To appear in *Workshop on Answer Set Programming and Other Computing Paradigms* (ASPOCP 2014), Vienna, Austria, July 2014.

Mobile Robot Planning using Action Language $\mathcal{BC}$ with Hierarchical Domain Abstractions

Shiqi Zhang, Fangkai Yang, Piyush Khandelwal and Peter Stone

Department of Computer Science, The University of Texas at Austin
2317 Speedway, Stop D9500, Austin, TX 78712, USA
{szhang,fkyang,piyushk,pstone}@cs.utexas.edu

Abstract. Action language $\mathcal{BC}$ provides an elegant way of formalizing robotic domains which need to be expressed using default logic as well as indirect and recursive action effects. However, generating plans efficiently for large domains using $\mathcal{BC}$ can be challenging, even when state-of-the-art answer set solvers are used. In this paper, we investigate the computational gains achieved by describing task planning domains at different abstraction levels using $\mathcal{BC}$, where lower levels describe more domain details by adding *fluents* not included in higher levels and actions at different levels are formalized independently. Two algorithms are presented to efficiently calculate the near-optimal short and low-cost plans respectively. We present a case study where at least an order of magnitude speedup was achieved in a robot mail collection task using hierarchical domain abstractions.

1 Introduction

Action language $\mathcal{C}+$ [11] was designed to formalize indirect action effects and default logic, and the recently proposed language $\mathcal{BC}$ [16] can also describe recursive action effects. These languages are attractive in robotic domains as they solve the *frame problem* [19] by formalizing the commonsense law of inertia. Action descriptions written in these languages can be automatically translated into a logic program under the stable model semantics [9, 10] by software such as CPLUS2ASP [1], and planning can be accomplished using computational methods of Answer Set Programming (ASP) [18, 21] through state-of-the-art answer set solvers such as CLINGO [8]. For these reasons, answer set programming and action languages have been widely used with mobile robots in recent years [2, 6, 13, 25].

This paper continues an existing line of research of using $\mathcal{BC}$ for robot task planning [13]. In that paper, an action description in $\mathcal{BC}$ is used to formalize a dynamic domain where a mobile robot operates inside a building and fulfills tasks such as collecting outgoing mail intended for delivery. Given a task, the robot can generate the shortest plan that minimizes the number of actions, or generate the lowest-cost plan that minimizes expected execution time by associating a cost (i.e. time) with each individual action. Generating the lowest-cost plan can take a prohibitively long time. For instance, in a mail collection task, for a reasonably sized domain which contains 20 rooms, 25 doors, and 10 people from whom mail needs to be collected, generating the plan and verifying that this plan is minimal can take more than 2 hours on a modern

desktop machine. Long planning times for a mobile robot can be problematic as a mobile robot has limited battery life. Furthermore, if the robot also frequently interacts with humans, planning times need to be reasonable for that robot to be deemed useful and accepted by humans. For these reasons, this paper presents a case study where a robot planning domain is formulated hierarchically improving planning efficiency.

Instead of using a single action description to formalize the domain, we use a list of action descriptions, $L_1,\ L_2,\ \cdots,\ L_n$ to formalize the domain at different abstraction levels. For two action descriptions L_i and L_j, where $i < j$, L_j formalizes more domain details than L_i, using more fluents to describe the states of the domain in finer granularity, and describing actions that can change these fluents. These action descriptions thus form an abstraction hierarchy, where the high-level descriptions are more abstract than the low-level descriptions, although each of them is an action description in $\mathcal{BC}$.

To generate a plan for a planning query, a higher level description is used to generate an abstract plan, which serves as a guideline to reduce the search space when generating plans at lower levels. Given a planning query Q, we begin with the top-level description L_1, and generate the most-abstract plan Pl_1. Pl_1 is turned into a set of "domain constraints" which are added to L_2 to generate plan Pl_2. This process continues until the lowest-level plan Pl_n is generated using L_n and the domain constraints obtained from Pl_{n-1}. Pl_n only contains elementary actions that can be directly executed by a robot. We use this idea to generate both near-optimal short plans and low-cost plans.

We compare our approach against previous work [13] for a mail collection task, and show that the efficiency of planning improves by at least an order of magnitude for generating both the short and low-cost plans.

2 Related Work

Previous work that has investigated ideas of macro-actions and plan expansion using STRIPS [15], HTN [7], Golog [17, 20], answer set programming [24, 4], and action languages [3, 12]. In these works, macro-actions (also called *complex actions* or *composite actions*) are described as a sequence of primitive actions and possibly some imperative constructs similar to those in procedural programming language. These macro-actions are either macros that are directly expanded after a plan is generated, or expanded in the reasoning process using a predefined structure. Encoding the *methods* to expand the macro-actions can be difficult and time-consuming even for domain experts [27]. In our work, each level is an action description in the syntax of action language $\mathcal{BC}$. We do not explicitly describe the relationship between actions in different levels. Their correspondence is established during plan generation through the usage of state constraints. This method provides higher flexibility than traditional hierarchical planning, because high-level actions are not necessarily treated as macros that can only be expanded into certain forms of primitive actions. Previous work also investigated hierarchical planning using partially observable Markov decision processes (POMDPs) on mobile robots [26]. However, it is a challenge to use POMDPs to solve complex problems in large domains with a large number of states, even after the problems have been decomposed hierarchically.

Planning with abstraction has been studied before [14, 23]. We borrow the idea of representing problem domains as a hierarchy of abstractions where successively finer

levels of detail are introduced. Unlike previous work that holds the *ordered monotonicity* property, which guarantees that the structure of an abstract plan is not changed in the process of refining it [14], our approach intentionally allows low-level plans to not strictly follow higher-level plans if lower-cost plans can be achieved. The independence of actions at different levels provides low levels higher flexibility in planning to reduce the overall cost, while still being able to plan faster. Additionally, our work uses action language $\mathcal{BC}$ that allows planning with incomplete knowledge and indirect action effects in domains with dynamic changes [16]. Finally, In all previous work, individual action costs are not considered, while we also generate low-cost plans.

3 Domain Representation

We use a mail-collection domain (that was previously presented in [13]) to describe our hierarchical planning approach:

> *The robot drops by offices at 2pm every day to collect outgoing mail from the residents. However, some people may not be in their offices at that time, so they can pass their outgoing mail to colleagues in other offices, and send this information to the robot. When the robot collects the mail, it should obtain it while only visiting people as necessary.*

We use an example floor plan, illustrated in Figure 1, to describe how the domain is formalized. In this example, we consider the objects shown below. In the following subsections, we will use meta-variables $P, P_1, P_2, \ldots$ to denote people, $R, R_1, R_2, \ldots$ to denote rooms, and $D, D_1, D_2, \ldots$ to denote doors.

- *alice*, *bob*, *carol*, *daniel* and *erin* are people.
- o_1, o_2, o_3, o_4 are offices, lab_1 is a lab and *cor* is a room, where offices and labs are sub-sorts of room.
- d_1, d_2, d_3, d_4, d_5, d_6 and d_7 are doors.

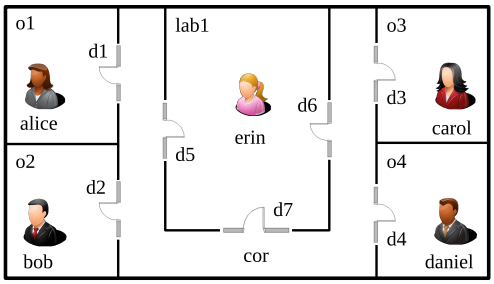

Fig. 1: The layout of the floor plan used in experiments.

3.1 Hierarchical Domain Representation

This mail collection problem can be hierarchically described using the following three levels of abstraction, all in the language of $\mathcal{BC}$ [16]. The highest level description (L_1) abstracts out all low-level domain details, and only formalizes if each person has been served or not. The plan generated at this level only dictates the order in which mail will be collected from people in the building. In level 2 description (L_2), more fluents are

added to describe the room connections through doors, so plans at this level decide not only the order of people to be served but also how to access people. A plan generated at level 1 is used as a guideline at level 2 by passing state constraints downward. The lowest level description (L_3) includes all domain details that are needed to select primitive actions that can be directly executed by the mobile robot, such as approaching a door, opening a door, going through a door, etc. Section 4 provides more details on the plan generation procedure. The following subsections detail these three levels.

Level 1 Formalization: L_1 only describes the people inside the building and their offices, the information about how mail was passed from one person to another, and the robot's location. The spatial details for navigation are completely abstracted out. This level is described by the following fluents and laws:

- The current location of a person is expressed by the fluent *inside*. *inside*(P, R) means that person P is located in room R. A person can only be inside a single room at any given time. The fluent is *inertial*[1]:

$$\sim\textit{inside}(P, R_2) \ \textbf{if} \ \textit{inside}(P, R_1) \ \ (R_1 \neq R_2). \quad \textbf{inertial} \ \textit{inside}(P, R).$$

- The current location of robot is represented by a multi-valued inertial fluent *loc*. *loc* $= R$ expresses that the robot is in room R.
- The fluent *passto*(P_1, P_2) describes whether a person P_1 has passed mail to person P_2. By default, a person P_1 has not passed mail to a person P_2.
- The fluent *mailcollected*(P) describes whether the robot has collected mail from P. This fluent is inertial. It is recursively defined as follows. The robot has collected P_1's mail if it has collected P_2's mail and P_1 has passed his mail to P_2.

$$\textit{mailcollected}(P_1) \ \textbf{if} \ \textit{mailcollected}(P_2), \ \textit{passto}(P_1, P_2).$$

We formalize the actions that can lead state transitions based on the fluents at the same level. Serving a person P results in his mail being collected, and the robot being present in his office:

$$\textit{serve}(P) \ \textbf{causes} \ \textit{mailcollected}(P). \quad \textit{serve}(P) \ \textbf{causes} \ \textit{loc} = R \ \textbf{if} \ \textit{inside}(P, R).$$

Level 2 Formalization: L_2 includes all the fluents described in L_1, and furthermore, adds information about doors, and how these doors make a room accessible from the adjacent room. The following fluents and laws are introduced in level 2:

- *hasdoor*(R, D): office R has door D, e.g., *hasdoor*(o_1, d_1) and *hasdoor*(lab_1, d_5). The default below expresses the closed world assumption [22] for *hasdoor* and states that an office R does not have a door D unless it is specified.

$$\textbf{default} \ \sim\textit{hasdoor}(R, D).$$

- *acc*(R_1, D, R_2): room R_1 is accessible from room R_2 via door D. Two rooms are not connected by a door unless specified.

$$\textit{acc}(R_1, D, R_2) \ \textbf{if} \ \textit{hasdoor}(R_1, D), \ \textit{hasdoor}(R_1, D).$$
$$\textit{acc}(R_1, D, R_2) \ \textbf{if} \ \textit{acc}(R_2, D, R_1). \ \textbf{default} \ \ \sim\textit{acc}(R_1, D, R_2).$$

[1] An inertial fluent is a fluent whose value does not change with time by default.

This level formalizes an action *collectmail*(P), which describes whether the robot collected mail from person P. A robot can only collect mail from a person if both are in the same room, if the person has not passed their mail to someone else, and if the person's mail has not been collected yet. Collecting mail from a person P results in the *mailcollected*(P) fluent being true, formalized as:

$$\textit{collectmail}(P) \ \textbf{causes} \ \textit{mailcollected}(P).$$

Action *cross*(D) allows the robot to cross door D to move from room R_1 to room R_2, if R_2 is accessible from R_1 through door D, formalized as:

$$\textit{cross}(D) \ \textbf{causes} \ \textit{loc} = R_2 \ \textbf{if} \ \textit{loc} = R_1, \ \textit{acc}(R1, D, R_2).$$

The next rule is a restriction on the executability of *cross*(D): the robot cannot cross a door if that door is not accessible from the robot's current location:

$$\textbf{nonexecutable} \ \textit{cross}(D), \ \textit{loc} = R, \ \sim\textit{hasdoor}(R, D).$$

Level 3 Formalization: L_3 contains all the fluents described in levels L_1 and L_2, and further introduces the following fluents[2]:

- *facing*(D) expresses that the robot is next to a door D and is facing it. The robot needs to face a door to sense if it is open, before going through it.
- *beside*(D) expresses that the robot is next to door D. *beside*(D) is true if *facing(D)* is true. Since *beside* is implied by *facing*, it will become an indirect effect of the actions that make the fluent *facing* true.
- *open*(D) expresses if the door D is open. By default, all doors are closed.

The actions at level 3, the lowest level, are all primitive actions that are executable on real robots. L_3 includes the following actions:

- *approach*(D): the robot approaches door D. The robot can only approach a door accessible from the the robot's current location if it is not facing that door already. Approaching a door causes the robot to face that door.
- *gothrough*(D): the robot goes through door D. The robot can only go through a door if the door is accessible from the robot's current location, if it is open, and if the robot is facing it. Executing the *gothrough* action results in the robot's location being changed to the connecting room and the robot no longer faces the door.
- *opendoor*(D): the robot opens a closed door D. The robot can only open a door that it is facing it.

3.2 Planning Query

A planing problem includes a domain description and a *planning query*. A planning query consists of a set of conditions that describes the initial state, goal state, and possibly some intermediate states. The planning query can be written as rules in ASP, and when merged with the ASP code obtained from the domain description will guide CLINGO to generate answer sets that satisfy all conditions in the planning query.

[2] More detailed action descriptions at this level were previously presented in [13].

Similar to previous work [13], the planner obtains the initial state from two sources. First, it outputs the initial state from the tables where it maintains the value of fluents *inside* and *passto*. As an example, we consider the following *passto* relationship where $passto(bob, alice)$ and $passto(daniel, bob)$ are true. Second, the planner polls the sensors to obtain the values for fluents *beside*, *facing*, *open* and *loc*. The sensors guarantee that the value for *loc* is always returned for exactly one location, and *beside* and *facing* are returned with at most one door. If the robot is facing a door, the value of *open* for that door is sensed and returned as well. For instance, in the initial state, if the robot is in the corridor and not facing any door, the planner senses and appends (1) to the description as the initial state.

$$0 : loc = cor,\ 0 : \sim beside(D),\ 0 : \sim facing(D). \tag{1}$$

The goal (2) indicates that the planner should find a plan at most *maxLength* in steps that satisfies this goal. In order to use the answer set solver CLINGO to generate a short plan, the solver is repeatedly called with increasing values of *maxLength* trying to search for a plan, up to a user defined *upper-bound*, until a plan is found.

$$maxLength : mailcollected(P). \tag{2}$$

While $\mathcal{BC}$ can be automatically translated to ASP using CPLUS2ASP, we follow the translation to ASP [16] manually to produce more optimized code.

4 Algorithms

This section will present two algorithms for efficiently generating near-optimal short and low-cost plans respectively. The latter problem generalizes the former by associating each action with a different cost. We will first describe the algorithm for generating short plans and then focus on the more challenging problem of generating low-cost plans. The goal is to enable robots to calculate plans efficiently while compromising on the optimality at a minimal degree.

4.1 Generating Short Plans

In the case of 3 levels of hierarchical domain abstraction, a short plan is generated as follows. Assuming that the query contains the following initial conditions:

$$0 : \sim mailcollected(P),\ 0 : loc = cor, \tag{3}$$

and goal state (2), this query is combined with L_1, tabular information about *passto* and *inside*, and relevant portions (only *loc*) of the sensor information (1) and then sent to CLINGO. CLINGO returns the plan with *maxLength* as 2, as shown in (4), which indicates that $serve(alice)$ is executed at time 0, and $serve(carol)$ is executed at time 1.

$$0 : serve(alice), 1 : serve(carol). \tag{4}$$

The output from CLINGO also contains the expected values of the fluents at each time step. For instance, the expected state after collecting mail from Alice at level 1 is shown in (5). Similarly, the expected values of fluents at time 2 are shown in (6).

$$\begin{array}{l} 1 : mailcollected(alice), 1 : mailcollected(bob), \\ 1 : \sim mailcollected(carol), 1 : mailcollected(daniel), 1 : loc = o_1, \end{array} \tag{5}$$

Algorithm 1 Generating Near-optimal Short Plan

Require: domain description at level i, $D_i, i \in \{1, \ldots, N\}$
Require: planning query Q
1: call CLINGO to generate answer set A_1 using D_1 and Q for the smallest value of *maxLength* ≥ 0.
2: extract from A_1 the sequence of states $S = (0 : s_0, \ldots, k : s_k)$ and sequence of actions $P = (0 : a_0, \ldots, k-1 : a_{k-1})$.
3: **for** level i, $i \in \{2, \ldots, N\}$ **do**
4: **for** $j \in \{0, \ldots, \mathit{length}(P) - 1\}$ **do**
5: **if** a_j is a non-primitive action **then**
6: define query Q_{ij} that contains initial condition $0 : s_{j-1}$ and goal condition *maxLength* $: s_j$
7: call CLINGO to generate the answer set A_{ij} using D_i and Q_{ij} for the smallest value of *maxLength* ≥ 0
8: extract from A_{ij} the sequence of states S_{ij} $(0 : s_0, \ldots, l : s_l)$ and sequence of actions $P_{ij} = (0 : a_0, \ldots, l-1 : a_{l-1})$.
9: elaborate P using P_{ij}, elaborate S using S_{ij}
10: **end if**
11: **end for**
12: **end for**
13: **return** P

$$\begin{array}{l} 2 : \mathit{mailcollected}(\mathit{alice}), 2 : \mathit{mailcollected}(\mathit{bob}), \\ 2 : \mathit{mailcollected}(\mathit{carol}), 2 : \mathit{mailcollected}(\mathit{daniel}), 2 : \mathit{loc} = o_3. \end{array} \tag{6}$$

To calculate a plan at level 2, the state (3) before executing action *serve*(*alice*) becomes the new initial state, and the state (5) after executing action *serve*(*alice*) becomes the goal state as shown in (7). The query, (3) and (7), combined with the level 2 formalization, is processed by CLINGO, which generates a short plan when *maxLength* $= 2$ as shown in (8).

$$\begin{array}{l} \mathit{maxLength} : \mathit{mailcollected}(\mathit{alice}), \mathit{maxLength} : \mathit{mailcollected}(\mathit{bob}), \\ \mathit{maxLength} : \sim \mathit{mailcollected}(\mathit{carol}), \\ \mathit{maxLength} : \mathit{mailcollected}(\mathit{daniel}), \mathit{maxLength} : \mathit{loc} = o_1, \end{array} \tag{7}$$

$$0 : \mathit{cross}(d_1), 1 : \mathit{collectmail}(\mathit{alice}). \tag{8}$$

Executing this plan achieves the same effect of executing action *serve*(*alice*), which can be verified by ensuring that the state described by CLINGO after executing *collectmail*(*alice*) is identical to (5). Similarly, as shown in (9) action *serve*(*carol*) can be elaborated using (5) as the initial state, and the following goal state from (6). CLINGO generates the plan (*maxLength* $= 3$) as shown in (10), which is the elaboration corresponding to *serve*(*carol*).

$$\begin{array}{l} \mathit{maxLength} : \mathit{mailcollected}(\mathit{alice}), \mathit{maxLength} : \mathit{mailcollected}(\mathit{bob}), \\ \mathit{maxLength} : \mathit{mailcollected}(\mathit{carol}), \mathit{maxLength} : \mathit{mailcollected}(\mathit{daniel}), \\ \mathit{maxLength} : \mathit{loc} = o_3. \end{array} \tag{9}$$

$$0{:}\textit{cross}(d_1), 1{:}\textit{cross}(d_3), 2{:}\textit{collectmail}(\textit{carol}) \tag{10}$$

Next, the plan generated from the level 2 formalization is obtained by replacing actions in (4) by their corresponding elaborated action sequences (8) and (10):

$$\begin{array}{l} 0{:}\textit{cross}(d_1), 1{:}\textit{collectmail}(\textit{alice}), 2{:}\textit{cross}(d_1), 3{:}\textit{cross}(d_3), \\ 4{:}\textit{collectmail}(\textit{carol}). \end{array} \tag{11}$$

Plan (11) can be similarly elaborated using the level 3 formalization. It should be noted that when elaborating task $\textit{cross}(d_1)$, the complete initial sensor state (1) and tabular information is used. The final elaborated short plan that is produced as below:

$$\begin{array}{l} 0{:}\textit{approach}(d_1), 1{:}\textit{opendoor}(d_1), 2{:}\textit{gothrough}(d_1), 3{:}\textit{collectmail}(\textit{alice}), \\ 4{:}\textit{approach}(d_1), 5{:}\textit{opendoor}(d_1), 6{:}\textit{gothrough}(d_1), 7{:}\textit{approach}(d_3), \\ 8{:}\textit{opendoor}(d_3), 9{:}\textit{gothrough}(d_3), 10{:}\textit{collectmail}(\textit{carol}). \end{array}$$

This plan only consists of primitive actions that can be executed by the robot. The complete algorithm for generating the near-optimal short plan over multiple levels of hierarchy is described in Algorithm 1, assuming the planning query is satisfiable.

4.2 Generating Low-Cost Plans

In this section, we will adapt lowest cost plan generation using $\mathcal{BC}$ [13] to work with hierarchical domain abstractions toward near-optimal low-cost plans.

Costs of Actions: All actions in the hierarchical formalization can be associated with costs. Intuitively, the cost of an action represents its execution time. While the actual cost values used in this paper are not relevant to demonstrate the computational advantage of using a hierarchical domain abstraction, it is still necessary to investigate how costs can be incorporated in this framework. In our example, actions *opendoor*, *gothrough*, and *collectmail* have fixed costs of 1, and the costs of *approach*, *cross* and *serve* are determined both by the argument of the action constant and the state before the action. Following the architecture proposed in previous work [13], the cost can be estimated using a dedicated low-level navigation module external to CLINGO. While solving, CLINGO makes external procedure calls to compute this cost.

In our domain, the cost of *approach* is computed in two different ways:

- When the robot approaches door D_1 from door D_2 while in room R, the fluents uniquely identify the start and finish physical locations of the robot in the environment. The cost of action $\textit{approach}(D_1)$ is specified by an external term `@cost(D1,D2,R)`, and computed by the external module.
- When the robot is not beside a door and begins to approach to a door, the logical abstraction cannot sufficiently capture the true location of the robot at the start of the action. However, this situation only occurs at the start of a plan, when the external module knows the true location of the robot. Approaching door D initially can be computed using the external term `@initialcost(D,R)`.

The cost for executing action *serve*(P) also depends on the physical location of the robot. When the robot serves P located in office R_1, and the robot itself is currently located in room R_2, the cost of action *serve*(P) is specified by an external term `@costserve(P,R1,R2)`. Similar to *approach*, the external module can estimate the cost of this action. It should be noted that the location abstraction provided to the external module for computing the cost of *serve* is at the room level, different from *approach* where proximity to a door was much more suitable for uniquely identifying the robot's location. As a result, cost estimates of *serve* may not be extremely accurate. In a similar fashion, the cost of action *cross*(D) can be computed using the external term `@costcross(D,R)`.

Estimating Upper-Bounds: When costs are used with CLINGO, unlike the case of short plan generation, it is no longer possible to call CLINGO repeatedly with increasing values of *maxLength*, as the shortest plan may not be the one that has the lowest cost. While searching for low-cost plans, *maxLength* is directly set to the user-specified upper-bound [13]. An upper-bound exponential relative to the number of grounded fluents can produce the provably correct low-cost plan [5]. Significant domain-specific knowledge is used to estimate this upper-bound which reaches a reasonable compromise between optimality and efficiency.

When planning across different levels of formalization in hierarchical domain abstraction, it is possible to supply an arbitrarily loose upper-bound to CLINGO at each level to ensure that the optimal plan is found. Instead, we'll improve computational efficiency by providing more appropriate upper-bounds at each level using knowledge about the hierarchy.

Furthermore, as we'll show in the next section, for generating the low-cost plans we recompute the entire plan at each level of the hierarchy using the plan at a higher level as a guide, instead of simply elaborating high-level actions. This recomputation makes it difficult to prespecify reasonable upper-bounds without knowledge of the plan generated in the level above. For this reason, in our framework, we provide a parametrized heuristic approach to estimate these upper-bounds. This heuristic allows users to provide domain knowledge independent of the planning query, which can then be used to compute the upper-bounds at runtime. In experiments, the upper-bounds estimated by this approach lead to a good compromise between optimality and efficiency.

To estimate the upper-bounds for the three levels at runtime, we need to prespecify three values (e_1, e_2, e_3):

- The value e_1 denotes the upper-bound of plans in level 1. In our case, at level 1, there exists only one macro-action *serve*. In order to serve 4 people, no plan should ever be more than 4 steps. Therefore, it is reasonable to assign e_1 to be 4.
- The value e_2 denotes the upper-bound on the elaborations of macro-actions from level 1. In our case, we need to estimate the upper-bound of a plan that achieves the same effect as *serve*(P). Consider the worst case of elaborating *serve*(P), where the robot is located in o_1, and needs to collect mail from *carol*. In this situation, the longest plan that *serve*(*carol*) can be elaborated to without crossing a door twice is *cross*(d_1), *cross*(d_5), *cross*(d_6), *cross*(d_3) and *collectmail*(*carol*). Therefore, it is reasonable to assign e_2 to be 5.

– The value e_3 denotes the upper-bound of the elaborations of macro-actions from level 2. In our case, we need to estimate the upper-bound of a plan that achieves the same effect of *cross*(D). The worst case is that the robot executes three actions to cross a closed door D: *approach*(D), *opendoor*(D), *gothrough*(D). Therefore, it is reasonable to assign e_3 to be 3.

Given the value of e_1, e_2, e_3, we can determine the value of the upper-bounds for the overall plan at all three levels:

– e_1 is the upper-bound in level 1. Denote it as B_1: $B_1 = e_1$.
– Let l_1 be the total steps in the low-cost plan generated in level 1 and $l_{\textit{serve}}$ denotes the number of actions *serve* occurring in that plan. Then the upper-bound in level 2, denoted as B_2, is

$$B_2 = e_2 \times l_{\textit{serve}} + (l_1 - l_{\textit{serve}}). \tag{12}$$

– Let l_2 be the total steps of the low-cost plan generated in level 2 and $l_{\textit{cross}}$ denotes the number of actions *cross* occurring in that plan. Then the upper-bound in level 3, denoted as B_3, is

$$B_3 = e_3 \times l_{\textit{cross}} + (l_2 - l_{\textit{cross}}). \tag{13}$$

These three upper-bounds are used during low-cost plan generation at each level. We will generalize this approach from the case study presented in this paper to an abstract framework in the future.

Illustrative Example: Consider the following planning query: initially the robot is located in $\textit{lab}_1$, and the goal of the robot is to collect mail from all 4 people. At level 1, we use the optimization statement, `#minimize{X,Y:costserve(X,Y)}`, to guide CLINGO to generate the lowest cost plan at *maxLength* $= B_1 = 4$. The robot only needs to collect mail from Alice and Carol, as Alice has Bob's and Dan's mail. There are a total of 12 plans that can satisfy this query, as CLINGO returns plans with no-operation for 2 of 4 timesteps. Plan (4) is one such plan. Another possible plan is:

$$1{:}\textit{serve}(\textit{carol}), 3{:}\textit{serve}(\textit{alice}).$$

Between all the 12 plans, there may be at most only 2 unique cost values. 6 equivalent plans will serve Alice first, and 6 equivalent plans will serve Carol first. Depending on the value of external cost term `costserve`, the plan that serves one of them first may have lower cost. One is arbitrarily selected from these 6 lowest cost plans by CLINGO. Let's assume for simplicity that (4) is returned. Recall that once (4) is generated, the states before and after executing each level-1 action are also given as part of the answer set. In this case, we have the initial state (3), the state (5) at time 1 (after serving Alice), and the state (6) at time 2 (after serving Carol). These three states will be passed downward to the next level as state constraints.

Different from the short plan generation, where CLINGO is called multiple times to elaborate high-level actions one by one, the low-cost plan in level 2 is computed by calling CLINGO just once. To reduce the search space in planning at level 2, states (5) and (6) are added to the planning query as state constraints. To do this, their time stamps will be shifted to reflect their places in a longer plan generated at this level. CLINGO

Algorithm 2 Generating Low-cost Plan

Require: domain description at level i $D_i, i \in \{1, \ldots, N\}$
Require: planning query Q and user-specified parameters $E = \{e_i : i = 1, \ldots, M\}$
1: call CLINGO to generate the low-cost answer set A_1 using D_1, Q and $B_1 = e_1$.
2: **for** level i, $i \in \{2, \ldots, N\}$ **do**
3: extract from A_{i-1} the sequence of states $S_{i-1} = (0 : s_0, \ldots, k : s_k)$ and plan $P = (0 : a_0, \ldots, k-1 : a_{k-1})$
4: **for** each state $j : s_j \in S_{i-1}$ **do**
5: compute shifted time stamp j' using P and E
6: add $j' : s_j$ into planning query Q
7: **end for**
8: compute upper-bound B_i using P and E
9: call CLINGO to generate the low-cost answer set A_i using D_i, Q, and B_i
10: **if** i is N **then**
11: **return** the plan extracted from A_N
12: **end if**
13: **end for**

then solves this planning query with intermediate state requirements using upper-bound $= B_2$ computed as $2 \times 5 = 10$.

The shifted time stamp is computed as follows. Let t be the time stamp of a state generated in level 1, l_{serve} denotes the number of actions *serve* before that state, then the shifted time stamp t' is

$$t' = l_{\mathit{serve}} \times e_2 + t - l_{\mathit{serve}}.$$

In the case of state (5), its timestamp shifts from 1 to 5, and it is added as part of the planning query for level 2 as follows:

$$\begin{aligned} &5 : \mathit{mailcollected}(\mathit{alice}), 5 : \mathit{mailcollected}(\mathit{bob}), \\ &5 : \sim \mathit{mailcollected}(\mathit{carol}), 5 : \mathit{mailcollected}(\mathit{daniel}), 5 : \mathit{loc} = o_1, \end{aligned} \tag{14}$$

Similarly, the state (6) is shifted from 2 to 10 and the following conditions are added into the query:

$$\begin{aligned} &10 : \mathit{mailcollected}(\mathit{alice}), 10 : \mathit{mailcollected}(\mathit{bob}), \\ &10 : \mathit{mailcollected}(\mathit{carol}), 10 : \mathit{mailcollected}(\mathit{daniel}), 10 : \mathit{loc} = o_3, \end{aligned} \tag{15}$$

Altogether with the initial state (3), using the optimization statement for action *cross(D)*, `#minimize{X,Y:costcross(X,Y)}`, CLINGO generates a low-cost plan that satisfies initial condition (3), intermediate state (14) and goal state (15):

$$\begin{aligned} &0 : \mathit{cross}(d_5), 1 : \mathit{cross}(d_1), 2 : \mathit{collectmail}(\mathit{alice}), 5 : \mathit{cross}(d_1) \\ &6 : \mathit{cross}(d_3), 7 : \mathit{collectmail}(\mathit{carol}). \end{aligned} \tag{16}$$

Note that the above plan exactly follows the order of serving people: *Alice* is served first, and *Carol* second, which corresponds to the plan generated at level 1. Furthermore, in this plan (16), d_5 rather than d_6 or d_7 is chosen while leaving lab_1 to visiting Alice's office, as the cost of navigating to Alice's office through d_5 is lower due to a shorter

navigation distance. There are some time instances where no actions are executed, such as time instances 3, 4, 8, and 9. These no-operations occur as the upper-bound specified for level 2 is loose. The true length of the above plan is 6.

Plan (16) can be further elaborated at level 3 in a similar way. From (16), we observe that $l_2 = 6$, $l_{cross} = 4$, so according to (13), the upper-bound B_3 for level 3 is 14. The complete algorithm that describes this low-cost plan generation procedure is given in Algorithm 2, where we assume the planning query is satisfiable.

5 Experiments

Experiments were designed to compare our proposed approach that has multiple levels of domain descriptions with a baseline approach that directly describes the domain without high-level abstractions. The latter formalization corresponds to the domain description used in previous work [13], and is equivalent to the domain description at level 3. We compare the performances of these two approaches both for generating short plans and for generating low-cost plans.

The framework described in this paper has been implemented using the newly released state-of-the-art answer set solver CLINGO 4.3. CLINGO 4.3 complements ASP's declarative input language by offering additional control through an embedding scripting language (Python or Lua), such that the complex reasoning process is achieved within a single integrated ASP grounding and solving script. In our experiments, CLINGO 4.3 is run with 6 parallel threads on a desktop machine with 11GB of memory and 2 Intel Xeon CPUs (6 cores each at 2.67GHz).

5.1 Generating Short Plans

To test short plan generation, we used the simulation domain used in previous work [13]. This domain consists of 10 people, 20 rooms and 25 doors inside a building.

In our experiments, mail needs to be collected from everyone inside the building. However we vary how mail is passed from one person to another such that the number of people that need to be visited to collect all the mail varies from 1 to 10. The length of the plan is proportional to the number of people that need to be visited.

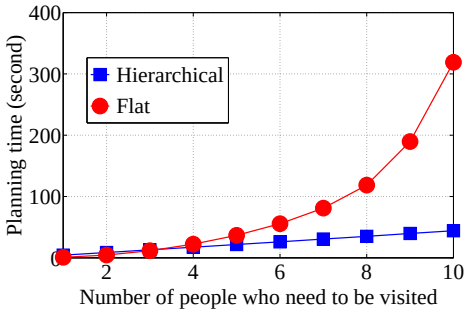

Fig. 2: Time needed to calculate the short plan using the proposed and baseline approaches.

The planning time is plotted in Figure 2. The two curves indicate that our proposed approach of using hierarchical domain abstraction to generate short plans leads to reduced planning time over the baseline flat approach. More importantly, this figure indicates that with the increase of the domain size and the plan size (in terms of the number of people to be visited), the planning time using hierarchical domain abstraction increases *polynomially*, while the planning time using the baseline formalization increases *exponentially*. We attribute this phenomenon to both a well-designed hierarchy and the inherent hierarchical nature of the domain.

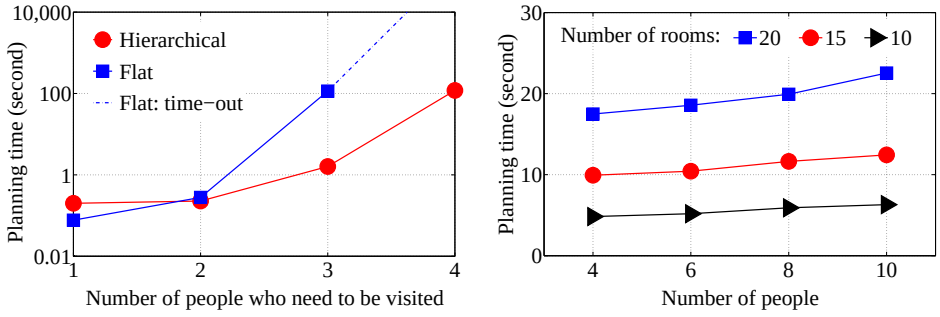

Fig. 3: (a) Time needed to calculate the low-cost plan using the proposed and baseline approaches in logarithmic scale; (b) Time needed to calculate the low-cost plan using the proposed approach in different sized domains.

5.2 Generating Low Cost Plans

We compare the efficiency of generating low-cost plans using the hierarchical and flat approaches. We use the same costs for the *approach* action in both methods. When using hierarchical domain abstraction, we further compile files that contain estimated costs for the *serve* and *cross* actions.

First, we investigate how complex a planning query can be solved by both approaches, and the corresponding planning time for both. The complexity of the planning query can be expressed in terms of the number of people that need to be physically visited to collect everyone's mail. This number directly affects the length of the low-cost plan and also affects the estimation of the upper-bound in the lower levels of the hierarchy. With more people that need to be visited, the number of possible plans to explore will increase exponentially and significantly affect the overall planning time. Figure 1 shows the domain used for evaluating low-cost plan generation.

The robot is initially placed in the corridor and asked to serve everyone. The baseline flat approach needs a upper-bound on plan length under which it will solve for the low-cost plan. We assign this upper-bound as 1.75 times the number of actions in the shortest plan based on our empirical knowledge. It should be noted that 1.75 is a relatively tight upper-bound for the baseline approach, but it makes it easier for the baseline approach to be solved within a predefined timeout of 6 hours. Figure 3(a) shows the time needed to calculate the low-cost plan using the proposed hierarchical approach and the baseline approach without abstraction. While calculating long plans, e.g., to serve more than 2 people, our hierarchical approach reduces the time needed to calculate the low-cost by at least 2 orders of magnitude. For the task of serving four people, the flat approach terminates at the end of 6 hours without verifying that the best plan found so far minimizes the cost. Similar to the case of generating the near-optimal short plan, planning with hierarchical domain abstraction leads to faster planning times. Finally, the plan where the robot individually obtains mail from all 5 people cannot be generated and verified by either approach within 6 hours, indicating the limit of handling complex planning queries. Indeed, when serving 4 people individually, the length of the optimal plan was found to have 31 steps in this experiment.

Finally, we investigate the scalability of our approach to handle reasonably complex planning queries in a large domain based on a real building. We keep the number of people from whom mail needs to be collected fixed at 3, and vary the total number of people in the building from 4 to 10, rooms from 10 to 20 and doors from 14 to 25. Figure 3(b) plots the planning time as the size of the domain increases.

Increasing the number of objects in the domain does not significantly change planning time. In particular, changing the number of people does not have any substantial effect. Adding more rooms and doors has a stronger effect on changing planning times, since it provides more possible routes for navigating from one office to another. Furthermore, the top rightmost point in the figure corresponds to the simulation experiment problem in [13]. Using our new approach, the near-optimal low-cost plan is generated in around 20 seconds. This indicates that for a reasonably complex planning query, our new approach can be applicable in large domains.

6 Conclusions

In this paper we proposed a framework of robotic task planning using action language $\mathcal{BC}$ with hierarchical domain abstraction that addresses efficiency and scalability while generating plans in large robotic domains. We perform a case study where this idea is implemented and evaluated in a mail collection task. Our experiments show that by using hierarchical domain abstraction, solving times for both near-optimal short and low-cost plans have been improved by at least an order of magnitude for reasonably sized domains.

7 Acknowledgments

This research has taken place in the Learning Agents Research Group (LARG) at the AI Laboratory, University of Texas at Austin. LARG research is supported in part by grants from the National Science Foundation (CNS-1330072, CNS-1305287), Office of Naval Research (21C184-01), and Yujin Robot. LARG research is also supported in part through the Freshman Research Initiative (FRI), College of Natural Sciences, University of Texas at Austin.

References

1. Babb, J., Lee, J.: Cplus2ASP: Computing Action Language $\mathcal{C}+$ in Answer Set Programming. In: International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR) (2013)
2. Chen, X., Ji, J., Jiang, J., Jin, G., Wang, F., Xie, J.: Developing High-Level Cognitive Functions For Service Robots. In: International Conference on Autonomous Agents and Multiagent Systems (AAMAS) (2010)
3. Chen, X., Jin, G., Yang, F.: Extending $\mathcal{C}+$ with Composite Actions for Robotic Task Planning. In: International Conference on Logical Programming (ICLP) (2012)
4. Dix, J., Kuter, U., Nau, D.S.: Planning in Answer Set Programming using Ordered Task Decomposition. In: Artëmov, S.N., Barringer, H., d'Avila Garcez, A.S., Lamb, L.C., Woods, J. (eds.) We Will Show Them! (1). College Publications (2005)
5. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: Answer Set Planning Under Action Costs. Journal of Artificial Intelligence Research (JAIR) (2003)

6. Erdem, E., Aker, E., Patoglu, V.: Answer Set Programming for Collaborative Housekeeping Robotics: Representation, Reasoning and Execution. Intelligent Service Robotics (2012)
7. Erol, K., Hendler, J.A., Nau, D.S.: HTN Planning: Complexity and Expressivity. In: National Conference on Artificial Intelligence (AAAI) (1994)
8. Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in *gringo* series 3. In: International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR) (2011)
9. Gelfond, M., Lifschitz, V.: The Stable Model Semantics For Logic Programming. In: International Logic Programming Conference and Symposium (ICLP/SLP) (1988)
10. Gelfond, M., Lifschitz, V.: Classical Negation In Logic Programs and Disjunctive Databases. New Generation Computing (1991)
11. Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic Causal Theories. Artificial Intelligence (AIJ) (2004)
12. Inclezan, D., Gelfond, M.: Representing Biological Processes in Modular Action Language ALM. In: AAAI Spring Symposium on Formalizing Commonsense (2011)
13. Khandelwal, P., Yang, F., Leonetti, M., Lifschitz, V., Stone, P.: Planning in Action Language $\mathcal{BC}$ while Learning Action Costs for Mobile Robots. In: International Conference on Automated Planning and Scheduling (ICAPS) (2014)
14. Knoblock, C.A.: Automatically Generating Abstractions For Planning. Artificial Intelligence 68(2) (1994)
15. Korf, R.E.: Planning as Search: A Quantitative Approach. Artificial Intelligence (AIJ) (1987)
16. Lee, J., Lifschitz, V., Yang, F.: Action Language $\mathcal{BC}$: A Preliminary Report. In: International Joint Conference on Artificial Intelligence (IJCAI) (2013)
17. Levesque, H.J., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.B.: GOLOG: A Logic Programming Language for Dynamic Domains. Logic Programming (JLP) (1997)
18. Marek, V., Truszczynski, M.: Stable Models and An Alternative Logic Programming Paradigm. In: The Logic Programming Paradigm: a 25-Year Perspective. Springer Verlag (1999)
19. McCarthy, J., Hayes, P.: Some Philosophical Problems From The Standpoint of Artificial Intelligence. In: Machine Intelligence. Edinburgh University Press (1969)
20. McIlraith, S.A., Fadel, R.: Planning with Complex Actions. In: International Workshop on Non-Monotonic Reasoning (NMR) (2002)
21. Niemelä, I.: Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. Annals of Mathematics and Artificial Intelligence (1999)
22. Reiter, R.: On Closed World Data Bases. In: Logic and Data Bases. Plenum Press (1978)
23. Sacerdoti, E.D.: Planning In a Hierarchy of Abstraction Spaces. Artificial intelligence 5(2) (1974)
24. Son, T.C., Lobo, J.: Reasoning about policies using logic programs. In: AAAI Spring Symposium on Answer Set Programming (2001)
25. Zhang, S., Sridharan, M., Gelfond, M., Wyatt, J.: An Architecture for Knowledge Representation and Reasoning in Robotics. In: International Workshop on Non-Monotonic Reasoning (NMR) (2014)
26. Zhang, S., Sridharan, M., Washington, C.: Active Visual Planning for Mobile Robot Teams Using Hierarchical POMDPs. IEEE Transactions on Robotics 29(4) (2013)
27. Zhuo, H.H., Muoz-Avila, H., Yang, Q.: Learning Hierarchical Task Network Domains From Partially Observed Plan Traces. Artificial Intelligence (2014)