Discovering User-Interpretable Capabilities of Black-Box Planning Agents

Pulkit Verma, Shashank Rao Marpally, and **Siddharth Srivastava**
Autonomous Agents and Intelligent Robots Lab,
School of Computing and Augmented Intelligence, Arizona State University, AZ, USA
{verma.pulkit, smarpall, siddharths}@asu.edu

Abstract

Several approaches have been developed for answering users' specific questions about AI behavior and for assessing their core functionality in terms of primitive executable actions. However, the problem of summarizing an AI agent's broad capabilities for a user has received little research attention. This is aggravated by the fact that users may not know which questions to ask in order to understand the limits and capabilities of a system. This paper presents an algorithm for discovering from scratch the suite of high-level "capabilities" that an AI system with arbitrary internal planning algorithms/policies can perform. It computes conditions describing the applicability and effects of these capabilities in user-interpretable terms. Starting from a set of user-interpretable state properties, an AI agent, and a simulator that the agent can interact with, using arbitrary decision-making paradigms over primitive operations (unknown to the user), our algorithm returns a set of high-level capabilities with capability descriptions in the user's vocabulary. Empirical evaluation on several game-based scenarios shows that this approach efficiently learns interpretable descriptions of various types of AI agents in deterministic, fully observable settings. User studies show that such interpretable descriptions are easier to understand and reason with than the agent's primitive actions.

1 Introduction

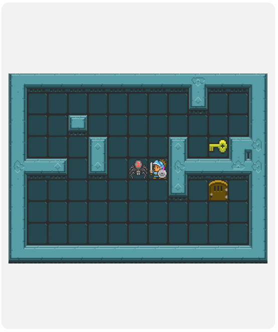
AI systems are rapidly developing to an extent where they can be expected to be used by non-experts who may not understand how they work or what they can and cannot do. Ongoing research on the topic focuses on the significant problem of how to answer such a user's questions about the system's behavior (Anjomshoae et al. 2019; Barredo Arrieta et al. 2020; Chakraborti et al. 2017; Dhurandhar et al. 2018). However, most non-experts hesitate to ask questions about new AI tools (Mou et al. 2017) and quite often do not know which questions to ask in order to assess the safe limits and capabilities of an AI system. This problem is aggravated in situations where an AI system can carry out planning or sequential decision making and the user's conceptual vocabulary may not be rich enough to express simulator-based models of AI systems and their solution policies. Lack of understanding about the limits of an imperfect system can result in unproductive usage or, in the worst-case, serious accidents (Randazzo 2018). This, in turn, limits the adoption and productivity of the AI systems.

This work presents a new approach for learning user-interpretable expressions of a black-box AI system's capabilities. The AI system may use arbitrary internal models, representations, and processes for computing solutions to user-assigned tasks. This paradigm captures a wide variety of AI agents, including ATARI-game playing agents that may use a deep-learned Q function for selecting actions, as well as agents that carry out variants of automated planning. It addresses two main challenges that make it difficult for a user to assess the limits and capabilities of an AI system: (1) the scale mismatch problem, as discussed above and (2) the need for describing an agent's "capabilities", that are defined by the suite of AI algorithms that it uses for behavior synthesis, rather than its core functionality as represented by its primitive, executable actions.

Prior work on the topic addresses complementary problems of deriving symbolic descriptions for pre-defined skills (Konidaris et al. 2018) and of learning users' conceptual vocabularies (Kim et al. 2018; Sreedharan et al. 2022). However, they do not address the problem of discovering high-level user-interpretable capabilities that an agent can perform using arbitrary, internal behavior synthesis algorithms without a specification of the skills that need to be described. Verma et al. (2021) learn user-interpretable models of the AI system using query-answering, but they specifically assume that problems (1) and (2) do not hold: they assume that the user's vocabulary can discern between any two states, and develop methods for learning descriptions of the agent's *primitive* actions rather than the capabilities resulting from its planning/learning algorithms. A greater discussion of related work is presented in Sec. 5.

As a starting point, we assume determinism and full observability on part of the AI system. Since there are no solution approaches for solving the problem even in this foundational setting, our framework can serve as a foundation that can be extended to the more general setting in the future.

Running example Consider a game based on "The Legend of Zelda" (Fig. 1). This game features a protagonist *Link* who must defeat the antagonist *Ganon*, and escape through the door using a key. (a) is the game state as the agent sees it, and its primitive actions are keystrokes (b). But these keystrokes are meaningless to the user as (i) they are too fine-grained and (ii) they showcase only the raw functionality of a game-playing agent–its true capabilities depend on the AI planning



(a)

State as available to the agent:
```
(pixel 1-1 #FD2310)
(pixel 1-1 #B24319)
.
.
```
Agent actions (keystrokes):
```
W, A, S, D, E
```

(b)

State in user's vocabulary:
```
(at ganon 5-3)
(at key 9-4)
(alive ganon)
.
.
```
Desired:
Capabilities such as:
```
(Go next to Key), (Go next
to Door), (Defeat Ganon),
(Pick Key), (Go next to
Ganon), (Open Door)
```

(c)

```
(:capability a9
 :parameters ()
 :precondition
  (and (at link 6-3)
    (at ganon 5-3)
    (alive ganon)
    (next_to ganon))
 :effect
  (and (clear 5-3)
    (not(at ganon 5-3))
    (not(next_to ganon))
    (not(alive ganon))))
```

(d)

Figure 1: (a) A Zelda-like game; (b) States available to the agent and its actions; (c) States represented in user vocabulary, and possible set of desired capabilities; (d) A capability description learned by our method.

and learning algorithms that it uses.

In such situations, the user may want to know what a given game-playing agent can and can't do using its internal algorithms, rather than a listing of the primitive actions. E.g., would it be able to go pick up the key from anywhere in the room? Fig. 1c shows common English terms that a user might understand, and the types of capabilities that they may want to know about.

This paper shows how we can discover and describe an agent's capabilities in a grounded form such as Fig. 1d. This particular capability can be readily transcribed as "If Link is at cell (6-3); Ganon is at cell (5-3), is not defeated, and is next to Link; then Link can act to reach a state where Ganon is defeated, cell (5-3) is empty and Link is not next to Ganon." Our empirical evaluation shows that our system effectively discovers high-level capabilities, and generates grounded descriptions of them; our user study shows that the discovered capabilities help users effectively summarize and analyze the feasibility of different agent behaviors.

The rest of this paper is organized as follows. The next section presents a formal framework for capabilities as well as notions of correctness for discovered agent capabilities. Sec. 3 describes our main algorithms and their formal properties and Sec. 4 presents empirical results and results from user studies. Finally, Sec. 5 discusses the relationship of the presented methods with prior work.

2 Formal Framework

We model an AI system ("agent" henceforth) as a 3-tuple $\langle S, A, T\rangle$, where S is the state space, A is the set of actions that the agent can execute, $T : S \times A \rightarrow S$ is a deterministic black-box transition function determining the effects of the agents primitive actions on the environment. For brevity of notation, we use $a(s)$ to represent $T(s, a)$, where $a \in A$, and $s \in S$. Given a goal set $G \subseteq S$, a black-box *deterministic policy* $\Pi : S \rightarrow A$ maps each state to the action that the agent should execute in that state to reach a $g \in G$.

In this paper, we use "actions" to refer to the core *functionality* of the agent, denoting the primitive actions that the agent could execute (e.g., keystrokes in our running example). In contrast, we use the term "capabilities" to refer to the high-level behaviors that the agent can perform using its AI algorithms for behavior synthesis, including planning and learning (e.g., defeating Ganon or picking up the key). Thus, actions refer to the set of choices that a tabular-rasa agent may possess, while capabilities are a result of its agent function (Russell 1997) and can change as a result of algorithmic updates even as the agent uses the same actions.

2.1 Abstraction

We now define the notion of abstraction used in this work. Several approaches have explored the use of abstraction in planning (Sacerdoti 1974; Giunchiglia et al. 1992; Helmert et al. 2007; Bäckström et al. 2013; Srivastava et al. 2016; Konidaris 2019). We refer to $\tilde{S}$ as the set of *high-level* or *abstract* states, and S as the set of *low-level* or *concrete* states. We define abstraction as in (Srivastava et al. 2016):

Definition 1. Let S and $\tilde{S}$ be sets such that $|\tilde{S}| \leq |S|$. An *abstraction* from S to $\tilde{S}$ is defined by a surjective function $f : S \rightarrow \tilde{S}$. For any $\tilde{s} \in \tilde{S}$, the concretization function $f^{-1}(\tilde{s}) = \{s \in S : f(s) = \tilde{s}\}$ denotes the set of states represented by the abstract state $\tilde{s}$.

Following this notion, we use $\tilde{}$ whenever we refer to a state, a predicate, or an action pertaining to the abstract state space. The next sections use abstraction to formalize the concept of capabilities.

2.2 Capability Descriptions

Our objective is to develop a capability discovery algorithm that learns the capability description of a black-box AI agent using as input (i) the agent; (ii) a compatible simulator using which the agent can simulate its primitive action sequences; and (iii) the user's concept vocabulary, which may be insufficient to express the simulator's state representation. Such assumptions on the agent are common. In fact, use of third-party simulators for development and testing is the bedrock of most of the research on taskable AI systems today (including game playing AI, autonomous cars, and factory robots). Providing simulator access for assessment is reasonable as it would allow AI developers to retain freedom and proprietary controls on internal software while supporting calls for assessment and regulation using approaches like ours.

Several threads of ongoing research address the problem of identifying user-specific concept vocabularies (Kim et al. 2018; Sreedharan et al. 2022), and the field of intelligent tutoring systems develops methods for helping users understand a fixed concept vocabulary. These methods can be used to either elicit or impart a vocabulary for a given user, and such systems can be used to complement the methods developed in this paper. E.g., for a 3-D Blocksworld simulator with objects a and b, and coordinates x, y, and z, "$on(a, b)$ means $z(a) > z(b)$, $x(a) = x(b)$, and $y(a) = y(b)$." As this example illustrates, such vocabularies can be inaccurate.

However, since the problem of capability discovery is not well understood even in settings where user-concept definitions are readily available, we focus on capability discovery with a given vocabulary with known definitions and formalize our approach using them. Furthermore, our empirical evaluation and user studies don't place requirements on user-training or concept acquisition other than the implicit requirement of non-technical English comprehension, which is common to user studies conducted in English. We formalize these concept definitions as follows:

Definition 2. Given a concrete state $s \in S$, a set of concepts/predicates $\tilde{P} = \tilde{p}_1^{k_1}, \dots \tilde{p}_n^{k_n}$ with their arities k_i, a set of object tuples $\tilde{O}$ (of dimension at most k_n), and a Boolean evaluation function $e : S \times \tilde{P} \times \tilde{O} \rightarrow \{T, F\}$, we define $s \models_e \tilde{p}(\tilde{o}_1, \dots, \tilde{o}_n)$ iff $e(s, \tilde{p}, \tilde{o}_1, \dots, \tilde{o}_n) = T$. We define *the abstraction $\tilde{s}_{\tilde{P},\tilde{O}}$ of a state $s \in S$* as the set of all literals over $\tilde{P}$ and $\tilde{O}$ that are true in s. $\tilde{S}_{\tilde{P},\tilde{O}}$ denotes the abstract state space $\{\tilde{s}_{\tilde{P},\tilde{O}} : s \in S\}$.

We omit subscripts $\tilde{P}$ and $\tilde{O}$ unless needed for clarity.

STRIPS-like models (Fikes et al. 1971; McDermott et al. 1998) are natural candidates for formalizing interpretable capability descriptions. This is because when used with a user's vocabulary, such models can be readily transcribed into statements such as "in situations where X holds, if the agent executes actions $a_1, \dots, a_k$ it would result in Y", where X and Y are in the user's vocabulary (Camacho et al. 2019; Verma et al. 2021). In our running example, such a description could indicate that if Link is next to Ganon then Link can defeat it. We express capability descriptions using a STRIPS-like representation.

Definition 3. A *capability description* is a tuple $M = \langle \tilde{P}, \tilde{A}, \tilde{O} \rangle$, where $\tilde{P} = \{\tilde{p}_1, \dots, \tilde{p}_n\}$ is a finite set of predicates; $\tilde{A} = \{\tilde{a}_1, \dots, \tilde{a}_k\}$ is a finite set of capabilities; and $\tilde{O}$ is a finite set of objects. Each capability $\tilde{a} \in \tilde{A}$ is represented as a tuple $\langle pre(\tilde{a}), \mathit{eff}(\tilde{a}) \rangle$, where $pre(\tilde{a})$ and $\mathit{eff}(\tilde{a})$ are sets of literals over $\tilde{P}$ and $\tilde{O}$.

Here each atom could be absent, positive, or negative in the precondition and effects of an action, but an atom cannot be positive (or negative) in both preconditions and effects simultaneously. Semantics of capabilities are close to those of STRIPS actions, but they address vocabulary disparity: an agent can perform a capability $\tilde{a}$ in any concrete state s where $\tilde{s} \models pre(\tilde{a})$; as a result, the system reaches a concrete state s' (a member of an abstract state $\tilde{s}'$). Atoms that don't appear in $\mathit{eff}(\tilde{a})$ retain their truth values from $\tilde{s}$ in $\tilde{s}'$ while others are set to their modes in $\mathit{eff}(\tilde{a})$, i.e., $\forall \ell \in \mathit{eff}(\tilde{a})$, $\tilde{s}' \models \ell$. For brevity, we represent this as $\tilde{s}' = \tilde{a}(\tilde{s})$.

In these terms, the technical problem addressed in this paper is to discover the capability description (Def. 3) for an agent as defined at the start of this section. The next section formalizes notions of correctness for assessing discovered capabilities and algorithms for capability discovery.

2.3 Correctness of Discovered Capabilities

We collect execution traces to discover an agent's capabilities. We define these traces as follows.

Definition 4. An *execution trace* e is a sequence of states of the form $\langle s_0, s_1, \dots, s_{n-1}, s_n \rangle$, such that $\forall i \in [1, n] \quad \exists a_i \in A \; a_i(s_{i-1}) = s_i$.

Since we do not assume that the user's vocabulary is precise enough to discern all states available to the agent, more than one low-level state in an execution trace may be abstracted to a single high-level or abstract state in $\tilde{S}$. E.g., in Fig. 1a, the state available to the agent expresses pixel-level details of the game (Fig. 1b), whereas the user's vocabulary can express it only as an abstract state that represents multiple similar low-level states (Fig. 1c). We define the following notion of consistency to assess the correctness of our learned capability descriptions with available execution traces.

Definition 5. A *capability description $M = \langle \tilde{P}, \tilde{A}, \tilde{O} \rangle$ is consistent with an execution trace* $e = \langle s_0, \dots, s_n \rangle$ iff there exist abstract states $\tilde{s}, \tilde{s}' \in \tilde{S}$ and a transition point $m \in [1, n-1]$ such that $\forall i \leq m \quad \tilde{s}_i = \tilde{s}, \forall i > m \quad \tilde{s}_i = \tilde{s}'$, and $\exists \tilde{a} \in \tilde{A} \; \tilde{s}' = \tilde{a}(\tilde{s})$.

We extend this terminology to say that a capability description is consistent with a set of execution traces E iff it is consistent with every trace in E. We next define the notion of maximal consistency to assess the correctness of a capability description with a set of execution traces E.

Definition 6. Let E be a set of execution traces, and Λ be the set of possible agents that can generate all execution traces in E. A *capability description $M = \langle \tilde{P}, \tilde{A}, \tilde{O} \rangle$ is maximally consistent with a set of execution traces E* iff (i) M is consistent with every trace in E, and (ii) adding any atom as positive or negative precondition or effect of an action in M makes it inconsistent with at least one execution trace that can be generated by at least one agent $\mathcal{A}^* \in \Lambda$.

Finally, we formalize the notion of downward refinability, that the discovered capabilities are indeed within the agent's scope. Recall the notion of abstract state spaces (Def. 2).

Definition 7. Let $M = \langle \tilde{P}, \tilde{A}, \tilde{O} \rangle$ be a capability description with $\tilde{S}$, the induced state space over $\tilde{P}, \tilde{O}$, for an agent $\mathcal{A} = \langle S, A, T \rangle$. M is refinable iff $\forall \tilde{a} \in \tilde{A}, \forall \tilde{s} \in \tilde{S}$, if $\tilde{s} \models pre(\tilde{a})$ then $\forall s \in f^{-1}(\tilde{s}) \quad \exists a_1, \dots, a_n : a_n(a_{n-1} \dots (a_1(s)) \dots) \in \tilde{a}(\tilde{s})$.

In these terms, discovered capabilities are more likely to be useful if they are accurate in the sense that they are consistent with execution traces and refinable, i.e., true repre-

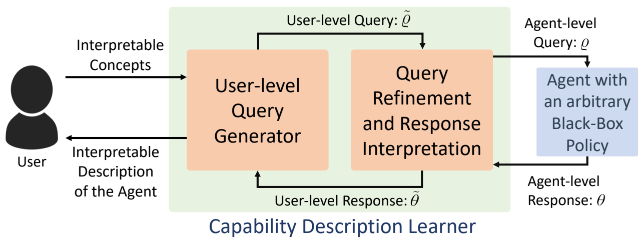

User-level Query: $\tilde{\varrho}$
Interpretable Concepts
User-level Query Generator
Query Refinement and Response Interpretation
Agent-level Query: ϱ
Agent with an arbitrary Black-Box Policy
User
Interpretable Description of the Agent
User-level Response: $\tilde{\theta}$
Agent-level Response: θ
Capability Description Learner

Figure 2: The capability description learner uses a user's preferred vocabulary, generates the agent query, converts it to the low-level query, and based on its responses returns a user-interpretable description of the agent's capabilities.

sentations of what the agent can do. In practice, approximation algorithms could yield discovered capabilities that accommodate real-world uncertainty by providing varying degrees of refinability and consistency.

3 Active Capability Discovery

Fig. 2 illustrates our overall approach. We utilize a query-based paradigm to identify useful execution traces and derive a capability description. The core problem in capability discovery is to effectively abstract the agent's functionality as witnessed through execution traces in a simulator to capabilities that can be described in the user's vocabulary. We use Agent Interrogation Algorithm (AIA), by Verma et al. (2021) to generate potentially useful execution traces, but AIA does not address these core problems of capability discovery. We address the gap by developing a hierarchical query-answering paradigm where queries are initially generated in the user's vocabulary and then translated into primitive-state/action based low-level queries. Finally, the execution trace generated as a result of the query is abstracted back into a user-level response and assimilated into the capability description being computed. Formally, a *user-level query* is a function that maps an agent to a response.

Definition 8. A *user-level query* $\tilde{\varrho}\langle\tilde{s}_I, \tilde{\pi}\rangle : \mathcal{A} \rightarrow N \times \tilde{S}$ is parameterized by a start state $\tilde{s}_I$ and a plan $\tilde{\pi} = \langle\tilde{a}_1, \ldots, \tilde{a}_N\rangle$. It maps agents to responses $\theta = \langle n_\theta, \tilde{s}_\theta\rangle$ such that n_θ is the length of the longest prefix of π that $\mathcal{A}$ can execute and the result of that execution is $\tilde{s}_\theta$.

The input to the capability discovery algorithm (Alg. 1) includes user interpretable predicates $\tilde{P}$ and the agent $\mathcal{A}$.

Generating execution traces As a first step, Alg. 1 collects a set of low-level execution traces E (line 1). These traces are obtained by giving $\mathcal{A}$ a set of tasks of the form $\langle s_I, s_G\rangle$, where $s_I, s_G \in S$, and asking it to reach s_G from s_I. The actions it takes to complete the task and the intermediate states form the set of execution traces E. Partial capability descriptions $\tilde{A}$ in the user's vocabulary are generated from E (line 2) (see Sec. 3.1).

Comparing abstract models The algorithm iteratively determines how each high-level predicate $\tilde{P}$ appears in a capability $\tilde{A}$, in the precondition and effect (line 4). To achieve this, it generates three abstract capability descriptions each

Algorithm 1: Capability Discovery Algorithm

```
Require: predicates $\tilde{P}$, agent $\mathcal{A}$
 1: $E \leftarrow generate\_execution\_traces(\mathcal{A})$
 2: $\tilde{A} \leftarrow generate\_partial\_capability\_descriptions(E)$
 3: Set $\tilde{M}^* = \phi$, $\tilde{L} \leftarrow \{pre, eff\}$
 4: for each $\langle\tilde{L}, \tilde{A}, \tilde{P}\rangle$ do
 5:     Generate $M_+, M_-, M_\emptyset$ by setting $\tilde{P}$ in $\tilde{A}$ at $\tilde{L}$ to
        $+, -$, and $\emptyset$ in $\tilde{M}^*$
 6:     for each pair $M_1, M_2$ in $\{M_+, M_-, M_\emptyset\}$ do
 7:         $\tilde{\varrho} \leftarrow generate\_query(M_1, M_2)$
            //$\tilde{\varrho}$ is of the form $\langle\tilde{s}_0, \tilde{a}_1, \tilde{a}_2, \ldots, \tilde{a}_k\rangle$
 8:         $s_0 \leftarrow refine\_state(\tilde{s}_0)$                    } Query Refinement
 9:         for $i$ in range $[1, k]$ do
10:             Set $\tilde{s}_i$ by applying $\tilde{a}_i$ in state $\tilde{s}_{i-1}$
11:             $s_i \leftarrow$ concretize state $\tilde{s}_i$
12:         end for
13:         for $i$ in range $[0, k-1]$ do
14:             $\varrho \leftarrow \langle s_i, s_{i+1}\rangle$
15:             $\theta \leftarrow ask\_agent(\varrho, \mathcal{A})$          } Response Interpretation
16:             if $\theta = \perp$ then
17:                 $\tilde{\theta} \leftarrow \langle i, \tilde{s}_i\rangle$
18:             end if
19:         end for
20:         $\tilde{\theta} \leftarrow \langle k, \tilde{s}_k\rangle$
21:         $\tilde{M}^* \leftarrow consistent\_description(\tilde{\theta}, M_1, M_2)$
22:     end for
23: end for
24: return $\tilde{M}^*$
```

representing that $\tilde{P}$ can be a positive or negative precondition (or effect) of $\tilde{A}$, or can be absent in $\tilde{A}$. Then the algorithm picks two of these descriptions at a time (line 6) and generates a user-level query $\tilde{\varrho}$ that can distinguish between the two descriptions similar to the query generation of AIA (line 7). Such queries cannot be posed directly to the agent, hence they are converted to one or more agent-level queries using the query refinement process (lines 8-14). Then the algorithm uses a response interpretation process to collect the responses of all agent-level queries that correspond to the same user-level query, and process them to generate a single user-level response $\langle n_\theta, \tilde{s}_\theta\rangle$ (lines 15-20) (See Sec. 3.2).

Learning capability descriptions Alg. 1 finds a capability description that is consistent with $\mathcal{A}$'s response (line 21). This process includes asking the same user-level query to the two abstract capability descriptions and checking which description's response is consistent with $\mathcal{A}$'s response. This process is repeated until Alg. 1 finds how each predicate in $\tilde{P}$ appears in precondition and effect of each capability in $\tilde{A}$.

3.1 Generating Partial Capability Descriptions

This component corresponds to line 2 of Alg. 1. Given an execution trace $e = \langle s_0, s_1, \ldots, s_n\rangle \in E$, whenever $\tilde{s}_i \neq \tilde{s}_{i+1}$, we store the transition as a possible new capability $\tilde{a}_{\tilde{s}_i - \tilde{s}_{i+1}}$. It is possible that multiple low level states map to a single state, i.e., $\tilde{s}_i = \tilde{s}_{i+1}$. Hence, the abstractions we use in this work are a special case of forall-exists abstraction (Srivastava et al. 2016) defined as:

Definition 9. An abstraction is a *forall-exists abstraction* iff $\forall \tilde{s}' \in \tilde{a}(\tilde{s}), \forall s \in \tilde{s}, \exists s' \in a(s)$ such that $s' \in \tilde{s}'$.

The only difference we have as compared to this is that high-level capabilities $\tilde{a} \in \tilde{A}$ that we learn are deterministic. For each new possible capability $\tilde{a}_{\tilde{s}_i - \tilde{s}_{i+1}}$, the states before and after executing these capabilities are stored as possible sets of preconditions and effects. We then combine sets of possible capabilities that cause similar state transitions. In a manner similar to prior work by Stern et al. (2017), for each of these sets, we create a possible set of preconditions by taking the intersection of the predicates that were true in the states where these capabilites were executed. Similarly, we create possible effects using the states after the capabilities were executed. This gives us partial capability descriptions of these high-level capabilities. Further queries in line 7 are used to complete the descriptions of these capabilities.

3.2 Query Refinement and Response Interpretation Process

Query refinement corresponds to lines 8-14 of Alg. 1. It converts a user-level query $\tilde{\varrho}$ to a set of agent-level queries that are posed to the agent. A user-level query $\tilde{\varrho} = \langle \tilde{s}_I, \tilde{\pi} \rangle$ can be represented as $\langle \tilde{s}_0, \tilde{a}_1, \tilde{a}_2, \ldots, \tilde{a}_N \rangle$, where the initial state $\tilde{s}_I = \tilde{s}_0$ and plan $\tilde{\pi} = \langle \tilde{a}_1, \tilde{a}_2, \ldots, \tilde{a}_N \rangle$. The first step is to convert the trace to the form $\langle \tilde{s}_0, \tilde{a}_1, \tilde{s}_1, \tilde{a}_2, \tilde{s}_2 \ldots, \tilde{a}_k, \tilde{s}_N \rangle$ using the partial capability descriptions $\tilde{A}$ learned earlier from the execution traces (line 2). Each of the high-level states are then concretized to the low-level states, and the consecutive low-level states are paired in the form $\langle s_i, s_{i+1} \rangle$, where i ranges from 0 to $N-1$. These pairs are directly used as agent-level queries posed to the agent sequentially.

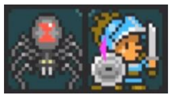
Response Interpretation corresponds to lines 15-20 of Alg. 1. The pairs of states $\langle s_i, s_{i+1} \rangle$ are given sequentially to the agent and the agent responds true (or false) if it can (or cannot) reach from state s_i to s_{i+1} using its internal policy. If the agent responds true for all such pairs, then it shows that the agent can execute the high-level plan $\tilde{\pi}$ successfully, and the final high-level state along with the plan length is set as a response $\tilde{\theta}$ (line 20). However, if the agent fails to reach a state s_{i+1} from the state s_i, then it is treated as a failure to execute the capability $\tilde{a}_i$ in state $\tilde{s}_i$, and the response $\tilde{\theta}$ is set as $\langle i, \tilde{s}_i \rangle$ (line 17). This also results in updating the partial description of the failing high-level capability in line 21.

The following lemma formalizes the property that the response interpretation correctly converts the agent-level responses to the user-level responses.

Lemma 1. Given that the agent responses to the agent-level queries are correct, the response $\tilde{\theta}$ to the user-level query generated by Alg. 1 is always correct.

Desirable properties of Alg. 1 The following section formalizes the properties that the learned capability description must satisfy for correctness.

Theorem 1. Given the predicates $\tilde{P}$ in the user vocabulary and the set of low-level execution traces E generated in stationary fully observable settings, the capability description $M = \langle \tilde{P}, \tilde{A}, \tilde{O} \rangle$ of an agent $\mathcal{A}$ with deterministic black-box policy learned by the capability discovery algorithm (Alg. 1) is consistent with E and is refinable.


(a)

(b)

```
If Link  is at cell (6-3),
Ganon is at cell (5-3), is
not defeated,  and is next
to Link, then Link can act
to  reach  a  state  where
Ganon   is   defeated,  cell
(5-3) is empty and Link is
not  next to Ganon.
```
(c)

Keystrokes 1: `W` $\rightarrow$ `A` $\rightarrow$ `E`
Keystrokes 2: `S` $\rightarrow$ `S` $\rightarrow$ `A` $\rightarrow$ `W` $\rightarrow$ `W` $\rightarrow$ `A` $\rightarrow$ `E`

(d)

Figure 3: Learning the *defeat monster* capability of Zelda-like game. Sub-figures (a) and (b) show the states available to the agent immediately before and after executing either of the keystroke sequences shown in (d). Sub-figure (c) shows a boilerplate readable description that can be generated from the learned description for a capability that we might understand as *Defeat Ganon* in Fig. 1(d).

Theorem 2. Let $\mathcal{A} = \langle S, A, T \rangle$ be an agent operating in a deterministic, fully observable environment with a state space S using a set of primitive actions A. Given an input vocabulary $\tilde{P}$, the set of execution traces E generated by $\mathcal{A}$, and the set of possible agents Λ that can generate all execution traces in E, the set of capabilities maintained in Alg. 1 is *maximally consistent* with the set of execution traces E.

4 Empirical Evaluation

We implemented and evaluated Alg. 1 with two broad categories of input test agents: *Policy agents* can use (possibly learned) black-box policies to plan and to respond to agent-level queries. We used policy agents with hand-coded policies for this evaluation. *Search agents* respond to agent-level queries using arbitrary search algorithms. We used search agents that use A* search. We now describe the setup of our experiments used for evaluation.

4.1 Experimental Setup

Our test agents use the General Video Game Artificial Intelligence (GVGAI) framework (Perez-Liebana et al. 2016, 2019). We performed experiments on four domains – Zelda, Cook-Me-Pasta, Escape, and Snowman. Details about these domains and the user vocabularies are available in the appendix. Since the complete list of an agent's capabilities may be irrelevant to a user's current needs, w.l.o.g, our implementation supports an input including sets of formulas representing the properties that may be of interest to the user. This set can be the set of all grounded predicates in the user's concept vocabulary. We then query the agent in a way that yields capability descriptions relevant for achieving these properties.

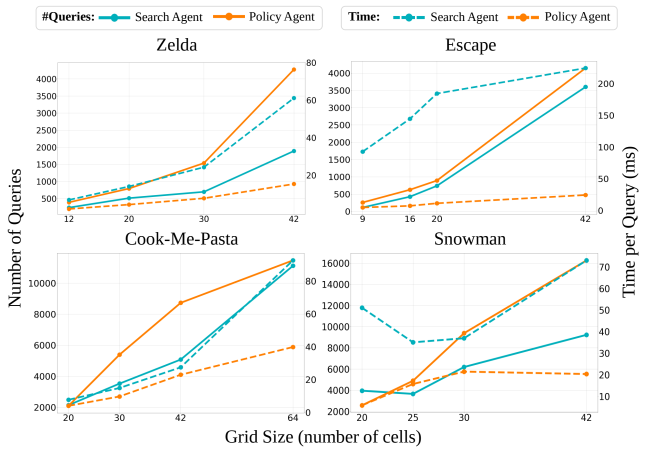


Figure 4: Performance comparison of search-based and policy-based agents in terms of the number of queries asked and time taken per query when increasing the grid size.

For each domain, and for each grid size in that domain, we create a random game instance with the goal of achieving one of the user's specified properties of interest. The number of obstacles in all domains, except Escape, is set to 20% of the total cells in the grid, whereas all other objects are generated randomly. In Escape domain, we set the number of holes, walls, and movable blocks to 10%, 20%, and 20% of the total cells respectively. We use the solution to that instance to generate the execution trace that is used in lines 1-2 of Alg. 1. These solutions are not always optimal. All experiments are run on 5.0 GHz Intel i9 CPUs with 64 GB RAM running Ubuntu 18.04.

4.2 Empirical Results

As shown in Sec. 3, our algorithm is guaranteed to compute capability descriptions that are correct in the sense that they are consistent with the execution traces, and refinable and executable with respect to the true capabilities of the agent. Fig. 1d shows an example of a learned capability description from the Zelda-like domain, corresponding to the *Defeat Ganon* capability shown in Fig. 3. We now present the main conclusions of our empirical analysis.

We evaluated our algorithm's performance along two aspects; (i) how the performance of our approach changes with respect to the size of the problem; and (ii) how its performance differs for search-based vs policy-based agents.

Scalability analysis We increase the size of each domain to analyze its effect on the performance of the search and policy agents. Fig. 4 shows the graphs for the experimental runs on the four domains. In all four domains, for both kinds of agents, *the number of queries increase as we increase the grid size*. This happens because the queries are grounded, and the number of possible groundings of the predicates increases as we increase the grid size. The increasing number of queries is expected behavior and this is also clear in approaches that use passive observations of agent behavior (Yang et al. 2007; Aineto et al. 2019; Bonet et al. 2020).

Agent type analysis The number of queries required by the policy agent is higher than that of the search agent in almost all cases. This is because a large number of user-level queries fail to run successfully on the agent as the high-level plan in the user-level query does not always align with the policy of the agent. However, the time per query is lesser for the policy agents as they can answer the queries by following their policy, whereas the search agents perform an exhaustive search of the state space for every low-level query.

	S1	S2	S3	S4	S5	S6	S7	S8
C1	1.0	0	0	0	0	0	0	0
C2	0	1.0	0	0	0	0	0	0
C3	0	0	0.94	0	0	0	0.06	0
C4	0	0	0	1.0	0	0	0	0
C5	0	0	0	0	0.94	0	0	0.06
C6	0	0	0	0	0	1.00	0	0

Table 1: Accuracy of capability summarization study for the Zelda-like game. An element in row Ci and column Sj represents the fraction of instances when capability Ci was summarized as Sj by the study participants. Correct summarization of Ci is Si (in green). C1,S1: *Go next to Ganon*; C2,S2: *Go next to Key*; C3,S3: *Go next to Door*; C4,S4: *Defeat Ganon*; C5,S5: *Pick Key*; C6,S6: *Open Door*; S7: *Go next to Wall*; S8: *Break Key*.

4.3 User Study

We conducted a user study to evaluate interpretablity of the capability descriptions computed by Alg. 1. Intuitively, our notion of interpretability matches that of common English and its use in AI literature, e.g., as enunciated by Doshi-Velez et al. (2018): *"the ability to explain or to present in understandable terms to a human"*. We evaluate this through two operational hypotheses:

H1. The user can effectively summarize the learned capability descriptions.

H2. The discovered capabilities make it easier for users to analyze and predict the outcome of the agent's possible behaviors.

We designed two studies to evaluate these hypotheses.

Capability summarization study This study evaluates the interpretability of the discovered capability descriptions. The user is explained the rules of the Zelda-like game described earlier, and then presented with a text description of the six learned capabilities. Finally, the user is asked to choose a short summarization for each description, out of the eight possible summarizations that we provide.

Behavior analysis study This study compares the predictability and analyzability of agent behavior in terms of the agent's low-level actions and high-level capabilities. Each user is explained the rules of Zelda-like game. One set of users are presented with text descriptions of the agent's primitive actions, while others are asked to complete the summarization study. Then each user is given same 5 questions in order. Each question contains two game state images; start and end state. The user is asked what sequence of actions should the agent take to reach the end state from the start state. Each question has 5 possible options for the

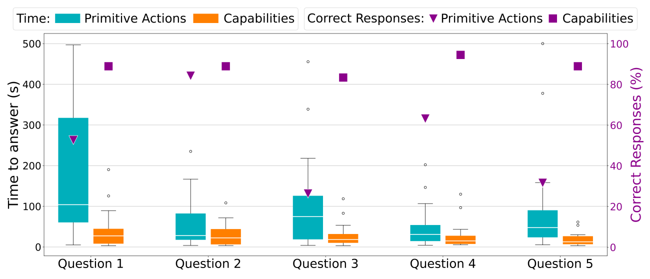


Figure 5: Data from behavior analysis shows that using computed capability descriptions took lesser time and yielded more accurate results. See Sec. 4.3 for details.

user to choose from, and these options differ depending on their group. We then collect the data about the accuracy of the answers, and the time taken to answer each question.

Study design A total of 50 participants were recruited from Amazon Mechanical Turk and randomly divided into two groups of 25 each. Each user in one group (capability group) was asked to complete the summarization study followed by the behavior analysis study with capability descriptions, whereas each user from another group (primitive action group) was asked to complete the behavior analysis study with primitive actions. We used screeners (Arndt et al. 2021; Kennedy et al. 2020) to ensure quality of the data collected, and discarded 13 responses. The results are based on the responses of 19 and 18 users in primitive action and capability group, respectively.

Results The results of the capability summarization study (Tab. 1) demonstrate that the users are able to summarize the descriptions almost uniformly accurately except for C3 and C5. This verifies H1 that the users can effectively summarize the learned capability descriptions. The results for the behavior analysis study (Fig. 5) indicate that the users took less time to answer questions and they got more responses correct when using the capabilities as compared to using primitive actions. This validates H2 that the discovered capabilities made it easier for the users to analyze and predict the agent's behavior correctly.

5 Related Work

High-level skills from input options Given a set of options encoding skills as input, Konidaris et al. (2018) and James et al. (2020) propose methods for learning high-level propositional models of options representing various "skills." They assume access to predefined options and learn the high-level symbols that describe those options at the high-level. While they use options or skills as inputs to learn models defining when those skills will be useful in terms of auto-generated symbols (for which explanatory semantics could be derived in a post-hoc fashion), our approach uses user-provided interpretable concepts as a priori inputs to learn agent capabilities: high-level actions as well as their interpretable descriptions in terms of the input vocabulary.

Learning symbolic models using physics simulators Zhang et al. (2018) learn symbolic transition models and use them effectively for planning using a set of input interpretable attributes. Some approaches learn different kinds of symbolic models of the functionality of ATARI or physics based simulators using methods like conjunctions of binary input features (Kansky et al. 2017), graph neural networks (Battaglia et al. 2016; Cranmer et al. 2020), CNNs (Agrawal et al. 2016; Fragkiadaki et al. 2016), etc. Some methods create interpretable descriptions of reinforcement learning policies using trees (Liu et al. 2018) or specialized programming languages (Verma et al. 2018). These approaches solve the orthogonal problem of learning the functionality of an agent that could help an user understand how an agent would solve a problem, whereas we focus on learning capabilities of the agent that could help a user understand and answer what type of problems it could solve.

Action model learning The planning community has also worked on learning STRIPS-like action models of agents from observations of its behavior (Gil 1994; Yang et al. 2007; Cresswell et al. 2009; Zhuo et al. 2013; Aineto et al. 2019). Jiménez et al. (2012) and Arora et al. (2018) present a comprehensive review of such approaches. These methods work with broad assumptions that the agent model is internally expressed in the same vocabulary as the user's (Gil 1994; Weber et al. 2011), or at a similar level of abstraction (Mehta et al. 2011; Verma et al. 2021). Our approach is able to learn the capability descriptions in terms of fewer concepts than used by the agent.

Concept acquisition There is also a lot of ongoing and existing work on the orthogonal problem of learning predicates or concepts from user-provided examples or feedback (Amershi et al. 2009; Kim et al. 2015; Koh et al. 2017; Kim et al. 2018; Sreedharan et al. 2022; Lage et al. 2020) – this complements our research and can be used with our methods. Additionally, resolving the problem of obtaining user-interpretable predicates does not resolve the research problem that we focus on: deriving high-level descriptions of agent capabilities using those predicates.

6 Conclusion

We presented a novel approach for learning the grounded capability description of an AI system in terms of user-interpretable concepts by combining information from passive execution traces and active query answering. Our approach works for settings where the user's conceptual vocabulary is imprecise and cannot directly express the agent's capabilities. Our empirical analysis showed that for the agents that internally use black-box deterministic policies, or search techniques, we can successfully discover the capabilities and their descriptions. Extending this approach for partially observable settings and relaxing the various assumptions we made are some of the promising future directions for this work.

Acknowledgements

We thank Nancy Cooke, Akkamahadevi Hanni, and Sydney Wallace for their help with the user study. We also thank anonymous reviewers for their helpful feedback on the paper. This work was supported in part by the NSF grant IIS 1844325, and the ONR grant N00014-21-1-2045.

References

Agrawal, P.; Nair, A. V.; Abbeel, P.; Malik, J.; and Levine, S. 2016. Learning to Poke by Poking: Experiential Learning of Intuitive Physics. In *Proc. NIPS*.

Aineto, D.; Celorrio, S. J.; and Onaindia, E. 2019. Learning Action Models With Minimal Observability. *Artificial Intelligence* 275: 104–137.

Amershi, S.; Fogarty, J.; Kapoor, A.; and Tan, D. 2009. Overview Based Example Selection in End User Interactive Concept Learning. In *Proc. UIST*.

Anjomshoae, S.; Najjar, A.; Calvaresi, D.; and Främling, K. 2019. Explainable Agents and Robots: Results from a Systematic Literature Review. In *Proc. AAMAS*.

Arndt, A. D.; Ford, J. B.; Babin, B. J.; and Luong, V. 2021. Collecting Samples from Online Services: How to Use Screeners to Improve Data Quality. *International Journal of Research in Marketing* ISSN 0167-8116.

Arora, A.; Fiorino, H.; Pellier, D.; Métivier, M.; and Pesty, S. 2018. A Review of Learning Planning Action Models. *The Knowledge Engineering Review* 33: E20.

Bäckström, C.; and Jonsson, P. 2013. Bridging the Gap Between Refinement and Heuristics in Abstraction. In *Proc. IJCAI*.

Barredo Arrieta, A.; Díaz-Rodríguez, N.; Del Ser, J.; Bennetot, A.; Tabik, S.; Barbado, A.; Garcia, S.; Gil-Lopez, S.; Molina, D.; Benjamins, R.; Chatila, R.; and Herrera, F. 2020. Explainable Artificial Intelligence (XAI): Concepts, Taxonomies, Opportunities and Challenges toward Responsible AI. *Information Fusion* 58: 82–115.

Battaglia, P.; Pascanu, R.; Lai, M.; Jimenez Rezende, D.; and Kavukcuoglu, K. 2016. Interaction Networks for Learning about Objects, Relations and Physics. In *Proc. NIPS*.

Bonet, B.; and Geffner, H. 2020. Learning First-Order Symbolic Representations for Planning from the Structure of the State Space. In *Proc. ECAI*.

Camacho, A.; and McIlraith, S. A. 2019. Learning Interpretable Models Expressed in Linear Temporal Logic. In *Proc. ICAPS*.

Chakraborti, T.; Sreedharan, S.; Zhang, Y.; and Kambhampati, S. 2017. Plan Explanations as Model Reconciliation: Moving Beyond Explanation as Soliloquy. In *Proc. IJCAI*.

Cranmer, M.; Sanchez Gonzalez, A.; Battaglia, P.; Xu, R.; Cranmer, K.; Spergel, D.; and Ho, S. 2020. Discovering Symbolic Models from Deep Learning with Inductive Biases. In *Proc. NeurIPS*.

Cresswell, S.; McCluskey, T.; and West, M. 2009. Acquisition of Object-Centred Domain Models from Planning Examples. In *Proc. ICAPS*.

Dhurandhar, A.; Chen, P.-Y.; Luss, R.; Tu, C.-C.; Ting, P.; Shanmugam, K.; and Das, P. 2018. Explanations based on the Missing: Towards Contrastive Explanations with Pertinent Negatives. In *Proc. NeurIPS*.

Doshi-Velez, F.; and Kim, B. 2018. *Considerations for Evaluation and Generalization in Interpretable Machine Learning*, 3–17. Springer International Publishing.

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 2(3-4): 189–208.

Fragkiadaki, K.; Agrawal, P.; Levine, S.; and Malik, J. 2016. Learning Visual Predictive Models of Physics for Playing Billiards. In *Proc. ICLR*.

Gil, Y. 1994. Learning by Experimentation: Incremental Refinement of Incomplete Planning Domains. In *Proc. ICML*.

Giunchiglia, F.; and Walsh, T. 1992. A Theory of Abstraction. *Artificial Intelligence* 57(2-3): 323–389.

Helmert, M.; Haslum, P.; Hoffmann, J.; et al. 2007. Flexible Abstraction Heuristics for Optimal Sequential Planning. In *Proc. ICAPS*.

James, S.; Rosman, B.; and Konidaris, G. 2020. Learning Portable Representations for High-Level Planning. In *Proc. ICML*.

Jiménez, S.; De La Rosa, T.; Fernández, S.; Fernández, F.; and Borrajo, D. 2012. A Review of Machine Learning for Automated Planning. *The Knowledge Engineering Review* 27(4): 433–467.

Kansky, K.; Silver, T.; Mély, D. A.; Eldawy, M.; Lázaro-Gredilla, M.; Lou, X.; Dorfman, N.; Sidor, S.; Phoenix, S.; and George, D. 2017. Schema Networks: Zero-shot Transfer with a Generative Causal Model of Intuitive Physics. In *Proc. ICML*.

Kennedy, R.; Clifford, S.; Burleigh, T.; Waggoner, P. D.; Jewell, R.; and Winter, N. J. G. 2020. The Shape of and Solutions to the MTurk Quality Crisis. *Political Science Research and Methods* 8(4): 614–629.

Kim, B.; Shah, J. A.; and Doshi-Velez, F. 2015. Mind the Gap: A Generative Approach to Interpretable Feature Selection and Extraction. In *Proc. NIPS*.

Kim, B.; Wattenberg, M.; Gilmer, J.; Cai, C.; Wexler, J.; Viegas, F.; and Sayres, R. 2018. Interpretability Beyond Feature Attribution: Quantitative Testing with Concept Activation Vectors (TCAV). In *Proc. ICML*.

Koh, P. W.; and Liang, P. 2017. Understanding Black-Box Predictions via Influence Functions. In *Proc. ICML*.

Konidaris, G. 2019. On the Necessity of Abstraction. *Current Opinion in Behavioral Sciences* 29: 1–7.

Konidaris, G.; Kaelbling, L. P.; and Lozano-Perez, T. 2018. From Skills to Symbols: Learning Symbolic Representations for Abstract High-Level Planning. *Journal of Artificial Intelligence Research* 61(1): 215–289.

Lage, I.; and Doshi-Velez, F. 2020. Learning Interpretable Concept-Based Models with Human Feedback. In *ICML 2020 Workshop on Human Interpretability in Machine Learning*.

Liu, G.; Schulte, O.; Zhu, W.; and Li, Q. 2018. Toward Interpretable Deep Reinforcement Learning with Linear Model U-Trees. In *Proc. ECML PKDD*.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D. S.; and Wilkins, D. 1998. PDDL – The Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Mehta, N.; Tadepalli, P.; and Fern, A. 2011. Autonomous Learning of Action Models for Planning. In *Proc. NIPS*.

Mou, Y.; and Xu, K. 2017. The Media Inequality: Comparing the Initial Human-Human and Human-AI Social Interactions. *Computers in Human Behavior* 72: 432–440. ISSN 0747-5632.

Perez-Liebana, D.; Liu, J.; Khalifa, A.; Gaina, R. D.; Togelius, J.; and Lucas, S. M. 2019. General Video Game AI: A Multitrack Framework for Evaluating Agents, Games, and Content Generation Algorithms. *IEEE Transactions on Games* 11(3): 195–214.

Perez-Liebana, D.; Samothrakis, S.; Togelius, J.; Schaul, T.; and Lucas, S. 2016. General Video Game AI: Competition, Challenges and Opportunities. In *Proc. AAAI*.

Randazzo, R. 2018. What went wrong with Uber's Volvo in fatal crash? Experts shocked by technology failure. *The Arizona Republic* .

Russell, S. J. 1997. Rationality and Intelligence. *Artificial Intelligence* 94(1-2): 57–77.

Sacerdoti, E. D. 1974. Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence* 5(2): 115–135.

Sreedharan, S.; Soni, U.; Verma, M.; Srivastava, S.; and Kambhampati, S. 2022. Bridging the Gap: Providing Post-Hoc Symbolic Explanations for Sequential Decision-Making Problems with Inscrutable Representations. In *Proc. ICLR*.

Srivastava, S.; Russell, S.; and Pinto, A. 2016. Metaphysics of Planning Domain Descriptions. In *Proc. AAAI*.

Stern, R.; and Juba, B. 2017. Efficient, Safe, and Probably Approximately Complete Learning of Action Models. In *Proc. IJCAI*.

Verma, A.; Murali, V.; Singh, R.; Kohli, P.; and Chaudhuri, S. 2018. Programmatically Interpretable Reinforcement Learning. In *Proc. ICML*.

Verma, P.; Marpally, S. R.; and Srivastava, S. 2021. Asking the Right Questions: Learning Interpretable Action Models Through Query Answering. In *Proc. AAAI*.

Weber, C.; Morwood, D.; and Bryce, D. 2011. Goal-Directed Knowledge Acquisition. In *ICML 2011 Workshop on Planning and Acting with Uncertain Models*.

Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning Action Models from Plan Examples Using Weighted MAX-SAT. *Artificial Intelligence* 171(2-3): 107–143.

Zhang, A.; Sukhbaatar, S.; Lerer, A.; Szlam, A.; and Fergus, R. 2018. Composable Planning with Attributes. In *Proc. ICML*.

Zhuo, H. H.; and Kambhampati, S. 2013. Action-Model Acquisition from Noisy Plan Traces. In *Proc. IJCAI*.

A Domains and their Semantics

This section describes the four domains and the semantics of the user-interpretable predicates in these domains. Note that information like orientation of the agent (player) in each of these domains is not captured by any of the predicates. This information is important for the low-level policies as certain actions can only be executed in certain orientations.

A.1 Zelda

The Zelda-like domain, as shown in Fig. 1a, consists of a key, a door that opens using that key, the antagonist player *Link*, and the protagonist monster *Ganon*. To win the game, Link must defeat Ganon, and then should use the key to open the door to escape. Link can move one cell at a time in the direction it is facing. If Link moves into the cell adjacent to the key, Link picks up the key by executing the keystroke `E` (special keystroke). The same keystroke is used to Defeat Ganon when Link is facing Ganon and is in a cell adjacent to Ganon, and to escape when Link is in a cell adjacent to the door and facing it. The user provided vocabulary for this domain is shown below:

Predicate	**Meaning**
`at(?ob ?loc)`	True if an object `?ob` is at location `?loc`
`wall(?loc)`	True if there is a wall at location `?loc`
`clear(?loc)`	True if location `?loc` is empty, i.e., it has no object, wall, or player
`has_key()`	True if Link has the key.
`escaped()`	True if Link has escaped (game is over).
`alive(?m)`	True if Ganon is still alive
`next_to_ganon()`	True if Link is in a cell adjacent to Ganon.
`next_to_key()`	True if Link is in a cell adjacent to Key.
`next_to_door()`	True if Link is in a cell adjacent to Door.

A.2 Cook-Me-Pasta

The Cook-Me-Pasta domain, as shown in Fig. 6a, consists of raw pasta, sauce, boiling water, tuna (fish), lock, and key. The objective is to cook tuna pasta using a three step process. First the pasta is cooked by adding boiling water to the raw pasta, this can be done by pressing `E` while holding both the ingredients. Similarly tuna is cooked by mixing sauce and tuna. Finally, the cooked pasta and the cooked tuna are to be mixed together. One or more of the ingredients can be locked in a room which must be opened using a key. The user provided vocabulary for this domain is shown below:

Predicate	**Meaning**
`at(?ob ?loc)`	True if an object `?ob` is at location `?loc`
`wall(?loc)`	True if there is a wall at location `?loc`
`clear(?loc)`	True if location `?loc` is empty, i.e., it has no object, wall, or player
`has_key()`	True if the player has the key
`pasta_cooked()`	True if the pasta is cooked
`is_door(?loc)`	True if the location `?loc` has a door

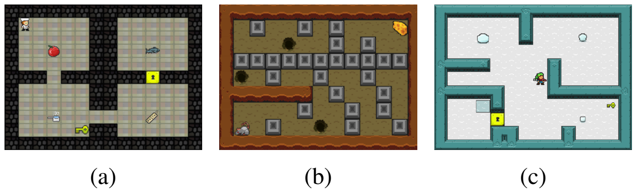


Figure 6: GVGAI's domains; (a) Cook-Me-Pasta, (b) Escape, and (c) Snowman.

A.3 Escape

The Escape domain, as shown in Fig.6b, consists of movable blocks, fixed holes, and cheese. The blocks can be pushed into the holes to clear out a path. The game is finished when the player reaches the location with cheese. The user provided vocabulary for this domain is shown below:

Predicate	**Meaning**
`at(?ob ?loc)`	True if an object `?ob` is at location `?loc`
`wall(?loc)`	True if there is a wall at location `?loc`
`clear(?loc)`	True if location `?loc` is empty, i.e., it has no object, wall, or player
`is_hole(?loc)`	True if the location `?loc` has a hole
`is_goal(?loc)`	True if the location `?loc` is he goal location
`is_block(?loc)`	True if the location `?loc` has a movable block

A.4 Snowman

The Snowman domain, as shown in Fig.6c, consists of three pieces of a snowman: the top, middle, and bottom piece; a key that can be used to unlock a door (like other domains), and the goal cell. The objective of the game is to assemble the snowman in the goal location in order, constrained by the player being able to hold only one piece at any given time.

The user provided vocabulary for this domain is shown below:

Predicate	**Meaning**
`at(?ob ?loc)`	True if an object `?ob` is at location `?loc`
`wall(?loc)`	True if there is a wall at location `?loc`
`clear(?loc)`	True if location `?loc` is empty, i.e., it has no object, wall, or player
`has_key()`	True if the player has the key
`player_has(?ob)`	True if the player has object ?ob
`is_goal(?loc)`	True if the location `?loc` is he goal location
`placed(?part)`	True if part `?part` is placed at the goal location.
`is_door(?loc)`	True if the location `?loc` has a door

B Learned Capability Descriptions

Some sample learned grounded capabilities for the four domains are shown in Fig.7. Note that the name of capabilities are not learned by our approach, but were manually assigned after the capability descriptions were learned.

Zelda

```
(:capability defeat-ganon
  :parameters ()
  :precondition (and
    (at link 6-3)
    (at ganon 5-3)
    (alive ganon)
    (next_to ganon))
  :effect (and
    (clear 5-3)
    (not (at ganon 5-3))
    (not (next_to_ganon))
    (not (alive ganon))))
```

```
(:capability get-key
  :parameters ()
  :precondition (and
    (at Link 1-2)
    (next_to_key)
    (at key 0-2))
  :effect (and
    (not (at Link 1-2))
    (not (at key 0-2)
    (clear 1-2)
    (has-key)
    (at Link 0-2)))
```

```
(:capability open-door
  :parameters ()
  :precondition (and
    (at link 1-1) (has_key)
    (at door 2-1) (clear 2-1)
    (next_to_door)
    (not (alive ganon)))
  :effect (and
    (not (at link 1-1))
    (not (clear 2-1))
    (clear 1-1) (escaped)
    (at link 2-1)))
```

Cook-Me-Pasta

```
(:capability combine
  :parameters ()
  :precondition (and
    (at p1 4-2)
    (at rpasta 2-2)
    (at bwater 3-2))
  :effect (and
    (not (at p1 4-2))
    (not (at rpasta 2-2))
    (not (at bwater 3-2))
    (at p1 3-2)(clear 4-2)
    (at cpasta 2-2)))
```

```
(:capability finish-pasta
  :parameters ()
  :precondition (and
    (at p1 1-0)
    (at cpasta 1-2)
    (at csauce 1-1))
  :effect (and
    (not (at p1 1-0))
    (not (at cpasta 1-2))
    (not (at csauce 1-1)
    (clear 1-0)(at p1 1-1)
    (pasta-cooked)
    (at fpasta 1-2)))
```

```
(:capability push-bwater
  :parameters ()
  :precondition (and
    (at p1 0-1)
    (at bwater 1-1)
    (clear 2-1))
  :effect (and
    (not (at p1 0-1))
    (not (at bwater 1-1))
    (clear 2-1) (clear 0-1)
    (at bwater 2-1)
    (at p1 1-1)))
```

Escape

```
(:capability push-block
  :parameters ()
  :precondition (and
    (at p1 2-2)
    (at b6 2-1)
    (clear 2-0))
  :effect (and
    (not (at p1 2-2))
    (not (at b6 2-1))
    (not (clear 2-0))
    (at b6 2-0)(at p0 2-1)
    (clear 2-2)))
```

```
(:capability pushto-hole
  :parameters ()
  :precondition (and
    (at p1 3-4)
    (at b2 2-4)
    (is-hole 1-4))
  :effect (and
    (not (at p1 3-4))
    (not (at b2 2-4))
    (clear 3-4)
    (at p1 2-4)))
```

```
(:capability escape
  :parameters ()
  :precondition (and
    (at p1 2-3))
  :effect (and
    (not (at p1 2-3))
    (clear 2-3)
    (at p1 2-4)
    (escaped)))
```

Snowman

```
(:capability stack-top
  :parameters ()
  :precondition (and
    (at p1 3-4)(is-goal 2-4)
    (at middle 2-4)
    (at bottom 2-4)
    (player_has top)
    (placed bottom)
    (placed middle))
  :effect (and
    (not (at top 2-4))
    (not (player_has top))
    (placed top)))
```

```
(:capability pick-bottom
  :parameters ()
  :precondition (and
    (at p1 2-0)
    (at bottom-2-1)
    (not (player_has middle))
    (not (player_has top)))
  :effect (and
    (not (at bottom 2-1))
    (clear 2-1)
    (player_has bottom)))
```

```
(:capability drop-top
  :parameters ()
  :precondition (and
    (at p1 0-2) (clear 0-1)
    (player_has top)
    (not (at top-part 0-2)))
  :effect (and
    (not (clear 0-1))
    (not (player_has top))
    (at top 0-1)))
```

Figure 7: Some sample learned grounded capabilities for the four GVGAI domains