Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015)

Raising Expectations in GDA Agents Acting in Dynamic Environments

Dustin Dannenhauer and **Hector Munoz-Avila**
Department of Computer Science and Engineering
Lehigh University
Bethlehem, PA 18015 USA
{dtd212,hem4}@lehigh.edu

Abstract

Goal-driven autonomy (GDA) agents reason about goals while introspectively examining if their course of action matches their expectations. Many GDA agents adopt a hierarchical planning model to generate plans but limit reasoning with expectations to individual actions or projecting the expected state. In this paper we present a relaxation of this limitation. Taking advantage of hierarchical planning principles, our GDA agent elicits expectations that not only validate the next action but the overall plan trajectory without requiring validation against the complete state. We report on (1) a formalization of GDA's expectations that covers trajectories, (2) an implementation of these ideas and (3) benchmarking on two domains used in the GDA literature.

1 Introduction

Goal-driven autonomy (GDA) is a goal reasoning model where an agent interacts in its environment through introspectively examining its own decisions to decide which goal to achieve next [Muñoz-Avila *et al.*, 2010; Weber *et al.*, 2010; Molineaux *et al.*, 2010]. One of the key steps of GDA is discrepancy detection: discerning if the agent's own expectations are met in the current state of the environment. When expectations are not met (i.e., a discrepancy occurs), GDA agents trigger a process that leads to the selection of a new goal for the agent to fulfill. Hence, the agent reasoning about its own expectations is a crucial step in the GDA process.

Despite this, not much effort is reported on how the GDA agents infer their own expectations. Most research on GDA focuses on other aspects of the GDA process such as: explaining why the expectations were not met [Molineaux *et al.*, 2012; Finestrali and Muñoz-Avila, 2013], the procedure by which a new goal is selected [Paisner *et al.*, 2013], and automated learning of GDA knowledge [Jaidee *et al.*, 2011; Weber *et al.*, 2012].

Inferring the expectations of a GDA agent can have a significant impact on the performance of the agent. If the expectations are too narrow, then relevant discrepancies might be missed. As a result, the agent will not change its course of action (by choosing a new goal) in situations where it should have done so. If the expectations are too broad, the agent might unnecessarily trigger the process to generate a new goal. This could lead to the agent taking unnecessary actions that negatively affect the agent's overall performance [Muñoz-Avila *et al.*, 2010].

Research on GDA agents has explored two kinds of expectations thus far [Muñoz-Avila *et al.*, 2010; Molineaux *et al.*, 2010]. The first approach is to check if the preconditions of the next action are satisfied before it is executed and to check that its effects are satisfied after it is executed. We refer to these expectation as *immediate* expectations. This ensures the validity of the actions and the rapid evaluation of their applicability. An argument can be made that since previous actions are already committed, there is no point in validating them. However, such evaluation ignores if the trajectory of the plan is still valid (i.e., if its overarching objectives will be fulfilled). Consider an agent that is navigating in an environment activating beacons. In this environment external factors cause beacons to be disabled which cause plan failure. The *immediate* expectations will only check a particular beacon immediately after it is activated. In the event that the beacon becomes disabled later, the agent will not detect this discrepancy.

The second approach explored is to annotate every action with the expected state after the action is executed. We refer to these expectations as *state* expectations. This ensures that the system not only validates that the next action is valid but it also validates that the overall trajectory of the plan being executed will fulfill its overarching goals. Continuing our example the agent will detect beacons which it had previously switched on but were later disabled, unlike an agent using *immediate* expectations. Yet discrepancies in the state do not necessarily imply that the plan trajectory or even individual actions are no longer valid. For example, beacons unrelated to the agent's goal might activate or deactivate as a result of the environment. These changes do not affect the agent's current plan. However an agent using *state* expectations will flag this as a discrepancy and may trigger a process to correct the false discrepancy.

In this paper we introduce a new kind of expectation: *informed* expectations. Taking advantage of hierarchical task network (HTN) representations, which many GDA agents already adopt, our agents elicit expectations that not only validate the next action but the overall plan trajectory without requiring validation against the complete state.

Our contributions are the following:

- A formalization of GDA's expectations that covers the validation of, both, the next action and the plan trajectory without requiring to check the complete state.
- A GDA agent implementing this formalization. This implementation is based on HTN representations. Hence, our research can be applicable to other GDA agents without the need to encode additional knowledge.
- An empirical validation in two variants of domains used in the GDA literature. We evaluate our GDA agent versus alternative GDA agents that either check immediate effects or check for expected states. Our experiments demonstrate improved performance of our GDA agent.

2 Goal-driven Autonomy

Goal Driven Autonomy is a goal reasoning method for problem solving in which autonomous agents dynamically identify and self-select their goals throughout execution. The objective of goal-driven autonomy is to enable autonomous agents to direct the focus of their activities, and thus become more self-sufficient. GDA has been shown to be amenable to many application areas including cognitive architectures [Choi, 2011], goal generation [Hanheide *et al.*, 2010], goal selection in a mixed initiative [Baxter and Richards, 2010], and meta-reasoning [Cox, 2007]. Reports on potential applications include simulated [Meneguzzi and Luck, 2007] and physically-grounded [Hanheide *et al.*, 2010] robots, real-time strategy games [Weber *et al.*, 2010; Jaidee *et al.*, 2013; Dannenhauer and Muñoz-Avila, 2013], first-person shooter games [Muñoz-Avila *et al.*, 2010], and Navy training simulators [Molineaux *et al.*, 2010; Powell *et al.*, 2011], among others.

GDA agents monitor execution and check if the outcomes of the agent's actions match the agent's own expectations. When discrepancies occur (i.e., when the outcomes do not meet expectations), a GDA monitor will suggest alternative goals. GDA agents generally follow a four-step cycle: discrepancy detection (i.e., check if the outcome of the agent's actions match expectations), explanation (i.e., explain the reasons for discrepancies), goal formulation (i.e., formulate new goals based on the explanations) and goal management (i.e., managing two or more goals that the agent is tasked with pursuing).

The GDA model makes no commitments to the choice of algorithms and representation for these four tasks. Although most GDA systems adopt standard planning formalisms including notions of state and action, which we also adopt.

3 Formalization

3.1 Preliminaries

We use HTN task decomposition as in the SHOP planner [Nau *et al.*, 1999] and implemented in the Python version, PyHop. PyHop uses the state-variable representation [Bäckström and Nebel, 1995]. Informally, a variable can take one of several values (e.g., one may write *above(x)=y* to indicate that block x is on top of block y) and a state S indicates specific values for each variable (we also use a generic *undefined* value when variables have not been instantiated). A *task* is a symbolic representation of an activity in the world. Tasks can be primitive or compound. *Primitive* tasks are accomplished by operators, which have the form $o = (h, pre, eff)$, where h is a primitive task, and *pre*, *eff* are the *preconditions* and the *effects* of the operator. *Actions* are grounded instances of operators.

An action a is *applicable* to a state S if its preconditions hold in that state. The result of applying a to S is a new state $S' = result(a, S)$, that changes the variables' values as indicated in the effects. A *plan* π is a sequence of actions.

A *compound* task is a symbolic representation of a complex activity. An HTN *method* describes how and when to decompose compound tasks into simpler tasks. A *method* is a triple $m = (h, pre, subtasks)$, where h is a compound task, pre are the preconditions of the method, and $subtasks$ is a totally-ordered sequence of tasks.

A method m is *applicable* to a state S and task t if h matches t and its preconditions are satisfied in S. The result of applying method m on state S to decompose task t are the $subtasks$ ($subtasks$ are said to be a *reduction* of t in state S).

An *HTN planning problem* is a 3-tuple (S, T, D), where S is a state, $T = (t_1, ..., t_n)$ is a sequence of tasks, and D is the domain consisting of a set of operators and a set of methods.

A plan $\pi = (a_1...a_m)$ is a *solution for the HTN planning problem (S,T,D)* if the following are true:

Case 1. If $T = \emptyset$ then $\pi = ()$ (i.e., $m = 0$)

Case 2. If $T \neq \emptyset$ (i.e., $m \geq 1$)

Case 2.1 If t_1 is primitive and a_1 is applicable in S and $(a_2...a_m)$ is a solution for $(result(a_1, S), (t_2, ..., t_n), D)$

Case 2.2 If t_1 is compound and $(r_1, ..., r_d)$ is a reduction of t_1 in state S and π is a solution for $((r_1, ..., r_d, t_2, ..., t_n), S, D)$.

3.2 Expectations

The first form of expectations are *immediate expectations*. Given a plan $\pi = (a_1...a_m)$, the *immediate expectation*, $EX_{imm}(S, \pi) = (pre(a_m), eff(a_m))$. That is, $EX_{imm}(S, \pi)$ consists of the preconditions and effects of the last action in π. The same definition can be applied to any prefix of the plan. Immediate expectations check the validity of the next action to be executed.

The second form of expectations are *state expectations*. Given a state S and a plan $\pi = (a_1...a_m)$, $Result(\pi, S)$, applying a plan to a state, extends the notion of applying an action to a state, $Result(a, S)$, as follows:

Case 1. If $\pi = ()$, then $Result(\pi, S) = S$

Case 2. If $\pi \neq ()$, then $Result(\pi, S) = Result((a_2, ..., a_m), Result(a_1, S))$

We assume that action a_1 is applicable in S and that every action a_k ($k \geq 1$) in π is applicable to state $Result((a_1, ..., a_{k-1}), S)$. Otherwise, $Result(\pi, S) = nil$. This recursive definition computes the resulting state for any prefix of the plan and checks the complete state after each action is executed.

Given a state S, a collection of tasks $T = (t_1, ..., t_n)$, and a plan $\pi = (a_1...a_m)$ solving an HTN planning problem

(S, T, D), we define *informed expectations*, the third form of expectation, $EX_{inf}(T, S)$, and the focus of this paper. To define it, we first define $Result(T, S)$, the state resulting after applying tasks to states:

Case 1. If $T = \emptyset$, then $Result(T, S) = S$

Case 2. If $T \neq \emptyset$

Case 2.1 If t_1 is primitive then:

$Result(T, S) = Result((t_2, ..., t_n), Result(a_1, S))$

Case 2.2 If t_1 is compound then:

$Result(T, S) = Result((t_2, .., t_n), S')$ where $S' = Result((r_1, ..., r_d), S)$ and $(r_1, ..., r_d)$ is the reduction of task t_1 on state S generating π.

Let ϕ denote the state where every variable is undefined. Let ξ denote the expectation, which is a placeholder for EX_{inf} We can now define informed expectations, $EX_{inf}(T, S) = Result_{inf}(T, \phi, S)$, as follows:

Case 1. If $T = \emptyset$, then $Result_{inf}(T, \xi, S) = \xi$

Case 2. If $T \neq \emptyset$

Case 2.1 If t_1 is primitive then:

$Result_{inf}(T, \xi, S) = Result_{inf}((t_2, ..., t_n), \xi', S')$, where $\xi' = (\xi \triangleright pre(a_1)) \triangleright eff(a_1)$ and $S' = Result(a_1, S)$. Given two states A and B, $A \triangleright B$ denotes the state in which each variable v is instantiated as follows: (1) v is undefined if v is undefined in A and B, or (2) v takes the non-undefined value c if $v = c$ in B, or (3) v takes the non-undefined value c if $v = c$ in A and v is undefined in B.

Case 2.2 If t_1 is compound then:

$Result_{inf}(T, \xi, S) = Result_{inf}((t_2, ..., t_n), \xi', S')$, where $\xi' = Result_{inf}((r_1, .., r_d), \xi, S)$, $S' = Result((r_1, ..., r_d), S)$ and $(r_1, ..., r_d)$ is a reduction of task t_1 on state S.

$EX_{inf}(T, S)$ represents an intermediate point between checking the action's effects and checking for a complete state after each action in the plan is executed. In general, for the two cases above, ξ will have variables that are undefined whereas the corresponding state S will have no undefined variables. An agent using informed expectations only needs to check in the environment for the values of those variables that are not undefined. In contrast, an agent using state expectations must check the values for all variables. $EX_{inf}(T, S)$ is a recursive definition and, hence, defines the expectation not only for the top level tasks T but for any task sequence in the task hierarchy decomposing T.

4 Computing Expectations in HTN Planning

4.1 Immediate and State Expectations

Immediate expectations and state expectations are straightforward to compute within the SHOP planning algorithm (due to space limitations and because of their simplicity we describe them verbally here). Immediate expectations are computed by building a list of expectations as the plan (i.e., list of actions) is generated. Each time a new action is appended to the plan, that action's definitions (i.e., preconditions and effects) are added as an expectation to the expectations list. State expectations are generated in an analogous manner; namely, a list of expectations is generated alongside the plan. The next state is stored after the current action is applied to the previous state (like any forward state-search planner, SHOP maintains the current state).

4.2 Informed Expectations

The pseudocode for calculating informed expectations is described in Algorithm 1. For any task sequence $(t_1, ..., t_n)$ occurring during the HTN planning process, ex stores the expectations, including the expectation of the last task, t_n, of that sequence. Our algorithm extends the SHOP planning algorithm [Nau *et al.*, 1999]. The extensions to the SHOP algorithm presented here are underlined for easy reference. This algorithm computes the following: (1) a plan π solving the HTN planning problem (S, T, D), (2) a dictionary, ex, mapping for each task t its expectation $ex[t]$, (3) another dictionary, sub, mapping for a task t its children subtasks, $sub[t]$.

The auxiliary procedure *SEEK-EX* is called with the HTN planning problem (S, T, D), the plan π computed so far (initially empty) and the informed expectation ex_{prev} of the previous action in the current plan (initially empty) (Lines 3 and 4). Because informed expectations are cumulative, the planner must maintain the previous expectation throughout the recursion.

Algorithm 1 Informed Expectations

```
 1: Global $\underline{\text{ex, sub}}$
 2: procedure FIND-EX$(S, T, D)$
 3:     return SEEK-EX$(S, T, D, (), ())$
 4: procedure SEEK-EX$(S, T, D, \pi, \underline{\text{ex}_{\text{prev}}})$
 5:     if $T = nil$ then return $((), \underline{\text{ex}_{\text{prev}}})$
 6:     $t$ as the first task in $T$; $R$ = the remaining tasks
 7:     if $t$ is primitive then
 8:         if there is an action $q$ achieving $t$ then
 9:             if $\underline{\text{this is the first action}}$ then
10:                 $\underline{ex[\text{start}] = \text{pre}(q)}$
11:             $\underline{\text{ex}[t] \leftarrow (ex_{prev} \triangleright pre(q)) \triangleright eff(q)}$
12:             $\underline{\text{sub}[t] \leftarrow ()}$
13:             $\pi \leftarrow \text{append}(\pi, q)$
14:             return SEEK-EX$(\text{result}(q, S), \pi, R, D, \underline{\text{ex}[t]})$
15:         else
16:             return Fail
17:     else
18:         for every reduction $r$ of $t$ in $S$ do
19:             $\underline{\text{sub}[t] \leftarrow r}$
20:             $\underline{(\pi_t, \text{ex}[t])} \leftarrow$ SEEK-EX$(S, r, D, (), \underline{\text{ex}_{\text{prev}}})$
21:             $\pi' \leftarrow append(\pi, \pi_t)$
22:             $S' \leftarrow result(\pi_t, S)$
23:             $t' \leftarrow$ last task of $T$
24:             $(\pi, \underline{\text{ex}[t']}) \leftarrow$ SEEK-EX$(S', R, D, \pi', \underline{\text{ex}[t]})$
25:             if $\pi \neq$ Fail then
26:                 return $(\pi, \underline{\text{ex}[t']})$
27:         return Fail
```

If the first task t in T is primitive (Lines 6 and 7), then it finds an action q that achieves t (Line 8). If this is the first action in the plan, then the preconditions of that action are stored as the very first expectation (Line 10). This expectation is stored in the dictionary with a key value of a special start symbol and not a task (Lines 9 and 10). All other key values of the dictionary are task names.

Line 11 produces the informed expectation for the primitive task t. Since t is primitive, it has no children (Line 12). The expectation for task t, $ex[t]$ becomes the previous expectation and it is then passed into the recursive call of SEEK-EX (Line 14).

Line 18 handles the case that t is not primitive, and therefore may have one or more reductions for t in state S. There are two recursive calls. First, a call is made on the reduction r; the resulting expectation is stored as the informed expectation of t (Line 20). The current plan, π' is obtained by appending the subplan π_t to the end of the previous plan π (Line 21). The current state, S' is obtained by applying the subplan π_t to state S (Line 22). Second, a call is made on the remaining tasks R (Line 24). The resulting expectation is the expectation for the last task, t' in T.

5 Experimental Setup

Each of the three different kinds of expectations (*immediate, state, informed*) are implemented in otherwise identical GDA agents. Specifically, all agents use the same goals, HTN planning domain, explanations, and goal formulation knowledge. For the explanation knowledge, we have simple rules assigning discrepancies to explanations (i.e., if the agent is not moving then it must be stuck). Analogously, for the goal formulation we have simple rules assigning explanations to new goals (i.e., if stuck then achieve unstuck goal). The simplicity of our GDA agents is designed so we can adjudicate performance differences among the agents to the different kinds of expectations.

We use two domains from GDA literature. The first domain, which we call *Marsworld*, is inspired from *Mudworld* from [Molineaux and Aha, 2014]. *Mudworld* is composed of a discrete grid and every tile can either have mud or no mud; mud is randomly generated with a 40% chance probability per tile at the beginning of each scenario. The agent can only observe its current and adjacent locations. The goal of the agent is to navigate from its starting position to another tile at least 4 tiles away.

In *Mudworld* mud is an obstacle that causes the speed of the agent to be halved; in our domain *Marsworld* mud causes the agent to be stuck. This difference allows us to examine obstacles that require choosing new goals. The other difference between *Marsworld* and *Mudworld* is the addition of a second task and a new obstacle. The new task is a beacon-placing perimeter construction task in which the agent must place and activate three beacons in locations at least two spaces from any other beacon. The new obstacle is a magnetic radiation cloud which may appear on tiles and if sharing the same tile as a deployed beacon, deactivates the beacon from transmitting its signal. Unlike mud which is visible to the agent when it is in an adjacent tile, magnetic radiation clouds are not visible. While deployed and activated, the beacons broadcast a signal. The agent may generate a goal to reactivate a beacon by sending a signal to the beacon, which will then become active unless another radiation cloud disables it again. These additions to the original *Mudworld* were designed to add more complexity to the domain by having the agents facing obstacles which affect some goals but not others. Mud affects both navigation and perimeter-construction whereas radiation clouds only affect the perimeter goal.

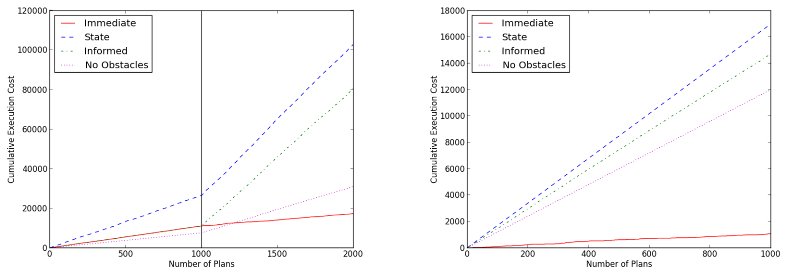
Execution cost of a plan in *Marsworld* is calculated as follows: moving from one tile to another has a cost of 1 and placing a beacon has a cost of 1. The 'unstuck' action has a cost of 5 (can be used when the agent finds itself in mud) and the 'reactivate' beacon via signal action has a cost of 1. The following parameters were used in the *Marsworld* setup: the grid was 10 by 10, the probability of mud was 10%, all distances from start to destination were at least 5 tiles, and magnetic radiation clouds had a 10% probability per turn per tile to appear.

The second domain is a slight variant of the *Arsonist* domain from Paisner et al [Paisner *et al.*, 2013]. This domain uses the standard operators and goals from blocksworld domain, except the Arsonist domain features an arsonist who randomly lights blocks on fire. The variation in our model is the addition of a precondition to the stacking operator prohibiting for the block underneath to be on fire so plans can fail if the fire is not extinguished. In the original *Arsonist* domain, the effect of a fire that was not extinguished was undefined in the planning domain. Originally fires that were not put out would cause the score of the agent to decrease. Since our work focuses on how different expectations affect plan completion, adding this precondition causes execution failure when fires are ignored. Execution cost of a plan in our modified *Arsonist* domain is computed as follows: normal blocksworld operations (pickup, stack, unstack, etc) cost 1 and the action to extinguish a fire costs 5. The following parameters were used in the Arsonist domain: the domain contained 20 blocks, the start state had every block on the table, each goal was randomly generated where there were 3 towers each with 3 blocks, and the probability of fire was 10%. In both domains, if there are no obstacles then the execution cost is the length of the plan.

6 Results

Figure 1 shows the average results of 5 runs from the GDA agents on *Marsworld* and *Arsonists* domains respectively. The x-axis is the number of plans the agent has executed thus far and the y-axis is the cumulative execution cost (every data point is the execution cost of the plan plus the previous execution cost). The solid red line presents the results for the agent using immediate expectations, the blue dashed line is for the agent using state expectations, and the green dot dashed line is for the agent with informed expectations. As a reference, the execution cost of plans with no obstacles is included: this is the purple dotted line.

In the first domain (Figure 1 (a)), *Marsworld*, during navigation goals the informed expectations agent and immediate expectations agent performed equally and took slightly longer to execute than an agent facing no obstacles (Figure 1 (a)). However, the state agent ended up taking much longer because it triggered false anomalies. This is due to mud ob-



(a) Marsworld (b) Arsonist

Figure 1: Cumulative Execution Cost

stacles that were not directly in its path but still counted as discrepancies since the expected and actual states were not the same. This is the result seen from the left half of the graph (plans 1 to 1000). The right hand side of the graph measures execution cost of plans from perimeter construction. Here, the immediate expectations agent falls below the perfect agent (no obstacles) because it's execution starts failing when it fails to detect beacons are missing/unavailable. These failures result in a plan execution cost of 0 causing the line to fall below the baseline. We also see that both, informed and state agents, are able to succeed in completing the plan although the informed expectations costs significantly less.

Figure 2 (a) shows the percentage of completed plans for the immediate expectations agents since this was the only agent that fail to complete plans. Each data point in this graph represents the percentage of plans that were successful (i.e. did not fail), averaged over 5 runs. The x-axis is the probability per tile per turn that a magnetic radiation cloud will appear. The y-axis is the probability per tile of mud occurring. The y-axis is the percentage of plans that were successful after executing 20 perimeter plans with the corresponding probabilities of mud and clouds. We conclude that clouds are the only causes of failure (as opposed to mud). In the arsonist domain we see analogous behavior (Figure 2 (b)). As the probability of fire increases (x-axis) the immediate expectations agent fails to complete more plans (y-axis).

7 Related Work

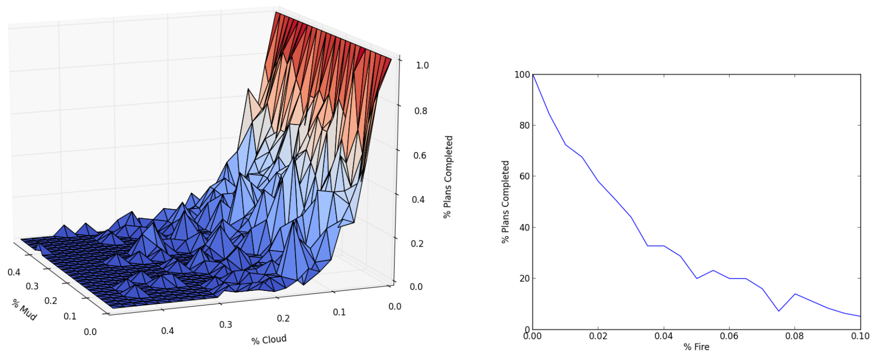
Plan-execution monitoring systems check if the current state satisfies the effects of the action just executed and the preconditions of the actions to be executed next. When this does not happen and as a result the action is inapplicable, this is called an expectation failure [Cox, 2007], which are the kinds of failures we are focusing in this work. Other kinds of failures have been proposed where even though they are not execution failures, the current execution exhibits some conditions that are not desirable [Myers, 1999]. Others have suggested failures associated with explicit quality conditions [Fritz and McIlraith, 2007].

The study of expectation failures have been long-standing particularly in the context of cognitive systems. Mechanisms to enhance a domain description when planning failures occur have been designed for cognitive systems (e.g., [Birnbaum *et al.*, 1990; Sussman, 1975]). In contrast to this research, GDA systems generate new goals as a result of expectation failures.

Hierarchical representations have been a central topic in many cognitive architectures (e.g., [Laird, 2012]). Part of the motivation is the principle that Humans learn complex skills by first acquiring simple skills, which then are combined to learn more complex ones [Langley *et al.*, 2004]. As a result, many cognitive architectures use hierarchical models to represent relations between skills of different complexity. Hierarchical modeling has led to a number of representation and reasoning formalisms including frames [Minsky, 1975], reasoning by abstraction [Amarel, 1968], and hierarchical task network (HTN) planning [Currie and Tate, 1991]. These formalisms have in common the use of certain kinds of constructs (i.e., objects, goals, and tasks) that represent knowledge of varying degrees of complexity and that are connected through hierarchical relations. In our work we adopt the HTN planning formalism because of its clear semantics.

Our work is related to plan repair, which aims at modifying the current plan when changes in the environment make actions in the plan invalid [Van Der Krogt and De Weerdt, 2005]. Plan repair has also been studied in the context of HTN planning [Warfield *et al.*, 2007]. The main difference between plan repair and GDA is that in the latter the goals might change whereas plan repairs stick with the same goals while searching for alternative plans.

Some GDA systems do not assume planning knowledge to be given. For example, in [Virmani *et al.*, 2008], a sequence of code calls (i.e., headings of actions) is collected as observed while a user is playing a computer game. The user is then asked to annotate the code calls with literals that must be valid in the state. Alternatively, learning of the planning model in the context of GDA has been studied [Jaidee *et al.*, 2011]. For GDA systems that assume the actions model (i.e., preconditions and effects) to be given, the expectations are the state, as determined by the previously observed state and the model of the actions executed [Cox, 2007]. In our work, we use HTN plan representations, adopted by many GDA systems [Muñoz-Avila *et al.*, 2010; Shivashankar *et al.*, 2013; Molineaux *et al.*, 2010; Klenk *et al.*, 2013]. Alternatives include using description logics to infer expectations after executing a plan [Bouguerra *et al.*, 2007] and Petri nets to follow the execution of the program [Ontañón and Ram, 2011].



(a) Marsworld (b) Arsonist

Figure 2: Plan Success vs. Obstacles (Immediate Expectations agent)

Each code call is one edge in the Petri net and states connect edges of the net. A token "jumps" from state to state as the code calls software routines are executed. If the state is not reached, this is a discrepancy and it can be represented by a Boolean variable. So this is akin to what we call state expectations.

Unlike STRIPS representations where the outcome of a plan is directly related to whether the literals in the final state reached satisfy the goals, HTN representations must, in principle, consider if the pursued task is accomplished regardless of the literals in the state reached. This is due to the semantics of HTN planners where the state that is reached is not necessarily linked to the task being accomplished. TMK and TMKL2 have a similar characterization of expectations presented here but support debugging and adaptation in the context of meta-reasoning [Murdock and Goel, 2008] while GDA uses it for goal formulation. Informed expectations check literals in the space by projecting the informed state across tasks at all levels of the HTN.

The notion of informed expectations is related to goal regression [Mitchell *et al.*, 1986]. Goal regression is the process of finding the minimal set of literals in the initial state required to generate a plan achieving some goals. This process traverses a plan to collect all literals that are not added by actions in the plan. In contrast, in our work we are identifying the literals in the final state that are expected to hold. The most important difference, however, is that we are computing the expectations for tasks at all levels of the HTN.

8 Final Remarks

Over the last few years we have seen a significant increase in research and applications of systems with increasing autonomous capabilities. As these systems become more common, concerns have been raised about how we can design autonomous systems that are robust. That is, a system that reliably operates within expected guidelines even when acting in dynamic environments. Research about agent's expectations can be seen as a step towards addressing this difficult problem; reasoning about the agents' own expectations enables agents to check if the expectations of the course of action are currently met. In this paper, we re-visit two kinds of expectations discussed in the GDA literature: immediate and state expectations. We formally define and implement a third kind, ***informed*** expectations. Informed expectations capture what is needed for plan trajectory (unlike immediate expectations) while ignoring changes in the environment irrelevant to the current plan (unlike an agent using state expectations). Our experiments demonstrate improved performance for GDA agents using informed expectations on two metrics: execution costs and percentage of execution failures.

In future work, we will like to explore how informed expectations combine with sophisticated failure explanation mechanisms such as DiscoverHistory [Molineaux *et al.*, 2012], which learns new explanation over time. Such combination could enable, for example, meta-reasoning on failure reasons, whereby the explanation module discovers flaws in the domain knowledge. For instance, the explanation module might identify a missing condition in the informed expectation and introspectively examine the HTN and suggest necessary changes to methods and/or operators to add the missing condition. The key insight is that we know informed expectations compute the exact expectations for the current HTN and hence missing conditions hypothesized by the explanation module as necessary would imply a flaw in the HTN knowledge base.

Acknowledgements: This work was supported in part by the National Science Foundation Grant #1217888.

References

[Amarel, 1968] Saul Amarel. On representations of problems of reasoning about actions. *Machine intelligence*, 3(3):131–171, 1968.

[Bäckström and Nebel, 1995] Christer Bäckström and Bernhard Nebel. Complexity results for sas+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

[Baxter and Richards, 2010] Jeremy Baxter and Dale Richards. Whose goal is it anyway? user interaction in an autonomous system [c]. In *Proc. of the workshop on Goal Directed Autonomy, AAAI2010, Atlanta*, 2010.

[Birnbaum *et al.*, 1990] Lawrence Birnbaum, Gregg Collins, Michael Freed, and Bruce Krulwich. Model-based diagnosis of planning failures. In *AAAI*, volume 90, pages 318–323, 1990.

[Bouguerra *et al.*, 2007] Abdelbaki Bouguerra, Lars Karlsson, and Alessandro Saffiotti. Active execution monitoring using planning and semantic knowledge. *ICAPS Workshop on Planning and Plan Execution for Real-World Systems*, 2007.

[Choi, 2011] Dongkyu Choi. Reactive goal management in a cognitive architecture. *Cognitive Systems Research*, 12(3):293–308, 2011.

[Cox, 2007] Michael T Cox. Perpetual Self-Aware Cognitive Agents. *AI magazine*, 28(1):32, 2007.

[Currie and Tate, 1991] Ken Currie and Austin Tate. O-plan: the open planning architecture. *Artificial Intelligence*, 52(1):49–86, 1991.

[Dannenhauer and Muñoz-Avila, 2013] D. Dannenhauer and H. Muñoz-Avila. LUIGi: A Goal-Driven Autonomy Agent Reasoning with Ontologies. In *Advances in Cognitive Systems (ACS-13)*, 2013.

[Finestrali and Muñoz-Avila, 2013] Giulio Finestrali and Héctor Muñoz-Avila. Case-based learning of applicability conditions for stochastic explanations. In *Case-Based Reasoning Research and Development*, pages 89–103. Springer, 2013.

[Fritz and McIlraith, 2007] Christian Fritz and Sheila A McIlraith. Monitoring plan optimality during execution. In *ICAPS*, pages 144–151, 2007.

[Hanheide *et al.*, 2010] Marc Hanheide, Nick Hawes, Jeremy Wyatt, Moritz Göbelbecker, Michael Brenner, Kristoffer Sjöö, Alper Aydemir, Patric Jensfelt, Hendrik Zender, and Geert-Jan Kruijff. A framework for goal generation and management. In *Proceedings of the AAAI workshop on goal-directed autonomy*, 2010.

[Jaidee *et al.*, 2011] Ulit Jaidee, Héctor Muñoz-Avila, and David W Aha. Integrated Learning for Goal-Driven Autonomy. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Three*, pages 2450–2455. AAAI Press, 2011.

[Jaidee *et al.*, 2013] Ulit Jaidee, Héctor Muñoz-Avila, and David W Aha. Case-based goal-driven coordination of multiple learning agents. In *Case-Based Reasoning Research and Development*, pages 164–178. Springer, 2013.

[Klenk *et al.*, 2013] Matthew Klenk, Matt Molineaux, and David W Aha. Goal-driven autonomy for responding to unexpected events in strategy simulations. *Computational Intelligence*, 29(2):187–206, 2013.

[Laird, 2012] John E. Laird. *The Soar Cognitive Architecture*. The MIT Press, 2012.

[Langley *et al.*, 2004] Pat Langley, Kirstin Cummings, and Daniel Shapiro. Hierarchical skills and cognitive architectures. In *Proceedings of the twenty-sixth annual conference of the cognitive science society*, pages 779–784. Citeseer, 2004.

[Meneguzzi and Luck, 2007] Felipe Meneguzzi and Michael Luck. Motivations as an abstraction of meta-level reasoning. In *Multi-Agent Systems and Applications V*, pages 204–214. Springer, 2007.

[Minsky, 1975] Marvin Minsky. A framework for representing knowledge. 1975.

[Mitchell *et al.*, 1986] Tom M Mitchell, Richard M Keller, and Smadar T Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine learning*, 1(1):47–80, 1986.

[Molineaux and Aha, 2014] Matthew Molineaux and David W Aha. Learning unknown event models. Technical report, DTIC Document, 2014.

[Molineaux *et al.*, 2010] Matthew Molineaux, Matthew Klenk, and David W Aha. Goal-Driven Autonomy in a Navy Strategy Simulation. In *AAAI*, 2010.

[Molineaux *et al.*, 2012] Matthew Molineaux, Ugur Kuter, and Matthew Klenk. Discoverhistory: Understanding the past in planning and execution. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 989–996, 2012.

[Muñoz-Avila *et al.*, 2010] Héctor Muñoz-Avila, David W Aha, Ulit Jaidee, Matthew Klenk, and Matthew Molineaux. Applying Goal Driven Autonomy to a Team Shooter Game. In *FLAIRS Conference*, 2010.

[Murdock and Goel, 2008] J William Murdock and Ashok K Goel. Meta-case-based reasoning: self-improvement through self-understanding. *Journal of Experimental & Theoretical Artificial Intelligence*, 20(1):1–36, 2008.

[Myers, 1999] Karen L Myers. Cpef: A continuous planning and execution framework. *AI Magazine*, 20(4):63, 1999.

[Nau *et al.*, 1999] Dana Nau, Yue Cao, Amnon Lotem, and Hector Muñoz-Avila. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, pages 968–973. Morgan Kaufmann Publishers Inc., 1999.

[Ontañón and Ram, 2011] Santiago Ontañón and Ashwin Ram. Case-based reasoning and user-generated artificial intelligence for real-time strategy games. In *Artificial Intelligence for Computer Games*, pages 103–124. Springer, 2011.

[Paisner *et al.*, 2013] Matt Paisner, Michael Maynord, Michael T Cox, and Don Perlis. Goal-driven autonomy in dynamic environments. In *Goal Reasoning: Papers from the ACS Workshop*, page 79, 2013.

[Powell *et al.*, 2011] Jay Powell, Matthew Molineaux, and David W Aha. Active and Interactive Discovery of Goal Selection Knowledge. In *FLAIRS Conference*, 2011.

[Shivashankar *et al.*, 2013] Vikas Shivashankar, UMD EDU, Ron Alford, Ugur Kuter, and Dana Nau. Hierarchical goal networks and goal-driven autonomy: Going where ai planning meets goal reasoning. In *Goal Reasoning: Papers from the ACS Workshop*, page 95, 2013.

[Sussman, 1975] Gerald Jay Sussman. *A Computer Model of Skill Acquisition*. Elsevier Science Inc., New York, NY, USA, 1975.

[Van Der Krogt and De Weerdt, 2005] Roman Van Der Krogt and Mathijs De Weerdt. Plan repair as an extension of planning. In *ICAPS*, volume 5, pages 161–170, 2005.

[Virmani *et al.*, 2008] Suhas Virmani, Yatin Kanetkar, Manish Mehta, Santiago Ontanón, and Ashwin Ram. An intelligent ide for behavior authoring in real-time strategy games. In *AIIDE*, 2008.

[Warfield *et al.*, 2007] Ian Warfield, Chad Hogg, Stephen Lee-Urban, and Héctor Munoz-Avila. Adaptation of hierarchical task network plans. In *FLAIRS Conference*, pages 429–434, 2007.

[Weber *et al.*, 2010] Ben George Weber, Michael Mateas, and Arnav Jhala. Applying Goal-Driven Autonomy to StarCraft. In *AIIDE*, 2010.

[Weber *et al.*, 2012] Ben George Weber, Michael Mateas, and Arnav Jhala. Learning from demonstration for goal-driven autonomy. In *AAAI*, 2012.