and Cybernetics 4, 100–107.
- [34] Helmert, M., 2003. Complexity results for standard benchmark domains in planning. Artificial Intelligence 143, 219–262.
- [35] Hoffmann, J., Sabharwal, A., Domshlak, C., 2006. Friends or foes? an ai planning perspective on abstraction and search., in: ICAPS, pp. 294–303.
- [36] Jiang, Y.q., Zhang, S.q., Khandelwal, P., Stone, P., 2019. Task planning in robotics: an empirical comparison of pddl-and asp-based systems. Frontiers of Information Technology & Electronic Engineering 20, 363–373.
- [37] Kaminski, R., Schaub, T., Wanko, P., 2017. A Tutorial on Hybrid Answer Set Solving with clingo. pp. 167–203.
- [38] Khandelwal, P., Zhang, S., Sinapov, J., Leonetti, M., Thomason, J., Yang, F., Gori, I., Svetlik, M., Khante, P., Lifschitz, V., et al., 2017. Bwibots: A platform for bridging the gap between ai and human-robot interaction research. The International Journal of Robotics Research 36, 635–659.
- [39] Knoblock, C.A., 1990a. Learning abstraction hierarchies for problem solving., in: AAAI, pp. 923–928.
- [40] Knoblock, C.A., 1990b. A theory of abstraction for hierarchical planning, in: Change of Representation and Inductive Bias. Springer, pp. 81–104.
- [41] Knoblock, C.A., 1991. Automatically Generating Abstractions for Problem Solving. Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.
- [42] Knoblock, C.A., 1992. An analysis of abstrips, in: Artificial Intelligence Planning Systems, Elsevier. pp. 126–135.
- [43] Knoblock, C.A., Tenenber, J.D., Yang, Q., 1991. Characterizing abstraction hierarchies for planning., in: AAAI, pp. 692–697.
- [44] Lee, J., Lifschitz, V., Yang, F., 2013. Action language bc: Preliminary

Listing 10: Rules of the Plan Conformance Module

```

1 #program conformance_module(j, k,  $\varphi$ ,  $\zeta$ ).
2 sgoal(L, F, V, T) :- occ(L, R, A, T), T >=  $\varphi$ , T <=  $\zeta$ ,
3     eff(L, R, A, F, V, T), pl(L-1).
4 cur_sgoal(L, F, V,  $\varphi$ , j) :- sgoal(A, F, V,  $\varphi$ ), pl(L-1).
5 cur_sgoal(L, F, V, T, k) :- sub_goal(L, F, V, T),
6     cur_sgoal(L, F, V, T, k-1),
7     sgoals_unach(L, T, k), pl(L-1).
8 cur_sgoal(L, F, V, T+1, k) :- sgoal(L, F, V, T+1),
9     cur_sgoal_ind(L, T, k-1),
10    not cur_sgoal_ind(L, T, k), pl(L-1).
11 cur_sgoal_ind(L, T, k) :- sgoal(L, F, V, T),
12    cur_sgoal(L, F, V, T, k), pl(L-1).
13 sgoal_sat(L, F, V, T, k) :- sgoal(L, F, V, T),
14    cur_sgoal(L, F, V, T, k-1),
15    hld(L, F, V, k), pl(L-1).
16 sgoals_unach(L, T, k) :- sgoal(L, F, V, T), cur_sgoal_ind(L, T, k-1),
17    not sgoal_sat(L, F, V, T, k), pl(L-1).
18 incomplete_plan(k) :- sgoal(L, F, V, T),
19    cur_sgoal(L, F, V, T, k), pl(L-1).
    
```

- report, in: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI).
- [45] Lifschitz, V., 2008. What is answer set programming?, in: Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3, pp. 1594–1597.
 - [46] Newell, A., Simon, H.A., et al., 1972. Human problem solving. volume 104. Prentice-Hall Englewood Cliffs, NJ.
 - [47] Russell, S.J., Norvig, P., 2016. Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited.
 - [48] Sacerdoti, E.D., 1974. Planning in a hierarchy of abstraction spaces. Artificial intelligence 5, 115–135.
 - [49] 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.
 - [50] Shannon, C.E., 1950. Xxii. programming a computer for playing chess. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 41, 256–275.
 - [51] Sridharan, M., Gelfond, M., Zhang, S., Wyatt, J., 2019. Reba: A refinement-based architecture for knowledge representation and reasoning in robotics. Journal of Artificial Intelligence Research 65, 87–180.
 - [52] Timpf, S., Volta, G.S., Pollock, D.W., Egenhofer, M.J., 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.
 - [53] Washington, R., 1994. Abstraction Planning in Real Time. Technical Report. STANFORD UNIV CA DEPT OF COMPUTER SCIENCE.
 - [54] Zhang, S., Sridharan, M., Wyatt, J.L., 2015a. Mixed logical inference and probabilistic planning for robots in unreliable worlds. IEEE Transactions on Robotics 31, 699–713.
 - [55] Zhang, S., Yang, F., Khandelwal, P., Stone, P., 2015b. Mobile robot planning using action language *BC* with an abstraction hierarchy, in: International Conference on Logic Programming and Nonmonotonic Reasoning, Springer. pp. 502–516.

Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography.

Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography.

Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography. Author biography.