2017

Self Monitoring Goal Driven Autonomy Agents

Dustin Dannenhauer
Lehigh University

SELF MONITORING GOAL DRIVEN AUTONOMY AGENTS

BY

DUSTIN DANNENHAUER

Presented to the Graduate and Research Committee
of Lehigh University
in Candidacy for the Degree of
Doctor of Philosphy

in

Computer Science

Lehigh University

May, 2017

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosphy.

____

Date

____

Accepted Date

Committee Members:

____

Dr. Héctor Muñoz-Avila

____

Dr. Jeff Heflin

____

Dr. Brian D. Davison

____

Dr. Michael T. Cox

Acknowledgements

I have received a tremendous amount of support during my studies and though I only mention a few individuals here, I am deeply appreciative of everyone who has helped me along the way. I am most grateful to my advisor, Prof. Héctor Muñoz-Avila who provided constant feedback, encouragement, and revisions. I would like to thank the rest of my committee: Prof. Jeff Heflin, Prof. Brian D. Davison, and Dr. Michael T. Cox. I am thankful to Dr. Cox for the opportunity to work on the MIDCA project and with the students at Wright State University.

I would like to acknowledge Dr. David W. Aha who first introduced me to the field of goal reasoning during my summer internship at the Naval Research Laboratory. My experiences there provided a strong foundation leading me to produce the work in this thesis.

I am thankful for the many undergraduate teaching assistant opportunities from Suzanne Menzel and my first summer of research advised by Dr. David Leake, without whom I would not have found my passion for artificial intelligence. I would also like to acknowledge the Ronald E. McNair Post-baccalaureate Achievement Program which funded my undergraduate research and supported me to pursue a doctoral program.

My work on inferred expectations for real-time strategy games started as a group project with my classmates Kostas Hatalis and Will West. Their help in building the ontology and interface with the Starcraft: Broodwar game engine allowed me to continue research in that direction, which lead to chapters 6 and 7 of this thesis. I also thank Kostas for his friendship and camaraderie as a fellow doctoral student with whom I could share my experiences.

Finally, I would like to dedicate this dissertation to my wife, Zohreh, for her understanding and support that kept me going through the late nights and long hours.

Dustin Dannenhauer
May 5th, 2017

Contents

List of Tables

List of Figures

Abstract

The growing abundance of autonomous systems is driving the need for robust performance. Most current systems are not fully autonomous and often fail when placed in real environments. Via self-monitoring, agents can identify when their own, or externally given, boundaries are violated, thereby increasing their performance and reliability. Specifically, self-monitoring is the identification of unexpected situations that either (1) prohibit the agent from reaching its goal(s) or (2) result in the agent acting outside of its boundaries. Increasingly complex and open environments warrant the use of such robust autonomy (e.g., self-driving cars, delivery drones, and all types of future digital and physical assistants). The techniques presented herein advance the current state of the art in self-monitoring, demonstrating improved performance in a variety of challenging domains.

In the aforementioned domains, there is an inability to plan for all possible situations. In many cases all aspects of a domain are not known beforehand, and, even if they were, the cost of encoding them is high. Self-monitoring agents are able to identify and then respond to previously unexpected situations, or never-before-encountered situations.

When dealing with unknown situations, one must start with what is expected behavior and use that to derive unexpected behavior. The representation of expectations will vary among domains; in a real-time strategy game like Starcraft, it could be logically inferred concepts; in a mars rover domain, it could be an accumulation of actions' effects. Nonetheless, explicit expectations are necessary to identify the unexpected.

This thesis lays the foundation for self-monitoring in goal driven autonomy agents in both rich and expressive domains and in partially observable domains. We introduce multiple techniques for handling such environments. We show how inferred expectations are needed to enable high level planning in real-time strategy games. We show how a hierarchical structure of Goal-driven Autonomy (GDA) enables agents to operate within large state spaces. Within Hierarchical Task Network planning, we show how informed expectations identify states that are likely to prevent an agent from reaching its goals in dynamic domains. Finally, we give a model of expectations for self-monitoring at the meta-cognitive level, and empirical results of agents equipped with and without metacognitive expectations.

Part I

Foundation

Chapter 1

Introduction

There is great value in autonomy. To the degree that one has autonomy, to that degree they are able to accomplish their pursuits without assistance. Surely as intelligent systems improve, we would like them to require less intervention from humans. A major limitation of many artificial intelligence solutions is the ability to deal with anomalous and unexpected situations. In complex environments, an agent designer cannot be expected to foresee and impart all possible outcomes to the agent. Therefore a truly autonomous agent would be robust to new and unexpected situations. This leads us to the primary research question:

How can we achieve robust autonomy capable of responding competently to new and unexpected situations?

An answer to this question will need to address the following sub-questions:

How can an agent identify a new and unexpected situation? (Identification)

How should an agent respond once it finds itself in such a situation? (Response)

This thesis begins to answer the first of the two sub-questions which we will refer to as the *identification question* (note that the second question is not relevant unless the first has been answered). The scope of the work is focused on systems with automated planning and goal reasoning methodologies. That is, we are concerned with agents that have explicit goals, generate plans to achieve their goals, and execute their plans in dynamic and complex environments. Furthermore, goal reasoning agents may formulate new goals, and self-select which goals to pursue over the course of their lifetime.

One of the primary motivations for goal reasoning is that when something unexpected occurs, it may be better to change one's goal(s) instead of re-planning. For example, suppose an autonomous mars rover needs to identify a nearby exposed rock and collect a sample. The original goal is to obtain a photograph and a piece of the rock. However, while the agent is navigating towards the rock, a storm begins blowing large amounts of sand, covering the rock. In this situation, the original goal may no longer be ideal because the rock is no longer exposed and the agent will need to remove a significant amount of sand in order to continue. It may be better for the agent to change its goal to look for nearby rocks or return another day. Goal reasoning agents have a higher level of robustness than re-planning only agents, since they are concerned with not only how to achieve their goals, but also which goals to pursue. We focus specifically on the Goal-Driven Autonomy (GDA) model which is one such implementation of goal reasoning. The defining characteristic of GDA agents is that they adhere to a four step cycle:

Step 1: ***Discrepancy Detection*** observes the environment for anomalies, and when found, generates corresponding discrepancies D.

Step 2: ***Explanation*** generates explanations E for discrepancies D.

Step 3: ***Goal Formulation*** may generate new goals G_{new} taking into account the discrepancies D and explanations E.

Step 4: ***Goal Selection*** selects which goals the agent will pursue next. Goal selection finishes the GDA process until it is triggered again during discrepancy detection.

In our attempt to answer the *identification question*, we focus on the first step, discrepancy detection, in GDA agents. Significant previous work has been carried out developing the remaining Steps 2, 3, and 4 (see Chapter 2 for some examples). Discrepancies are the violations of expectations observed when an agent finds itself with an anomalous or unexpected situation. Discrepancy detection uses expectations to find discrepancies. Expectations are knowledge artifacts that enable agents to check if they are operating as intended. When dealing with unknown situations, one must start with what is expected behavior and use that to derive unexpected behavior. The representation of expectations will vary among domains; in a real-time strategy game like Starcraft, it could be logically inferred concepts; in a mars rover domain, it could be an accumulation of actions' effects. The contributions of this thesis are concerned with the generation, representation, and use of such expectations.

The contributions are briefly summarized as:

- New classes of GDA Expectations
  - Informed Expectations: Using the description of an agent's actions, accumulate the effects of actions executed thus far by the agent.
  - Inferred Expectations: Using a semantic ontology to infer abstract concepts by reasoning over directly observable facts in the environment.

  - Hierarchical Expectations: Using a hierarchy of expectations that correspond to a hierarchical plan for plan execution.
  - Metacognitive Expectations: Using heuristic-like knowledge about cognitive processes to identify domain-independent failures that occur in cognitive processes.

- Formalization of the Guiding Sensing Problem: A problem where agents experience a trade-off between minimizing sensing costs and maximizing goal achievement in dynamic and partially observable environments. The Guiding Sensing Problem assumes the agent has a capability to verify its belief state by performing additional sensing at some cost.

- Empirical validation of these concepts: A number of domains are used to validate performance benefits of such techniques in dynamic domains that range in fully observable to partially observable, including one domain, NBeacons, that changes over time.

A guide to the contributions of this work and their corresponding locations in this document are shown in Table 1.1.

A formalism of GDA expectations is touched upon in every chapter in Part II: Contributions. Chapter 4 describes expectations generated from a Hierarchical Task Network (HTN) planner. Chapter 5 extends the formalization of approaches to expectations by introducing *eager* expectations as well as describing how informed expectations can be used by plans of length 1. Chapter 6 formalizes expectations by distinguishing between primitive and compound expectations as they relate to ontological inference. Chapter 7 describes hierarchical expectations for hierarchical plans. Finally, Chapter 8

Contribution	**Chapters**
A formalism of GDA expectations	4, 5, 6, 7, 8
Informed expectations: an improvement to previous approaches in dynamic and partially observable domains	4, 5
A formalism of the guiding sensing problem	5
LUiGi : A GDA agent for the Real-Time Strategy Game Starcraft: Broodwar that uses inferred concepts as expectations to enable high-level planning	6
LUiGi-H : A GDA agent that extends *LUiGi* by using hierarchical plans and hierarchical expectations.	7
A formalism for hierarchical plans and hierarchical expectations	7
An ontology for the domain of Starcraft: Broodwar, including concepts inferred from low-level triples	6, Appendix A
Formalism of the notion of metacognitive expectations	8

Table 1.1: Locations of contributions

formalizes metacognitive expectations, giving an example of expectations for individual cognitive processes.

Chapter 4 introduces the informed expectations approach to expectations and 5 discusses how they are useful in partially observable domains requiring exploration for goal achievement.

Chapter 5 formalizes the guiding sensing problem. The guiding sensing problem describes situations in which an agent must decide what additional sensing is cost-effective in domains that are dynamic and partially observable. When an agent interacts with an object as and moves out of sight of that object, the object's status may change. By performing additional sensing the agent could determine the true status of that object with some cost. Determining which objects are relevant to goal achievement provides a basis for sensing targets.

LUiGi is an agent that uses inferred concepts as expectations, enabling a higher level planning. Chapter 6 describes the inference mechanisms and system architecture used to detect discrepancies in the Real-Time Strategy game Starcraft: Broodwar. *LUiGi-H* is an extension of *LUiGi* that makes use of hierarchical plans and corresponding hierarchical expectations. The ontology is given in Appendix A.

Metacognitive expectations begin to solve the problem of identifying discrepancies originating in an agent's own cognitive processes. Using a model for mental states and mental actions (a mental action represents a cognitive process), we formalize metacognitive expectations regarding individual mental actions.

Chapter 9 provides descriptions of six new domains: Marsworld^1, Marsworld^2, Arsonist$^+$, NBeacons, BlocksCraft. Empirical results follow demonstrating the benefits of certain approaches.

This document is organized as follows. We give a background of related topics in Chapter 2. Chapter 3 contains a literature review of the state of the art in agents using discrepancy detection in planning and execution and goal reasoning research. Chapter 4 introduces informed expectations in HTN planning and execution. Chapter 5 introduces the guiding sensing problem and how informed expectations can be used for sensing. Chapter 6 presents an approach of using an ontology for inferred expectations which enable high level planning actions. Chapter 7 discusses an agent that can play full games of Starcraft by combining Case Based Reasoning, GDA, and high level planning operators. Chapter 8 introduces a formalism for metacognitive expectations. In Part III: Conclusions, we present future research (Chapter 10) and summarize the thesis in Chapter 11.

Chapter 2

Background

2.1 Planning

Many goal-driven autonomy agents employ planning capabilities. Planning is a problem of finding the actions needed to reach a goal. Informally, planning is a process that performs a search over actions and states. Actions have applicability conditions (often referred to as *preconditions*) that restrict in which states they can be applied. Here, a state is some representation of the environment. For example, consider a one-armed agent and an action to *pickup* an object. The *pickup* action can only be applicable if the agent is not already holding an object. Actions cause changes to new states (usually referred to as *effects*). In the example of the one-armed agent successfully executing the *pickup* object action, the agent would find itself in a new state where the object is located in its hand and not located on the table. By planning, an agent can generate a sequence of actions to take in order to change the current state into a state that contains the agent's goal.

More formally, given a model of the domain in the form of a state-transition system

$\Sigma = (S, A, \gamma)$ where S is the set of all states, A is the set of all actions, and $\gamma : S \times A \rightarrow S$ is a function that given a state s_i and action a_i produces the subsequent state s_{i+1} after a_i is executed in state s_i. A planning problem P is defined as $P = (\Sigma, s_0, g)$ with s_0 as the initial state and g as the goal state to be reached. A solution to the planning problem P is a plan. π, which is a sequence of actions $< a_0, a_1, ..., a_n >$ such that each action applied to the starting state s_0 will result in a $g \subset s_{n+1}$ where s_{n+1} is the state of the world after executing the last action in the plan and the goal is reached.

The actions that compose a plan are instantiations of planning operators. A planning operator o is a tuple $(head, params, pre, post)$ where $head$ is the name of the operator, $params$ are the arguments to the operator, pre is the preconditions of the operator, and $post$ is the postconditions of the operator. For an operator to be applicable in a given state, the preconditions pre must be valid in the state. The postconditions $post$ of an operator represent the changes to the state after the operator as been executed.

Head activate-beacon
Params ?agent, ?loc, ?bcn
Pre agent-at(?agent, ?loc), beacon-at(?bcn, ?loc), deactivated(?bcn)
Post ¬deactivated(?bcn), activated(?bcn)

Figure 2.1: Operator Definition for activate-beacon

activate-beacon(Curiosity, (5,4), B7)

Figure 2.2: Instantiated Action of activate-beacon

{activated(B2), activated(B5), activated(B7) }

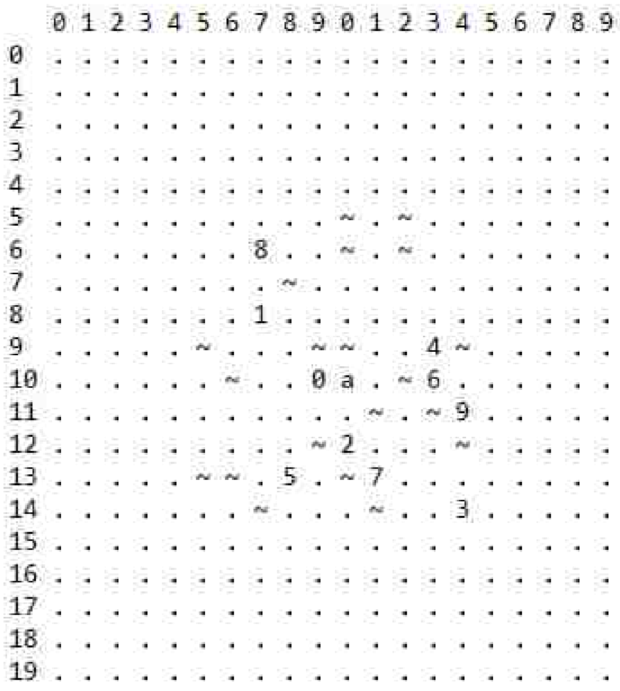
Figure 2.3: Example of a goal to activate beacons B2, B5, and B7

Figure 2.1 shows the operator definition for *activate-beacon* and Figure 2.2 shows an action instantiated from this operator. The instantiated action *activate-beacon* shown in Figure 2.2 is only applicable in a state that has the instantiated preconditions met: *agent-at(Curiosity, (5,4))*, *beacon-at(B7, (5,4))*, *deactivated(B7)*.

Goals are represented as subsets of states. For example, a goal to activate beacons B2, B5, and B7 would take the form shown in Figure 2.3. Goals can be combined via a set union operation if there are no contradictory atoms. For example, individual goals to activate beacons B2, B5, B7 are easily combined into the goal shown in Figure 2.3

2.1.1 Domain Representations

The autonomous agents used in this work use an explicit model of the environment[1]. Domains are represented as a set of atoms. An atom is defined as a predicate (e.g., *activated*) and 1 or more parameters (e.g., B7). An atom represents a fact in the environment. In the NBeacons domain (described in Section 9.1.6) an agent navigates a grid with the purpose of activating beacons. Beacons are activated when an agent is in the same location as a beacon and executes an *activate-beacon* action. A visual diagram of an initial scenario, S_0, is shown in Figure 2.4 and the corresponding collection of atoms representing S_0 is shown in Figure 2.1. Figure 2.5 shows a valid plan to achieve the goal *activated(B6)* given that the start state is S_0 shown in Figure 2.4.



```
   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
0  . . . . . . . . . . . . . . . . . . . .
1  . . . . . . . . . . . . . . . . . . . .
2  . . . . . . . . . . . . . . . . . . . .
3  . . . . . . . . . . . . . . . . . . . .
4  . . . . . . . . . . . . . . . . . . . .
5  . . . . . . . . . . ~ . ~ . . . . . . .
6  . . . . . . . 8 . . ~ . ~ . . . . . . .
7  . . . . . . . . ~ . . . . . . . . . . .
8  . . . . . . . 1 . . . . . . . . . . . .
9  . . . . . ~ . . . ~ ~ . . 4 ~ . . . . .
10 . . . . . . ~ . . 0 a . ~ 6 . . . . . .
11 . . . . . . . . . . . ~ . ~ 9 . . . . .
12 . . . . . . . . . ~ 2 . . . ~ . . . . .
13 . . . . . ~ ~ . 5 . ~ 7 . . . . . . . .
14 . . . . . . . ~ . . . ~ . . 3 . . . . .
15 . . . . . . . . . . . . . . . . . . . .
16 . . . . . . . . . . . . . . . . . . . .
17 . . . . . . . . . . . . . . . . . . . .
18 . . . . . . . . . . . . . . . . . . . .
19 . . . . . . . . . . . . . . . . . . . .
```

Figure 2.4: NBeacons Example Scenario: **a** represents the agent, integers between 0-9 represent beacons B0-B9, and $\sim$ represents sand pits

[1]The terms *domain* and *environment* are used interchangeably

agent-at(Curiosity,(10,10))	deactivated(B0)	sand-at((8,7))
beacon-at(B0,(9,10))	deactivated(B1)	sand-at((0,5))
beacon-at(B1,(7,8))	deactivated(B2)	sand-at((0,6))
⋮	⋮	⋮
beacon-at(B9, (14,11))	deactivated(B9)	sand-at((14,12))

Table 2.1: Set of atoms corresponding to example state S_0 in Figure 9.2

```
1: move-east(Curiosity,(10,10),(11,10))
2: move-north(Curiosity,(11,10),(11,9))
3: move-east(Curiosity,(11,9),(12,9))
4: move-east(Curiosity,(12,9),(13,9))
5: move-south(Curiosity,(13,9),(13,10))
6: activate-beacon(Curiosity,(13,10),B6)
```

Figure 2.5: Plan to achieve goal activated(B6) from state S_0

2.1.2 Hierarchical Task Network Planning

Hierarchical Task Network (HTN) is a planning paradigm that uses domain-specific knowledge about tasks to guide planning search. HTN planning has been shown to be useful in many real-world domains, in part due to the naturalness of encoding the domain knowledge by knowledge engineers (Nau, 2007). Knowledge is encoded within methods which decompose tasks into subtasks. A *task* is a symbolic representation of an activity in the world. Tasks can be primitive or compound. *Primitive* tasks are accomplished by operators (e.g., *activate-beacon*).

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. An example compound task in the Marsworld[1] domain is the *navigate* task (Figure 2.6). The corresponding method, *move*, achieves the *navigate* task by using a heuristic to pick the next direction to move until the agent arrives at its destination. In the example in Figure 2.6 the heuristic knows the destination is south of the agent and returns subtasks to move south and then to recur on the navigate task. The *move-south* task is a primitive task that decomposes to the move-south operator.

Head move
Params ?agent, ?dest
Pre ¬agent-at(?agent, ?dest)
Subtasks [(move-south, ?agent), (navigate, ?agent, ?dest)]

Figure 2.6: Method for *navigate* Task where destination is south of the agent

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 $((r_1, ..., r_d, t_2, ..., t_n), S, D)$.

State-Variable Representation

All of the Hierarchical Task Network (HTN) planning in this thesis was implemented with the SHOP planner (Nau et al., 1999), specifically 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).

2.2 Goal Driven Autonomy

Goal Driven Autonomy (GDA) is a conceptual model of an agent that performs goal reasoning in addition to planning and execution (Muñoz-Avila et al., 2010a; Klenk et al., 2013; Cox, 2007). It is part of a larger class of agents that have goal reasoning capabilities (Aha et al., 2015; Hawes, 2011). Such capabilities include mechanisms to create goals, select goals, and transform goals (Cox and Veloso, 1998). GDA agents perform goal reasoning in response to anomalous events or behavior. GDA agents make use of a four step cycle that first detect some kind of anomaly, then seeks to explain why the anomaly occurred, uses the explanation to formulate a new goal, and finally decides which goal(s) to pursue. This four step process is what generally identifies an agent as a GDA agent. A major underlying motivation for GDA agents is that, in the face of an anomaly, it may be better to change one's goal then replan. A diagram of a GDA agent is shown in Figure 2.7.

The four step GDA process is shown within the Controller. Since a GDA agent is also an agent that can plan and execute those plans, it has other components (e.g components to perceive, act, and plan). The diagram of Figure 2.7 shows multiple components outside of the GDA process. These are often found in GDA agents, because they enable planning and execution; once an agent has planning and execution, it may benefit from the GDA process.

Figure 2.7 makes use of an environment model M_Σ shown in the upper left. An environment model consists of the agent's actions and states. The state transition system Σ is where actions are executed. After an action is executed the agent will perceive the subsequent state. Often GDA agents start in some initial state s_0 with some initial goal g_0. The planner (shown at the top) receives the model of the environment (actions the agent can take and a state transition function, some state s (which is likely the current state) and a goal g. The planner then returns a plan p with corresponding expectations X.

As the GDA agent executes actions, the discrepancy detector component of the controller checks the expectations X against the current state s. If a discrepancy d is found, it is sent to the next GDA process of explanation. The explanation generator generates an explanation e, for the goal formulator to use in generating one or more goal(s) g to pursue. Then the goal manager updates the current goal g. At this point, the new goal g will be sent to the planner and the current plan will be updated with the new plan produced by the planner. As long as the discrepancy detector does not detect any anomalies, the GDA processes of explanation generation, goal formulation, and goal manager do not generally run. Discrepancy detection often starts the GDA process. Also, G_p represents the pending goals of the agent (the first pending goal is the initial goal g_0).

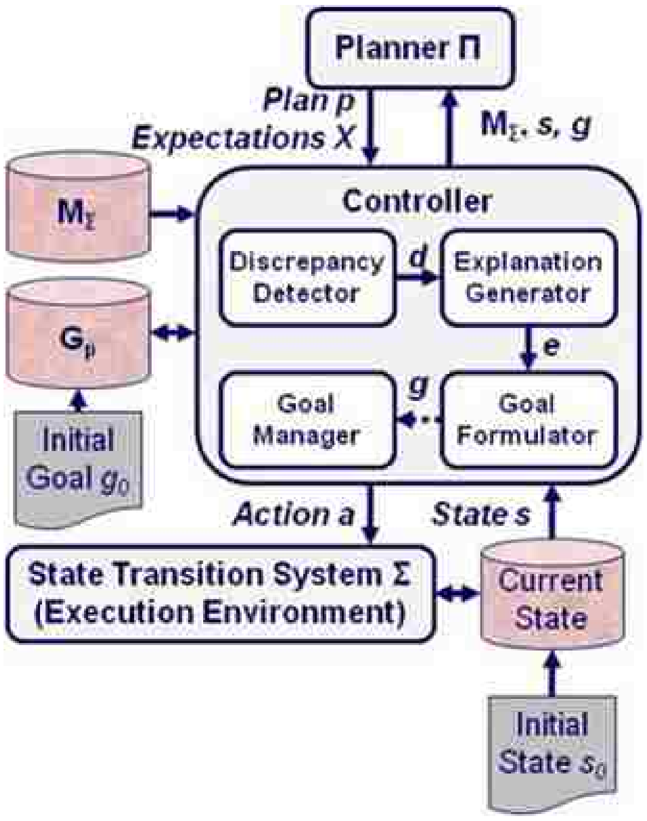


Figure 2.7: A basic model of a Goal Driven Autonomy agent (Molineaux et al., 2010)

For example, when discrepancies occur, the GDA process will result in potentially new alternative goals. An example, adapted from Molineaux et al. (2010), involves an agent performing navy operations. A naval convoy is in route to deliver some equipment and along the way an escort vessel identifies an unknown contact. At this point the agent could pursue one of multiple alternative goals including (1) abort the mission and route back the vessels to the departing port, (2) hold the convoy and send escort vessels to identify the contact.

2.3 Expectations

From the literature on planning and execution, given a plan, there are three kinds of expectations that can be generated: immediate, state, and goal regression. Informed expectations, one of the primary contributions of this work, is described in detail in chapters 4 and 5.

The first form of expectations are *immediate expectations*. Given a plan $\pi = (a_1...a_m)$, the *immediate expectation*, $EX_{imm}(S, \pi) = (pre(a_m), \mathit{eff}(a_m))$. 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.

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).

Chapter 3

Literature Review

The following literature review is a summary of papers related to discrepancy detection within the scope of goal driven autonomy agents. Much of the literature around detection of unexpected situations mostly focuses on the response to the detection as opposed to the detection itself.

The success of the planning field within the AI community has largely focused on planners for static, closed-world domains. Adapting these planners to dynamic, open worlds opens many challenges including how to discern if the plans execution is continuing as expected and how to react when this is not the case. A crucial step in dealing with these challenges is detecting discrepancies. Discrepancy detection is the mechanism by which the agent comes to know something unexpected has occurred, usually preventing the agent from reaching its current goal(s). The following literature review surveys the breadth of approaches used for discrepancy detection.

The focus of this thesis and subsequently, this literature review, is concerned with intelligent agents and the difficulty that arises from planning and execution in dynamic, complex domains. Such domains offer many challenges to independent agents, and are increasingly in demand as robotics continues to pervade real-world environments. Most agents in the work surveyed make use of a planner to accomplish their goals. A planner, in general, is a mechanism for reaching a goal state in an environment via a (partially or totally) ordered set of steps: concrete and simple actions that can be executed directly. As the agent executes these actions it performs discrepancy detection. Due to the dynamic nature of these domains, changes may occur at any time that prevent the agent from accomplishing its goals.

Discrepancy detection is the mechanism by which the agent comes to know something unexpected has occurred, usually preventing the agent from reaching its current goal(s). Discrepancy detection relies on knowledge of what is expected behavior, which comes in different forms. Vattam et al. (2013) describes an overview of various types of discrepancy detection performed by different agents. Table 3.1 lists the type of discrepancy detection, source of knowledge for performing discrepancy detection, and lists the papers within this review which use that form of discrepancy detection. While these types of discrepancy detection do not contradict each other, it so happens that the papers surveyed here tend to only use one form of discrepancy detection: using the preconditions and effects of actions from plan.

Dealing with dynamic, complex, open worlds is difficult. Discrepancy detection is the first step to enable planning endowed agents to execute their plans in these domains. Adapting these planners to dynamic, open worlds is not trivial. Talamadupula et al. (2010) describe three ways these planners are trivially adapted to open worlds: (1) blindly assuming that the world is indeed closed, (2) deliberately "closing" the world by acquiring all the missing knowledge before planning, and (3) accounting for all contingencies during planning by developing conditional plans. None of these solutions are ideal. Assuming a closed world as in (1) requires frequent re-planning during execution and can also lead to highly suboptimal plans in the presence of conditional goal rewards. In big open worlds, it is infeasible or impossible to acquire all knowledge in order to "close" the world as in (2). Lastly, contingency planning mentioned in (3) is known to be impractical in propositional worlds with bounded indeterminacy - which is the case in open worlds where the number of objects and their types are unknown. The discrepancy detection approaches in this paper are more realistic attempts to enable planning capable agents the ability to operate in these increasingly real-world domains.

The literature review is organized as follows: Section 3.1 provides a brief summary of the approaches used in Table 3.1. Section 3.2 discusses work around detecting failures for a specific plan and goal. Section 3.3 discusses the problem of detecting and adjusting planning search for discrepancies that occur at planning time. Section 3.4 discusses work dealing with detecting predicted failures. Section 3.5 discusses work on detecting anomalies not associated with a plan. Section 3.6 discusses domain-based approaches while Section 3.7 discusses object-based approaches. Finally, Section 3.8 summarizes the review, particularly noting various nuances and trade-offs of performing discrepancy detection as well.

3.1 A Classification of Plan Execution Monitoring Systems

As shown in Table 1, Vattam et al. (2013) describes five approaches to discrepancy detection. First, *plan monitoring* is using information from the plan to detect anomalies. Discrepancies can be detected by checking the preconditions and effects before

Discrepancy Detection Approach	**Source of Knowledge**	**Papers**
Plan Monitoring	Planner; preconditions, effects of actions	(Ayan et al., 2007), (Benson and Nilsson, 1993), (Fox et al., 2006), (Sugandh et al., 2008)
Periodic Monitoring	Whole environment at intervals	(Rao and Georgeff, 1995)
Expectation Monitoring	Agent's specific knowledge base	(Bouguerra et al., 2007), (Veloso et al., 1998), (Kurup et al., 2012), (Cox, 2007)
Domain-based Monitoring	Model of the environment	(Coddington et al., 2005), (Hawes et al., 2011)

Table 3.1: Different Approaches to Discrepancy Detection

and after actions are executed. Often the agent's planner returns both a plan and corresponding expectations to be checked at each step of the plan. There has also been work done to monitor the optimality of plans compared to alternative plans; if the current plan is deemed suboptimal or predicted to fail, an alternative plan can be chosen.

Second, *periodic monitoring* is defined as an agent that monitors the environment and uses that as a knowledge source for managing goals. Essentially the agent checks the environment at designated intervals. Periodic monitoring is frequently used in real-time systems.

Third, *expectation monitoring* is using knowledge about past experiences to create expectations and using them to monitor parts of the environment like in Veloso et al. (1998). Agents also use models of the environment (can be learned) to identify normal vs. anomalous behavior.

Fourth, *domain-specific monitoring* is monitoring specific variables in the state, and testing those values during plan execution. This approach is similar to *expectation monitoring*, except *expectation monitoring* is generally specific to a plan where as domain-specific monitoring can be used to trigger new goals to be formed at any time (Coddington et al., 2005).

3.2 Detecting Plan and Goal Failure During Execution

Much work on discrepancy detection is concerned with obstacles that cause task failure. This section provides an overview of these specific approaches. The common theme among these approaches is that they are all focused on the current plan (and goal) of the agent and they generate expectations using planning knowledge[1] A non-exhaustive list of approaches are shown in Table 3.2. The papers listed in the third column use the approach described.

3.2.1 Using Dependency Graphs for Plan Repair

Ayan et al. (2007) present a system coined HOTRiDE that detects plan failures and repairs plans using a novel technique involving a dependency graph. The dependency graph is a data structure that links the effects of an action in a plan satisfying the preconditions of later actions in the plan. These links explicitly show which actions depend on other actions. Thus, when an action fails it is possible to know what other parts of the plan will be affected, and those parts not affected. This is done by starting at the failed action and traversing its forward links in the dependency graph. This way the agent can determine what parts of the plan need to be re-planned for as well as what parts of the

[1]The work by Bouguerra et al. (2007) uses additional knowledge about the domain, but the initial expectations are derived from plan domain knowledge.

Approach	**Description**	**Cited Works**
Individual Action Expectations	Before executing an action, check that the preconditions of the action hold; After executing an action, check the effects are true in the environment; During action execution check to see if alive conditions hold (optional)	(Sugandh et al., 2008) (Ayan et al., 2007) (Bouguerra et al., 2007)
Informed Expectations	Build up cumulative effects from all previous actions thus far	(Dannenhauer and Muñoz-Avila, 2015b)
Goal Regressed Expectations	Regress over all preconditions and effects for each step in the plan leading up to the goal	(Fritz and McIlraith, 2007)
State Based Expectations	Use whole states, stored during the planning process, and compare at execution time against the perceived state	(Cox et al., 2012) (Klenk et al., 2013) (Fox et al., 2006)

Table 3.2: Plan and Goal-based Discrepancy Detection Techniques

plan can be reused. In this way, the agent could save time re-planning as well as having more optimal plans.

The authors evaluate their agent in a simulation environment that randomly decides to fail an action. The discrepancy detection approach here simply checks to see if the current action has failed. They test the agent against a baseline system using only the SHOP planner Nau et al. (1999) that simply re-plans from the point of failure, which may be suboptimal. The results show that by repairing a plan HOTRiDE is able to spend less time planning and maintain optimal plans. An optimal plan is one which has the smallest amount of steps in order to achieve the goal.

The authors run three experiments: (1) SHOP and HOTRiDE compared against each other, each given a number of goal tasks to be accomplished. Results show HOTRiDE can revise more often than SHOP, causing more goals to be accomplished. (2) They modified their domain to add special knowledge ('bookkeeping') and SHOP essentially performed at roughly the same competency as HOTRiDE. Important because it shows HOTRiDE does not need that special domain knowledge. (3) The third experiment looked at the average number of actions needed to accomplish goals. Here, HOTRiDE keeps modifications to an optimal minimum level, while SHOP often re-planned previously achieved goals and trying to re-achieve them.

To conclude, the length of the plans found by HOTRiDE in the experiments was always optimum which indicates that HOTRiDE preserves the goals accomplished previously and makes minimum number of changes in the original plan. The discrepancy detection approach here is an example of checking individual actions.

3.2.2 Teleo-Reactive Agents

Benson and Nilsson (1993) describe a breadth of work for an autonomous agent architecture that: (1) quickly reacts to environmental situations taking goals into account, (2) selects goals to achieve in the presence of multiple, competing goals, (3) plans new action routines, and (4) learns effects of actions to be used later by the planner. For the purpose of this literature review, only how the agent detects and respond to failures is considered.

This work makes use of a Teleo-reactive (TR) planning and acting paradigm. *Teleo* refers to the ability for actions to be influenced by the goals of the agent while *reactive* refers to the ability for agents to respond quickly (within A.I. in general, reactive agents are not usually goal oriented). The agent's planning component is composed of a TR program library, where TR programs are collections of rules of the form condition $\rightarrow$ action. A condition refers to a variable in the state, and an action is similar to a STRIPS style operator. An example rule might contain a condition that the agent's distance from an object of type *bar* is less than some value x, and the corresponding action would be *pickup-bar*. Rules in TR programs fire when their conditions are met. In this example, when an agent gets close enough to a bar, it will pick it up. TR programs contain many rules which are organized in a hierarchy so that rules with the weakest conditions are the rules that fire first, and when the action of that rule is achieved, it triggers a condition in a rule higher up in the hierarchy. Conditions can be conjunctive and can be created so that parallel TR subprograms can be in any order. Actions are STRIPS style but can be either durative or atomic. STRIPS operators are discrete with definite effects. In order to achieve durative STRIPS style operators, the authors introduce teleo-operators (TOPs).

Teleo-operators (TOPs) are defined for condition $\rightarrow$ action rules of TR programs. For any literal λ_i (condition) and any action a_j, a TOP can be defined, τ_{ij}, that describes the process of continuously executing a_j until λ_i becomes true. TOPs have four components:

1. name of the action, a
2. *postcondition* literal λ_i, that is the intended effect of the TOP
3. a preimage of, π, of the TOP which is the weakest condition under which continuous execution of a will eventually satisfy λ while maintaining π until λ does come true. (this is analogous to STRIPS preconditions)
4. a set of side effects S, of the TOP. A side effect is a literal, σ, which is not necessarily true at the time action a was begun but (usually) becomes true by the time λ becomes true.

The authors elaborate on side effects describing positive side effects as "roughly corresponding" to a STRIPS add list, and negative side effects to STRIPS delete list. In this way, it seems TOPs are not any different than STRIPS operators except some parts of them are finer grained to be able to be used in robotic platforms.

The authors mention that for large TR programs, it is impractical to search the entire space of condition $\rightarrow$ action rules for each time step. To handle this the authors use a heuristic to select the next condition to check. The heuristic is as follows: at each new time step, the agent remembers which condition $\rightarrow$ action rule was active in the previous time step. It assumes that the action of the last time step will remain active until it finishes successfully, at which point in the new time step, the parent condition $\rightarrow$ action rule containing the condition from the previous rule, will be checked. This allows the agent to only have to check 2 rules in each time step: to see if the previous rule is still active, and if not, to see if the parent rule is active. The authors note that this heuristic runs in constant time and works generally as well as checking all of the rules (thus this heuristic saves a lot of time). However, if the current action fails, it causes a problem for the agent, as now the constant time heuristic can no longer be applied. Also, while actions are being executed, the agent spends a little chunk of computation searching a subset of rules higher in the hierarchy. If any of those conditions happen to be met, the agent automatically switches to that condition $\rightarrow$ rule and begins executing that action. In response of failures, the authors simple say that a delay in the agent occurs due to having to search over all of the rules in order to reassess where it should begin. The authors report that since a delay in the agent is due to a failure, it is reasonable.

It is interesting because this kind of teleo-reactive approach might become very fragile if an agent with large TR programs operates in a very dynamic environment and these changes cause action failures. However, it seems similar enough to STRIPS planning that STRIPS-style (preconditions, effects, etc.) expectations can be used. In this paper, the discrepancy detection approach used is an example of individual action failure.

3.2.3 Case-based Planning in Wargus

Sugandh et al. (2008) present a plan adaptation technique for an agent that plays full Real-Time Strategy (Wargus) games. The main idea is that they use a dependency graph for plan adaptation, similar to Ayan et al. (2007). In terms of discrepancy detection, failures were detected by looking up direct features in the state or by comparing states to see if that state had changed. Unfortunately, the paper did not go into much detail about this. Here, the behaviors were procedural (meaning a function call was used to carry out the action) so postconditions (effects) were not available. This limits the ability to compute expectations based on the effects of actions. In these agents, where the effects of an action is not known (or has many possibilities), expectations are not easily computed or checked. One possibility could be to learn the set of effects an action produces, or use inference to lump many different states into a single category that can be used as an expectation..

3.2.4 Semantic Execution Monitoring on a Mobile Robot

Bouguerra et al. (2007) describe an agent that uses semantic knowledge in order to validate its expectations of its current location. For example, if it expects to be in the room *kitchen* it assumes it will be able to sense objects specific to a *kitchen* environment such as having an object of type sink and counter. The authors implemented a system, on a Magellan Pro mobile robot, that was able to validate its expectations in a house-like environment. The agent would generate information gathering plans to actively sense which objects were in which room. The author's experimental evaluation was promising, with the agent performing well in a few test scenarios.

Using semantic knowledge to draw conclusions about what is to be expected in environments is a step towards solving agents in large, complex, and open environments. However this approach is knowledge intensive: the agent needed planning domain knowledge and semantic knowledge about the environment. What is also interesting about this approach is that simply verifying expectations involved a complete, separate planning-execution cycle of generating information gathering plans to move about the localized space (e.g., kitchen, bedroom) in order to more closely observe objects that could verify an agent's location. Using this kind of semantic knowledge seems promising, especially with the rise of the semantic web, and also seems to introduce many challenging aspects.

The authors mention the following areas for future work: (1) deeper experimental validation that addresses scalability and richer domains, (2) The issue of when to engage in information gathering needs to be investigated because it may be expensive, (3) Selecting which expectations should be pursued in information gathering is important because number of expectations might be big, (4) The authors discuss related work (conducted by them) that uses a probabilistic approach that computes probabilities for different action outcomes (i.e., different locations of the robot). This approach can support a more informed decision about whether or not to do information gathering. They could use PTLPLAN, with probabilities, to compute an expected cost of the monitoring plan.

3.2.5 Plan Stability vs. Plan Repair

Fox et al. (2006) introduce a new metric, *plan stability*, which they use to decide whether to repair the current plan or re-plan from scratch. They argue that there are many situations when the agent's goals will change but will be similar enough to the previous goal to warrant reusing parts of the previous plan. They define plan stability as a measure of the difference of two plans. Given an original plan π_0 and a new plan π_1, the difference between π_0 and π_1 is the number of actions that appear in π_0 and not in π_1 plus the number of actions that appear in π_1 and not in π_0. The authors use this metric of plan stability as a modification within their local search based planner LPG to guide the search. The details of the planner are outside the scope of this document, for more information see: (Gerevini et al., 2003). The authors perform extensive experiments on four different temporal planning domains. The results showed that their technique was an improvement over re-planning in terms of speed and plan quality (generally).

Although this paper introduces a novel approach to performing plan repair for a temporal, local search based planner, they do not go into much detail about how they detect discrepancies. The only clues they give the reader is that they only assume a few literals or a few of the agents goal conditions have changed. From this, it is likely the authors are simply using a complete state matching approach: if the expected state and current state are different in any way, then a discrepancy is triggered and the authors perform plan repair. This work is an example of state based expectations.

3.3 Detecting and Responding to Failures During Planning

The process of planning may take a considerable amount of time, long enough to warrant adjusting the process in the face of changes in the environment. Veloso et al. (1998) consider how to adjust the planning process when changes from the initial start state have been detected. In this way, the changes may not yet be considered discrepancies as described earlier because the current plan (as its being formed) is taking these changes under consideration. Thus the new plan will have accounted for any changes from the time planning began until planning finishes.

The process described in the paper is composed of three steps: (1) Generate monitors for relevant features in the state, (2) Deliberate over changes from these monitors to see if the plan being constructed should change, (3) Alter the plan being constructed to take into account changes in the state. The authors only focus on (1) and (3), leaving (2) for future work.

Monitors, as in (1), are generated in two ways. The first takes into account all parts of the state related to the current best plan the planner is constructing. This includes the preconditions of every operator in the plan. Should the values of the preconditions change, then the operators in the current plan will likely be affected. The second way monitors are generated involve using features relevant to operators *not* chosen for selection in the current best plan. The idea being that if the values of those preconditions change, it may warrant switching to a better, alternative plan.

Both of the monitor types generated make use of specific classes of monitors. *Subgoal monitors* deal with the preconditions and bindings of operators. *Usability-condition monitors* are similar to subgoal monitors except they focus on usability conditions instead of preconditions. *Quantified-condition monitors* take into account preconditions containing universally quantified predicates. Essentially these monitors will observe the set of objects related to the predicate (i.e., if the predicate involves all the packages in a city, this monitor will watch to see if the number of packages changes).

When monitors identify a change in the environment, the current best plan may be altered. Plans are altered in three ways: (1) adding operators to a plan, (2) removing operators from a plan, and (3) jumping in a plan. Adding operators to the plan, (1), may be necessary if two monitors fire. If *Subgoal monitors* fire because a condition required for an operator in the current plan was made false, more actions may be needed to insert operators to enable that condition. If *quantified-condition monitors* fire then the precondition involving the quantified condition may no longer be valid. For example, if a new package appears, the condition holding-all-packages may need extra operators inserted into the plan in order to go pick up the new package.

Removing operators from the plan, (2), operates in just the opposite way as above. If some condition that was false in the state becomes true, the part of the plan that establishes that condition no longer is necessary. For example, if a package is removed from the state then the part of the plan to go pick up that package can be cut.

Jumping within the plan, (3), can happen because of any monitors, essentially whenever an alternative plan becomes more attractive then the current plan.

The authors performed experiments using Prodigy. They created an artificial planning domain using fairly simple and straightforward operators that were not related to any kind of real-life scenario. As a result, the authors were able to vary the number of operators in the solution plan from 1 to 30. The authors fix the number of binding possibilities for the operators to be 2. During planning, two monitors fire, observing change the initial state. These changes happen at planning step 0,10,20. The results showed that when the monitors fired, the planner was able to find a plan more quickly, but the later the monitors fired, the less planning savings to be had (because the planner had already done a significant amount of work).

This paper presents a very novel and important contribution to work on discrepancy detection. Being able to produce plans that have no discrepancies in the current state at the time planning finishes is extremely important for agents requiring long periods of time to plan. Without being able to modify planning, you could run into infinite cycles, where the agent starts planning, then the world state changes, planning finishes, and when the agent goes to execute the plan it fails, and must restart planning.

The only drawback to this paper was the experimental evaluation, which was quite limited. Specifically, more experiments which increase and vary the kinds of changes in the state as well as report on plan optimality would be very helpful. It would also be nice to observe what properties of domains cause more or less problems to this planning approach (i.e., benchmarking this technique on many existing planning domains). One way to do this would be to randomly choose to change the state at any given plan step in each domain. How often changes happen could also vary, seeing at what point a domain is too dynamic to plan efficiently. It may be the case that in very dynamic domains, this approach will not work because the planner may never finish planning if it is constantly adjusting due to constant changes in the state. Nevertheless, this work is a solid contribution and plays an important role in agents with non-trivial planning problems.

3.4 Detecting Future Failures

Rao and Georgeff (1995) summarize the theory of Belief Desire Intention (BDI) agents and describes an evaluation of a BDI agent that manages an air-traffic management application. BDI agents are composed of three major components: beliefs, desires, and intentions. Beliefs are different from knowledge, in that it is only what the likely state of the environment is, whereas knowledge is considered to be more absolute. Desires are the motivational aspect of the agent, essentially its goals or objectives. In the formal model presented in the paper, desires correspond to any path through the execution tree. Intentions are the course of action that the agent has decided to take via its selection function. Their theoretical discussion how to handle changes in the environment is interesting. Essentially they describe two kinds of situations that can arise. First, the environment may change when the agent is selecting which actions to execute. One way to address this is to reduce the time it takes to perform selection which may result in less than optimal selection. The second situation that can arise is when the environment changes during the course of action chosen from the selection function. The authors provide an enlightening discussion in the paper about handling the second situation: the environment changing during some course of action.

There are two straightforward approaches on either end of the spectrum. On the one side of the spectrum, as soon as the agent becomes aware of changes, it re-applies its selection function to choose a different course of action (analogous to re-planning). Alternatively the agent can ignore changes until it finishes its course of action (ignore the discrepancy outright). The authors Rao and Georgeff discuss this spectrum in further detail. If a system always chooses one of these strategies (re-plan or ignore) it will result in a negative outcome. It is likely that recomputing the choice of action for every action is expensive but unconditional commitment would cause the agent to fail to achieve its objectives as its actions become invalid (due to changes in the environment). The authors then assume that if "potentially significant changes can be determined instantaneously" it is possible to limit the frequency of reconsideration and thus achieve a balance of too much reconsideration and not enough. Within the BDI architecture, the *intention* component keeps track of the current course of action in order to effectively solve this dilemma. This literature review is primarily focused on how to compute expectations which are used as a shortcut to know if changes in the environment require deliberation about changing the intended course of action. The deliberation: within the GDA community this is done through explanation and goal formulation.

The paper provides a theoretical foundation for discrepancy detection and the role it plays in intelligent agents. A critique of this paper is the assumption that "potentially significant changes can be determined instantaneously". For large, complex domains this may very well not be the case. In the paper's evaluation, the BDI system is composed of multiple agents: an aircraft agent for each arriving aircraft, and global agents including a sequencer, wind modeler, coordinator, and trajectory checker. Within this system discrepancy detection was performed by the sequencer agent using hard-coded knowledge. For example, the sequencer agent (responsible for coordinating arriving aircraft) would only change its commitment to its current course of action if the action completed (all aircraft landed) or the agent does not believe one of the current aircraft will reach its destination by its current ETA. This is essentially an expectation based on predictions of the environment given current knowledge now. The agent periodically samples the environment to re-evaluate its current course of action using these hard-coded expectations.

3.5 Non-Plan-based Anomaly Monitoring

3.5.1 Classifying Other Agent's Behavior

Kurup et al. (2012) performs discrepancy detection on image sequences of pedestrians walking in an urban setting in order to classify human behavior as normal or suspicious. This work is important as it shows that, in general, machine learning and classification techniques can be used to generate expectations for processes unrelated to the agent in the environment. The classification performed here is specific to the ACT-R Cognitive Architecture. Their technique used a combination of partial-matching and blending mechanisms[2], which was able to generate future states it had never seen before. The results showed that their learning algorithm was better when it came to detecting multiple features in the environment, but a kNN classifier was better for single behavior detection. Therefore in more complex environments, there technique is likely to perform better than kNN classification.

[2]Because these mechanisms are unrelated to planning, many of the details are not reported here, see Kurup et al. (2012) for more information

3.5.2 Detecting Knowledge Discrepancies

Cox (2007) presents novel work dealing with explaining knowledge failures in a story understanding domain, including the ability for the agent to use the same mechanism to understand its own reasoning failures. A motivation of this work is that if an agent is to be able to operate continuously and independently, an introspective component is necessary. This work shows how a preliminary system is able to determine its own goals by interpreting and explaining unusual events or states of the world. The resulting goals seek to minimize the difference between its belief state and the world state. The specific hurdle addressed in this work is for the agent to decide if it should do something or learn something when there is an anomaly. The system generates goals to choose which comprehension method (Case-based Reasoning, explanation generation, etc.) to use to understand stories. When something anomalous or interesting is detected, the system explains it and incorporates it to fit with the current understanding of the story. It classifies actions as interesting and then tries to explain why someone would perform that action[3].

3.6 Domain-specific Monitoring

3.6.1 Internal vs. External Motivations

Coddington et al. (2005) explores two different methods of using motivations in a goal-directed agent. The agent uses the stochastic and anytime planner ADAPT_LPG. Whenever a new goal is generated, it is passed to the planner, and the planner re-plans considering both the remaining current plan and the new goal. There are six specific mo-

[3]The paper does not go into any detail about how interesting and therefore anomalous actions are described, as the main focus is on explanation

tivations the paper mentions, four of which rely on thresholds and the other two use specific knowledge about the environment.

The first method the authors use are motivations that generate explicit goals, which are then sent to the planner. For example, the *conserve-energy* motivator generates a goal *recharge-battery* when a threshold of battery level is reached. The explicit goal recharge-battery is then sent to the planner, which plans to achieve the current goal and the new goal. The second method the authors mention is encoding the threshold knowledge in the planning knowledge so that the planner will only create plans that will not violate the thresholds using expected values for how long actions will take.

The authors only give a qualitative description of the agent's results but essentially there are pros and cons to each approach. Motivators that trigger explicit goals when values in the state exceed thresholds is nice for continuous autonomous behavior, in that the agent will always be generating its own goals (especially if one threshold is based on time). This is likely important in situations like the Mars rover, where it may have limited communication with the source of its externally imposed goals. The second approach, encoding thresholds in planning knowledge makes the planning problem harder (two of the three planners the authors tested could not find plans and the third planner produced a plan that was inefficient). However the third planner (SGPlan) did produce a plan in which no thresholds were violated. There are some nuances to this problem, for example some motivator thresholds are more resource-critical than others which affected how easy it was for the planners solve the planning problem. However, a major benefit of modeling mission-critical motivators implicitly in the planner is safer plans. When the battery-resource motivator is modeled implicitly in the planner, the planner cannot generate plans that will violate the battery threshold (i.e., if a plan gets long enough, it will insert actions to go recharge the battery and continue). The alternative is that the planner simply generates plans to achieve goals, and during execution the agent must monitor its battery level: when it falls below the threshold it will send a new goal to the planner to go recharge.

This is important work for discrepancies that are known to happen, are predictable (i.e., the agent can predict how much battery it uses), and discusses the trade-offs between encoding motivators explicitly vs. implicitly within the planner. The authors describe multiple areas for future research: (a) dynamically adjusting threshold values and (b) deciding which goals to pursue among a set of goals (over-subscription planning).

3.6.2 Reasoning with Explicit Knowledge Gaps

In Hawes et al. (2011), a mobile robot named Dora is given an incomplete tour by a human in an unknown environment. Dora is a robot that has autonomous behavior resulting from the use of a goal generations and management framework, a planner, and a spatial map-related knowledge base. Dora has drives (similar to motivators in Coddington et al. (2005)) that are used by goal generators to generate goals. The particular drive of interest in this work is the drive to have correct spatial knowledge of the environment. This drive works with the goal generators to generate goals such as *explore* and *categorize* each room in an apartment-like setting. Dora is also able to generate goals to *patrol*, where Dora will go to previously visited ares to validate its knowledge.

The focus of this work from a discrepancy detection perspective is detecting faults or gaps in Dora's knowledge base. This is different than monitoring directly for actions being in the state, but could be used in conjunction with validating actions (like how semantic and map knowledge is used to interleave planning and execution in Bouguerra et al. (2007)).

3.7 Summary and Discussion

Discrepancy detection is an essential and difficult problem for agents operating in dynamic and complex environments. There are multiple choices of where to perform discrepancy detection within the agent (during plan execution, during planning), what to apply it to in the environment (agent's own actions, external agents, external processes), and trade-offs that need to be taken into consideration. In this chapter we've looked at ways of monitoring the current plan and goal for possible failures: (Ayan et al., 2007), (Benson and Nilsson, 1993), (Sugandh et al., 2008), (Bouguerra et al., 2007), (Fox et al., 2006); directing sensing and responding to changes while planning is happening: (Veloso et al., 1998); detecting future failures using prediction: (Rao and Georgeff, 1995); detecting anomalies unrelated to the current plan: (Kurup et al., 2012), (Cox, 2007); and finally, domain-specific monitoring: (Coddington et al., 2005), (Hawes et al., 2011).

Due to the complex and agent-infrastructure dependent nature of discrepancy detection, interesting and non-trivial trade-offs and other issues arise:

- For agents equipped with explanation capabilities, discrepancy detection shares a trade-off with explanation. At one extreme, an agent with an instantaneous and perfectly accurate explanation system could simply explain every possible change between the perceived state and expected state, allowing a trivial exact-matching comparison implementation of discrepancy detection. However, explanation is often very expensive, requiring depth-limited search and may not always be accurate. Therefore using such techniques as described in the work survey is warranted, and can be taken in the perspective of moving some of the explanation capabilities into the discrepancy detection mechanism.

- We have seen that deliberation and reasoning within the discrepancy detection process itself can be beneficial (i.e., (Bouguerra et al., 2007) use semantic knowledge to validate STRIPS style effects via information-gathering planning and execution). When an agent observes a change in the environment that could be a potential future obstacle, reasoning over the perceived obstacle may be beneficial: If the obstacle is likely to remain in the environment and will prevent the current plan from achieving its goal, the agent may be better off deciding to change its course of action even though it could still execute part of its plan. However, if there is a chance the obstacle may be removed on its own, the agent may decide to continue. Thus, a discrepancy detection mechanism could have a deliberation component dealing with impending obstacles.

- In a similar light to dealing with future obstacles is dealing with informed and goal-regressed discrepancies. Suppose an effect from a previous action in the current plan was needed for the goal, and that effect is no longer true in the environment. The agent is now faced with the following options: (1) assume the effect will not return to the environment and take some action to either re-plan in order to achieve that effect and continue with the current goal, or change its goal altogether, or (2) complete the rest of the actions in its plan, and then check that all the goal conditions are met. If there is a likelihood that the effect has only temporarily been nullified, then the agent could continue. However, if the agent reaches the goal and the effect is still false, then the agent has now wasted resources executing the actions between when the effect was made false and reaching the goal. For long plans, this becomes an important trade-off to take into account and demonstrates another example of deliberation used within discrepancy detection.

- Most work on discrepancy detection does not take into account that sensing the environment is usually an action itself, that has both a direction (what part of the state to observe) and a time (when to observe it) cost. Veloso et al. (1998) consider the direction cost during the planning process by generating monitors for specific features in the state and only looking at those monitors; however they assume that those features can be observed anytime.

Part II

Contributions

Chapter 4

Informed Expectations from HTN Planners

Correctly identifying a violation of 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.

Research on GDA agents has explored two kinds of expectations thus far (Muñoz-Avila et al., 2010a; 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 expectations 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 tasked with creating a signal where a valid method of achieving the task is to activate beacons (the domain described here is called Marsworld^1 and described in more detail in Section 9.1.2.) In Marsworld^1, there are external factors which cause beacons to be disabled. When the goal of the agent is to activate *n* beacons, and one of them becomes disabled after the agent has activated it, the plan can fail to achieve the goal. An agent employing *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 the change. Therefore, immediate expectations fail to detect relevant changes of actions' effects if the changes occur after that action has already been verified.

The second discrepancy detection 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 (subsuming immediate expectations) 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 unexpected differences 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 (such as GDA) to correct such a false discrepancy.

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

Section 9.2.1 shows an empirical validation in two variants of domains used in the GDA literature. These experiments evaluate a GDA agent using informed expectations versus alternative GDA agents that either check immediate effects or check for expected states. Our experiments demonstrate improved performance of a GDA agent using informed expectations.

4.1 Formalization of Informed Expectations

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 $\xi' = (\xi \triangleright pre(a_1)) \triangleright$ $\mathit{eff}(a_1)$ 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.

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.

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.

Algorithm 1 Informed Expectations (extensions to the SHOP algorithm are underlined)

```
 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:             $\underline{\text{ex}[t] \leftarrow (ex_{prev} \triangleright pre(q)) \triangleright \textit{eff}(q)}$
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:             $(\pi_t, \underline{\text{ex}[t]}) \leftarrow$ 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
```

4.2 Discussion

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. Here 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 shown in Section 9.2.1.

This 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.

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.

Chapter 5

Informed Expectations for Sensing

In partially observable and dynamic environments it may not be obvious what the agent needs to know in its current state to achieve its goals. In the previous chapter we described informed expectations for dynamic environments, this work extends informed expectations for environments that are not only dynamic but also partially observable. We use partial observability as referring to both that the agent may not know the true values of atoms in the state and that the agent must explore the state to see what objects it can use to complete its goals. In domains such as these, agents need to perform sensing actions (including movement) to obtain the true value of atoms in the domain. However, since the domain is dynamic, those truth values may change over time. We assume there is a cost to sensing (unlimited sensing would remove partial observability and in that case we can just apply the previously discussed informed expectations) and show that by using informed expectations we achieve the lowest sensing costs of other expectation techniques.

Previous work on GDA agents Molineaux et al. (2010); Weber et al. (2012); Jaidee et al. (2011); Shivashankar et al. (2013) does not address sensing actions with associated costs. In contrast, researchers have long observed that acquiring knowledge about the state of the world can be expensive both in terms of running time to complete the tasks and in resource consumption (Knoblock, 1995). For example, virtual agents might require to plan the information gathering tasks including which information sources to access and what information to acquire, which in turn will enable the agent to identify other information sources (Knoblock, 1996); physical agents may use sensors which require power, time, and potentially other resources (Mei et al., 2005).

This extension of informed expectations integrates ideas of information-gathering agents into the GDA framework. In particular, we present a new family of GDA agents that adopt the convention of distinguishing between planning actions and sensing actions, the latter of which have associated costs. We investigate how different kinds of expectations affect the performance of GDA agents in environments that are partially observable and dynamic.

We evaluate agents where observability is limited in such a way that the agent cannot generate a grounded plan because it does not know which future actions will be available until it explores more of the environment. Section 9.2.2 contains the experimental evaluation and results.

5.1 Related Work

Deterministic (STRIPS) planning assumes that actions have a pre-determined outcome (Fikes and Nilsson, 1971). The result of planning is a sequence of actions that enable the agent to achieve its goals. A Markov Decision Process (MDP) is a frequently studied planning paradigm whereby actions have multiple outcomes (Howard, 1960). In MDPs, solutions are found by iterating over the possible outcomes until a policy is generated which indicates for every state that the agent might encounter, what action to take that will enable the agent to achieve its goals. A Partial Observable Markov Decision Process (POMDP) is an extension of MDP for planning when the states are partially observable (Kaelbling et al., 1995, 1998). In POMDPs, solutions are found by iterating over the possible states that the agent believes itself to be in and the possible outcomes of the actions taken on those belief states until a policy is found over the belief states. In GDA agents, goals may change while the agent is acting in the environment. Previous GDA research has used both plans (i.e., sequences of actions, Molineaux et al. (2010) and policies (Jaidee et al., 2012).

Research in GDA has resulted in techniques to learn GDA knowledge automatically; this includes research to learn goals and goal formulation knowledge (Jaidee et al., 2011) and learn explanations (Weber, 2012). Researchers have also explored applying GDA for playing computer games (Weber et al., 2010; Dannenhauer and Muñoz-Avila, 2013), for conducting naval operations and for controlling robots (Roberts et al., 2014) among others. Thus far, GDA work has not considered explicit models of the cost of sensing actions; examining the state is assumed to have no cost for the agent. Our work is the first to use GDA with an explicit model of partial observability that accounts for the cost of sensing actions. Furthermore, current GDA research assumes that enough information in the state is observable to plan ahead a sequence of grounded actions to achieve the goals. In our work we drop this assumption presenting a model that accounts for situations when such planning is not always possible (while at the same time not precluding this possibility).

There is a long-standing tradition of interleaving planning and execution (Goldman et al., 1996). Briefly, Sage was an early system interleaving planning and execution to enable the system to further plan when possible while, in parallel, sensing actions are executed (Knoblock, 1995). This enables the agent to cope with information gathering tasks including query execution for data integration (Ives et al., 1999). Interleaving planning and execution has also been used for multi-agent systems architectures (Paolucci et al., 1999). More recently, there has been a renewed call about the importance of interleaving planning and execution (Ghallab et al., 2014). In those works, the agents gather information in order to achieve predefined goals. In contrast, GDA agents might change its goals.

5.2 Problem Definition

We define the problem of sensing in a partially observable and dynamic environment whereby a variety of goals can be pursued. First, if Q is the collection of all states in the world, then a collection of atoms s is a *partial state* if there is a state $q \in Q$ such that $s \subseteq q$ holds. The 6-tuple input to the guiding sensing problem is defined as $(S, \Sigma, s_0, G, \phi_g, c)$ and is composed of the following elements:

S: The set of all partial states; if Q is finite, then S is finite too. $\mathbf{s_0}$: An initial partial state. **G**: A collection of goals.

Σ: Actions in the domain. $\Sigma = \Sigma_{plan} \cup \Sigma_{sense}$, where Σ_{plan} are planned actions in the domain (e.g., move the agent to a neighboring location from its current location). An action $a \in \Sigma_{plan}$ consists of the usual triple, $(prec(a), a^+, a^-)$ indicating the preconditions, positive effects and negative effects of a. Σ_{sense} consists of sensing actions; one, γ_τ, for every condition τ that may be satisfied in the environment. γ_τ returns $\{true, false\}$ depending if τ is a valid condition in the environment (e.g., checks if a specific beacon is activated).

$\phi_{\mathbf{g}}$: A heuristic function, one for each goal $g \in G$. It maps for every state, the next action to take, $\phi_g : S \rightarrow \Sigma_{plan}$. The heuristic function ϕ_g can be seen as encoding a strategy about how to achieve goal g. In our model the heuristic functions do not need to provide "hints" of actions to take that somehow circumvent the partial observability in the domain. However, our model does not preclude that possibility: if a plan π to achieve goal g is known, then the heuristic ϕ_g would simply pick the next action in π from the current state. It also does not preclude the case where we have a policy (i.e., a mapping from states to actions) indicating which action to take when visiting an state. In such a case when the goal changes, the plan/policy must change. In our empirical evaluation, we do not assume that we have knowledge about such plans/policies.

c: Sensing Cost Function, $c : \Sigma_{sense} \rightarrow \mathbb{R}_{\geq 0}$. Returns a non-negative number for each sensing action.

The guiding sensing problem is defined as follows: given a guiding sensing problem $(S, \Sigma, s_0, G, \phi_g, c)$, generate a sequence of actions $\pi = (a_1...a_m)$, each $a_k \in \Sigma$, and a sequence of partial states $(s_0...s_m)$ such that:

1. If $\pi_{plan} = (a_{k1}...a_{kn})$ denotes the subsequence of all planning actions in π (i.e., each $a_{kj} \in \Sigma_{plan}$), then the preconditions of each a_{kj} were valid in the environment at the moment when a_{kj} was executed.

2. One or more goals $g \in G$ hold in s_m.

3. If $\pi_{sense} = (a_{k1}...a_{kz})$ denotes the subsequence of all sensing actions in π (i.e., each action in π_{sense} is of the form $\gamma_\tau()$ for some condition τ), then the total sensing cost $C(\pi) = \sum_{i=1}^{n} c(a_{ki})$ is minimal.

Condition 1 guarantees that the actions taken while the agent was acting in the environment were sound. Condition 2 guarantees that at least one of the goals is achieved.

This condition is compatible with the special case of oversubscription planning (Smith, 2004), where the agent tries to achieve the maximum number of goals. It is also consistent with GDA where the agent chooses the goals to achieve as a result of situations encountered while acting on the environment. Condition 3 represents an ideal condition where the agent minimizes the cost of sensing while achieving its goals. In our work, we explore a variety of criteria (see next section) that affects the overall costs of the action trace pursued by the agent but our algorithms will provide no guarantees that Condition 3 is met.

5.3 GDA and Expectations

GDA agents repeatedly perform the following four steps as described in the introduction. We walk through these steps again with an example in the Marsworld^2 domain example here:

(1) **Discrepancy detection:** after executing an action a, the agent compares the *observed state* o after sensing and the agent's expectation x (i.e., it tests whether any constraints are violated, corresponding to unexpected observations). In the marsworld^{2}ew domain an expectation is the atom *activated(beacon5)* and a discrepancy is the observed contradicting atom *deactivated(beacon5).* If a discrepancy D is found (e.g., $D = x \setminus o$ is not empty), then the agent executes the following step.

(2) **Explanation generation:** Given an observed state o and a discrepancy D, this step hypothesizes an explanation e causing D.

(3) **Goal formulation:** This step generates a goal $g \in G$ in response to D, e, and o.

(4) **Goal management:** Given a set of existing/pending goals $\hat{G} \subseteq G$ (one of which may be the focus of the current plan execution) and a new goal $g \in G$, this step may

update $\hat{G}$ to create $\hat{G}'$ (e.g., $\hat{G}' = \hat{G} \cup \{g\}$) and will select the next goal $g' \in \hat{G}'$ to pursue.

The GDA cycle is triggered when discrepancies occur. In turn, discrepancies hinge on the notion of an agent's expectations. We explore 5 kinds of expectations from the literature (Muñoz-Avila et al., 2010a,b; Dannenhauer and Muñoz-Avila, 2015b; Mitchell et al., 1986), adapted to consider the guiding sensing problem. We use the following conventions: $\pi_{prefix} = (a_1...a_n)$ is the sequence of actions executed so far, $(s_0...s_n)$ are the sequence of partial states the agent believes to have visited so far, and $a_{n+1} \in \Sigma_{plan}$ is the next action to be executed.

1. *No expectations:* The agent performs sensing actions that are only related to the preconditions of a_{n+1} where $a_{n+1} \in \Sigma_{plan}$. For every precondition $\tau \in a_{n+1}$, the agent performs the sensing action of $\gamma_\tau()$.

2. *Immediate expectations:* The agent performs sensing actions for both the preconditions and effects of a_{n+1} where $a_{n+1} \in \Sigma_{plan}$.

3. *Eager expectations:* The agent checks if the belief state s_n is consistent with the current observed state and if s_{n+1} is consistent with the state observed after executing $a_{n+1} \in \Sigma_{plan}$.

4. *Informed expectations:* $Inf(\pi_{prefix}, s_0)$ move forward all valid conditions computed so far in π_{prefix}. Informed expectations are formally defined as follows: *Inf*$(\pi_{prefix}, s_0) = Inf(\pi_{prefix}, s_0, \emptyset)$

   - $Inf((), s, cc) = cc$.
   - $Inf((a), s, cc) = cc'$ if (1) $a \in \Sigma_{plan}$ and is applicable in s, then $cc' =$

$(cc \setminus a^-) \cup a^+$, where a^- are the negative effects from a and a^+ are the positive effects from a.

- $Inf((a_k a_{k+1} ... a_n), s_k, cc) = Inf((a_{k+1} ... a_n), s_{k+1}, Inf((a_k), s_k, cc))$.

5. *Goal-regression expectations:* Given a plan suffix $\pi_{suffix} = (a_{k+1} ... a_m)$ achieving a collection of goals G from some state, goal regression expectations *Regress*(π_{suffix}, G) is formally defined as follows:

   - $Regress((), cc) = cc$.
   - $Regress((a), cc) = (cc \setminus a^+) \cup precs(a))$, where $precs(a)$ are the preconditions of a.
   - $Regress((a_{k+1} ... a_m), cc) = Regress((a_{k+1} ... a_{m-1}), Regress((a_m), cc))$.

Checking for no expectations is typical of systems performing deliberative planning, where the agent's actions are not executed in the environment and therefore cannot fail. In a dynamic environment, actions may become invalid and hence an agent using no expectations is prone to fail to achieve its goals. Immediate expectations is an improvement in that the agent checks if the conditions for the next action to be applied hold in the observed state. But if earlier conditions are no longer valid, then the agent will be unaware (e.g., a beacon needed to achieve a goal condition has become deactivated). Hence, immediate expectations is also prone to fail to fulfill its goals.

Eager expectations check that all conditions in the agent's belief state are satisfied at every iteration. Hence, they are an improvement over the previous two kinds of expectations in that checking for eager expectations guarantees that if these conditions are valid, the plan is still valid (e.g., it will continue checking if a previously activated beacon, needed to achieve a goal, is still active). A drawback is that it may incur high sensing costs; it will also check for conditions that are not relevant for the current plan (e.g., any atom in the state, even if irrelevant to the current goal, will be checked and the agent will incur in the corresponding sensing costs). In contrast, goal regression expectations are an improvement in that they guarantee that the expectations computed, *Regress*(π_{suffix}, G), are minimal. That is, if any condition in *Regress*(π_{suffix}, G) is removed then some precondition in the suffix plan π_{suffix} is no longer applicable and therefore G cannot be fulfilled. The main drawback is that some of these conditions might become irrelevant if the agent needs to replan which is prone to occur in dynamic environments. Informed expectations addresses this limitation in that they move forward all conditions validated by the plan, π_{prefix}, executed so far. We state the following property implying advantages and disadvantages of informed expectations over goal regression expectations: Let s_0 be a state, G a collection of goals and a plan $\pi = \pi_{prefix} \bullet \pi_{suffix}$ achieving G from s_0 (where $\bullet$ is the concatenation of the two plans). Under these conditions, if *Inf*$(\pi_{preffix}, s_0)$ are applicable in a state s_k, then *Regress*(π_{suffix}, G) is also applicable in s_k but not the other way around.

This follows from the fact that *Regress*(π_{suffix}, G) computes the minimal conditions and our assumption that $\pi_{prefix} \bullet \pi_{suffix}$ achieves G from a state s_0. This result means that an agent checking for *Inf*(π_{prefix}, s_0) will check for unnecessary conditions assuming that there is no need to replan after executing π_{prefix}. On the other hand, if there is a need to replan to achieve the same goals G, then the informed expectations, $Inf(\pi_{prefix}, s_0)$, will compute the needed conditions regardless of how the plan is completed. In contrast, *Regress*(π_{suffix}, G) conditions might no longer be valid.

5.4 Formalism

Algorithm 1 shows the pseudo-code for our agent that is operating in a partially observable and dynamic environment. The main algorithm GDA starts on Line 18. The algorithm uses the following variables, initialized in Line 2: a plan π (initially empty), a collection of states (initially consisting of the starting partial state s_0), a default goal, g, and a collection of goals that the agent is currently pursuing $\hat{G}$. It also uses a global variable, G with all potential goals that the agent might pursue. The algorithm also uses an expectation function, $X(s, \pi)$, as defined in the previous section (excluding goal regression expectations).

The algorithm returns the pair $(\pi, \hat{S})$, where π is the trace of all planning and sensing actions executed and $\hat{S}$ all states the agent believes it visited (depending on the kind of

Algorithm 2

```
1:  **Global Variables** $\hat{S}, \Sigma, s_0, \hat{G}$,G, $\phi_g$, $X : S \times \Pi \rightarrow S$
2:  $\pi \leftarrow ()$; $\hat{S} \leftarrow (s_0)$; $g \leftarrow$ *defaultGoal*; $\hat{G} \leftarrow \{g\}$
3:  **return** *GDA()*
4:
5:  **procedure** CHECK($x, s, \pi$)
6:      $D \leftarrow \emptyset$; $s' \leftarrow s$                                   ▷ initialization
7:      **for** $\tau \in x$ **do**
8:          $cond \leftarrow \gamma_\tau()$                                        ▷ Sensing Action
9:          **if** ($positive(\tau)$ and $not(cond)$) or ($negative(\tau)$ and $cond$) **then**   ▷
    Discrepancy!
10:             $D \leftarrow D \bullet (\tau))$
11:             $\pi \leftarrow \pi \bullet \gamma_\tau()$
12:             **if** $positive(cond)$ **then**
13:                 $s' \leftarrow s' \setminus \{\tau\}$                          ▷ $\tau$ is not valid in $s$
14:             **if** $negative(cond)$ **then**
15:                 $s' \leftarrow s' \cup \{\tau\}$                               ▷ $\tau$ is valid in $s$
16:     **return** $D$
17:
```

```
procedure GDA()
    $s \leftarrow lastState(\hat{S})$
    if $terminationCondition(\hat{G}, s, \hat{S}, \pi)$ then
        return $(\pi, \hat{S})$
    $a \leftarrow \phi_g(s)$                                    ▷ selects applicable action
    $D \leftarrow check(precs(a), s, \pi)$
    if $D == \emptyset$ then
        execute(a)
        $s \leftarrow apply(a, s)$
        $\pi \leftarrow \pi \bullet (a)$
        $D \leftarrow check(X(s, \pi), s, \pi)$
        if $D == \emptyset$ then
            $\hat{S} \leftarrow \hat{S} \bullet (s)$
            return GDA()
        else $\hat{S} \leftarrow \hat{S} \bullet (s)$
    else replace$(\hat{S}, s)$
    $E \leftarrow explain(D, s)$                                ▷ There was a discrepancy
    $\hat{G} \leftarrow$formulate_new_goals$(G, \hat{G}, D, E, s)$
    $g \leftarrow$goal_selection$(\hat{G}, s)$
    return GDA()

procedure TERMINATIONCONDITION$(\hat{G}, s, \hat{S}, \pi)$
    $g' \leftarrow goalsSatisfied(s, \hat{G})$
    if $g' \neq \emptyset$ then
        if check$(g', s, \pi)$ then
            return true
        else return false
    else return false
```

expectations used, the agent may not have checked if every condition in the state is valid in the environment). The procedure begins by taking the last partial state, s, believed to be visited (Line 19). It first checks if the termination conditions are met in s. The *terminationCondition* procedure is detailed on Line 39 and is explained later. If the termination condition is not met, then the agent selects an applicable action a (based on the belief state s) to execute using the heuristic for the current goal (Line 22). It checks if the preconditions of a are valid in the environment by performing sensing actions (Line 23). The procedure *check* is detailed in Line 5. We will explain it later, but briefly, it will log in π any sensing action performed and it will modify s based on the discrepancies D it detected. If the preconditions are satisfied in the environment (Line 24), then the action is executed (Line 25), the belief state is moved forward by applying action a on s (Line 26). Action a is added into π (Line 27). Afterwards, it checks if the expectations (Line 28) are met in the environment (Line 29). If so, s is added into $\hat{S}$ and calls *GDA()* recursively (Lines 30-31). Otherwise, it adds s into $\hat{S}$ (Line 32; the procedure *check* updates s whenever there is a discrepancy).

If the preconditions are not satisfied, the last state in $\hat{S}$ is replaced with the updated state s (Line 33; calling the *check* function in Line 23 changes s based on the discrepancies detected). If there is a discrepancy either in the preconditions or in the expectations, Line 34 is reached. In it, the algorithm generates an explanation E for the discrepancy, formulates new goals $\hat{G}$ to achieve, selects a new goal g to pursue among those in $\hat{G}$ and calls GDA recursively (Lines 35-37).

We now discuss the auxiliary procedures *check* and *terminationCondition*. The *check* procedure (Line 5), receives as parameters the conditions x to be checked, the belief state s and the actions executed in the environment so far π. It checks for every atom τ in x if τ is sensed in the state (Line 8) while accounting for the fact if it is a positive or negative condition (Line 9). D maintains all discrepancies found (Lines 6 and 10). π is updated with any sensing actions performed (Line 11). The state is updated when there is a discrepancy (Lines 12-15). The procedure returns the discrepancies (Line 16). The auxiliary procedure *terminationCondition* (Lines 39-45) checks if the current goals g' (with $g' \subseteq \hat{G}$) are (1) satisfied in the belief state s (lines 40-41) and (2) satisfied in the environment (Line 42).

We present a formulation of the guiding sensing problem where sensing actions have associated costs for environments that are partially observable. Our formulation is amenable to, both, environments where GDA agents can plan ahead (e.g., as a sequence of grounded actions) and environments where this is not possible. We analyze trade-offs between five forms of expectations (i.e., none, immediate, eager, informed, and goal regression) used by GDA agents when dealing with dynamic environments. We presented an algorithm for a GDA agent operating in both partially observable and dynamic environments approximating a solution to the guiding sensing problem when planning ahead as a sequence of grounded actions is not possible. We evaluated our algorithm in two simulated environments. From this evaluation, we see that informed expectations performs the best among the four expectations (i.e., none, immediate, eager, informed) and using complete expectations (i.e., when full observability is enabled); informed expectations has less sensing costs compared to other expectations that achieve all goals.

Chapter 6

High Level Expectations in Complex Domains

A shortcoming in the current state of the art on GDA research is the lack of structured, high-level representations in their formalisms. Most rely on STRIPS representations. For example, the agents' expectations are defined as either specific states (i.e., collection of atoms) or specific atoms in the state (e.g., after executing the action move(x,y) the agent expects the atom location(y) to be in the state) expected to be true. Goals are desired states to be reached in the state or desired atoms in the state (e.g., the agent is at a certain location).

To address this shortcoming we introduce high-level ontological concepts to be used as expectations. Using an ontology with high-level concepts enhances the representation of GDA elements whereby these concepts can be directly related, and defined, in terms of low-level chunks of information. A motivation for using ontologies is that STRIPS representations are too limited to represent events that happen in the real world (or complex domains), more structured representations are needed in order to capture more complex constraints (Gil and Blythe, 2000).

Drawbacks to using ontologies include the knowledge engineering effort required for construction and maintenance of the ontology as well as the running time performance during reasoning. In past years, taking an ontological approach may have seemed intractable for systems acting in real-time domains with large state spaces. With the growth of the semantic web, description logics, and fast reasoners, ontologies may begin to become viable. As demonstrated in this work, we are reaching a point where ontologies are useful for fast-acting systems such as agents that play Real-Time Strategy games. Specifically, we investigate the use of ontologies for a GDA agent playing the Real-Time Strategy game Starcraft, in two agents we call *LUiGi* and *LUiGi-H* .

Both *LUiGi* (Dannenhauer and Muñoz-Avila, 2013) and *LUiGi-H* ((Dannenhauer and Muñoz-Avila, 2015a); discussed in more detail in Chapter 7) maintain an ontology that contain low-level facts of the environment (such as the x and y coordinates of each unit) as well as high-level concepts that can be inferred from low-level facts. One such high-level concept is the notion of controlling a region (a bounded area of the map). A player controls a region provided two conditions hold. First, the player must have at least one unit in that region. Second, the player must own all units in that region. Regions in which both players have units are considered contested. A region can only be one of unknown, contested, or controlled.

Once we have these concepts of unknown, controlled, and contested regions, we can use them as rich expectations. For example, in the beginning of the game when we are constructing the buildings and producing our first army, we expect that we control our region. If this expectation is violated (producing a discrepancy), it means that the enemy has a presence in our base early in the game. When this happens *LUiGi* explains the discrepancy and pursues a goal focused on defending its base region instead of building a considerable army. In this situation, some possible explanations include an immediate incoming rush attack or an attempt to distract our worker units. By having a concise expectation and corresponding discrepancy representing that we no longer have complete control of our base region, we can detect this change and then let the rest of the GDA cycle explain and respond. When the expectation "I control my starting region" fails during an enemy rush attack, the agent can choose a new goal to quickly defend their region. This is a higher reasoning level than many previous Starcraft and Wargus (Wargus is an open source version of Warcraft, another popular Real-Time Strategy game) bots who simply focus on optimizing micro-unit attacks. While micro-management is important, without high-level reasoning, bots are still at a disadvantage to humans. One reason for this that has been pointed out by domain experts is that the automated players are still weak in reasoning with high-level strategies (Churchill, 2013).

6.1 Ontologies for Expectations and Explanations

One of the main benefits of using an ontology with GDA is the ability to provide formal definitions of the different elements of a GDA system. The ontology used here chooses facts as its representation of atoms, where facts are represented as triples ⟨subject, predicate, object⟩. A fact can be an initial fact (e.g., ⟨unit5, hasPosition, (5,6)⟩ which is directly observable) or an inferred fact (e.g., ⟨player1, hasPresenceIn, region3⟩). By using a semantic web ontology, that abides by the open-world assumption, it is technically not possible to infer that a region is controlled by a player, unless full knowledge of the game is available. Starcraft is one such domain that intuitively seems natural to abide by the open world assumption because of fog of war. However, we can assume local closed world for the areas that are within visual range of our own units. For example, if a region is under visibility of our units and there are no enemy units in that region, we can infer the region is not *contested*, and therefore we can label the region as *controlled*. Similarly, if none of our units are in a region, then we can infer the label of *unknown* for that region.

The following are formal definitions for a GDA agent using a semantic web ontology:

- **State** S: collection of facts
- **Inferred State** S^i: $S \cup \{$ facts inferred from reasoning over the state with the ontology $\}$
- **Goal** g: a desired fact $g \in S^i$
- **Expectation** x: one or more facts contained within the S^i associated with some action. We distinguish between primitive expectations, x_p, and compound expectations, x_c. x_p is a primitive expectation if $x_p \in S$ and x_c is a compound expectation if $x_c \in (S^i - S)$. $(S^i - S)$ denotes the set difference of S^i and S, which is the collection of facts that are strictly inferred. In other words, a compound expectation x_c is an inferred fact.
- **Discrepancy** d: Given an inferred state S^i and an expectation, x, a discrepancy d is defined as:

    1. $d = x$ if $x \notin S^i$, or
    2. $d = \{x\} \cup S^i$ if $\{x\} \cup S^i$ is inconsistent with the ontology

- **Explanation** e: Explanations are directly linked to an expectation. For primitive expectations, such as $x_p = (player1, hasUnit, unit5)$ the explanation is simply the negation of the expectation when that expectation is not met: $\neg x_p$. For compound expectations, x_c (e.g., expectations that are the consequences of rules or facts that are inferred from description logic axioms), the explanation is the trace of the facts that lead up to the relevant rules and/or axioms that cause the inconsistency.

6.1.1 Example of Explanation

Assume we have the following description logic axiom (6.1) and semantic web rules (6.2) and (6.3):

$$KnownRegion \equiv DisjointUnionOf(ContestedRegion, ControlledRegion) \tag{6.1}$$

The class KnownRegion is equivalent to the disjoint union of the classes ContestedRegion and ControlledRegion. This axiom allows the ontology to infer that if an individual is an instance of ContestedRegion it cannot be an instance of ControlledRegion, and vice-versa. This also encodes the relationship that ContestedRegion and ControlledRegion are subclasses of KnownRegion.

$$Player(?p) \land Region(?r) \land Unit(?u) \land isOwnedBy(?u, ?p) \land isInRegion(?u, ?r) \rightarrow hasPresenceIn(?p, ?r) \tag{6.2}$$

With this rule, the ontology reasoner can infer that a player has a presence in a region if that player owns a unit that is located in that region.

$$Player(?p1) \land Player(?p2) \land DifferentFrom(?p1, ?p2) \land Region(?r) \land hasPresenceIn(?p1, ?r) \land hasPresenceIn(?p2, ?r) \rightarrow ContestedRegion(?r) \tag{6.3}$$

This rule allows the inference of a region to be an instance of ContestedRegion if two different players each have a presence in that region.

Figure 6.1 shows an example where the unsatisfied expectation is ⟨player0, controlsRegion, regionA⟩ in which the explanation is that regionA is contested. The explanation trace begins with the primitive facts of each player owning one or more units and those units' being located in regionA. Using rule (6.2), the next level of the tree is the inferred facts: ⟨player0, hasPresenceIn, regionA⟩ and ⟨player1, hasPresenceIn, regionA⟩. Now using rule (6.3) with the second level inferred facts, we infer that ⟨regionA, instanceOf, ContestedRegion⟩. From this level, the expectation ⟨player0, controlsRegion,

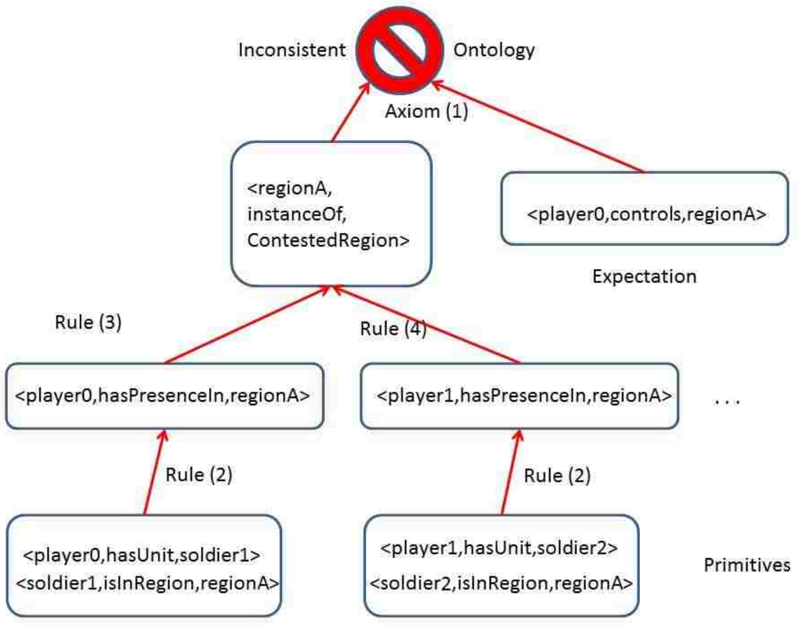

Inconsistent
Ontology
Axiom (1)
<regionA, instanceOf, ContestedRegion>
<player0,controls,regionA>
Expectation
Rule (3)
Rule (4)
<player0,hasPresenceIn,regionA>
<player1,hasPresenceIn,regionA>
. . .
Rule (2)
Rule (2)
<player0,hasUnit,soldier1>
<soldier1,isInRegion,regionA>
<player1,hasUnit,soldier2>
<soldier2,isInRegion,regionA>
Primitives

Figure 6.1: Inconsistency Explanation Trace

regionA⟩ conflicts with the inferred fact ⟨regionA, instanceOf, ContestedRegion⟩. Axiom (6.1) produces an inconsistency because a region can not be both a contested region and a controlled region (disjointUnionOf). The inconsistency produced by inserting the expectation into the ontology and subsequently reasoning over the ontology is how the agent knows an expectation has been violated.

In this example, the explanation was that the region was contested, and the trace both provides an explanation and shows how the ontology reasoner is able to produce an inconsistency. An inconsistency is the result of the unmet compound expectation ⟨player0, controlsRegion, regionA⟩. Given the explanation, a viable goal to pursue next would be to build a stronger group of units to attack the enemy units in regionA. As discussed before, an alternative explanation could have been that the player does not control the region because it did not have any of its units in the region. In that case, the inconsistency in the ontology would result from regionA being labeled as *Unknown* and a viable goal would be to travel to the region along a different path or a different medium (flying vs. ground units).

High-level concepts with an ontology provide an abstraction over low-level facts and conditional relationships among the facts. For example, we can either have a group of facts such as ⟨player0, hasUnit, unit1⟩, ⟨player0, hasUnit, unit2⟩, ⟨unit1, isAtLocation, (x1,y1)⟩, ⟨unit2, isAtLocation, (x2,y2)⟩, ⟨player1, hasUnit, unit3⟩, ⟨unit3, isAtLocation, (x3,y3)⟩ accompanied with the conditional relationships equivalent to the description logic rule (1) and rules (2) and (3) or we can have the high-level concept *Contested Region* accompanied with the ontology. The conditional relationships are needed for both being able to count how many units each of the players has in the region, and being able to infer what region each unit is located in based on its (x,y). Such numerical conditions cannot be represented in STRIPS. The explanation trace shows how the concepts are built from the low-level facts and the complex (often hierarchical) relationships among them (via description logic axioms and rules). Even if such an equivalent representation would be possible using STRIPS, representing such hierarchical knowledge in STRIPS could require a large number of in STRIPS atoms, thereby making reasoning on these domains very inefficient (Lotem and Nau, 2000).

The explanation trace makes apparent the conditional relationship details and facts of which the high level expectation (i.e., ContestedRegion) is composed. This allows the root causes of the unmet expectation to be identified. The benefit is richer expectations and explanations being able to express not only atoms (facts) but conditions and relations on those atoms as well.

6.2 System Details for *LUiGi* & *LUiGi-H*

LUiGi is implemented in two components, a bot and a GDA component, shown in Figure 6.2. The bot component interfaces directly with the Starcraft game engine to play the game. The GDA component runs as a separate program, and connects to the bot component via a TCP/IP connection with both the bot and GDA component running on the same computer. The bot is responsible for building up an initial base and executing tasks sent to it by the GDA component. The bot component contains knowledge of how to execute individual plan steps and knowledge of when a task has finished so that it can request the next plan step from the GDA component. During gameplay the bot dumps all of its game state data every n frames (the evaluation scenarios in section 9.2.3 use $n = 20$), making it available for the GDA component to use during ontological reasoning.

The bot component is implemented in a C++ DLL file that is loaded into Starcraft and the GDA component is implemented in a Java standalone application. Running the reasoner, Pellet (Sirin et al., 2007), over the ontology takes on average between 1-2 seconds. Such an amount of time would normally be unacceptable for micro-management strategies controlling individual units in battle, but for more general strategies, such as what units to produce next and how to attack the enemy, the following scenarios demonstrated that this amount of time is acceptable when played at the speed setting used in human vs. human tournaments. This is because executing a strategy involving unit production and movement takes anywhere from 20 seconds to minutes in these scenarios.

LUiGi (and *LUiGi-H*) is composed of two major components, the controller and the bot. The controller handles the goal reasoning processes while the bot interfaces with the game directly. The controller and bot operate separately from each other, and communicate via a socket and file system. There are two methods of data transfer between the controller and the bot. First, every n frames the ”bot” dumps all visible gamestate data to the controller via a file (visible refers to the knowledge that a human player would have access to; the bot does not have global knowledge; the bot is bound by fog of war just as is a human player). The controller then uses this data to populate the semantic web ontology described previously, in which to reason about the game to infer more abstract conclusions. The other method of data transfer is the controller sending messages to the bot which happens via a socket. Both the bot and controller run as completely different processes, use their own memory, and are written in different languages (bot is c++ and controller is java).

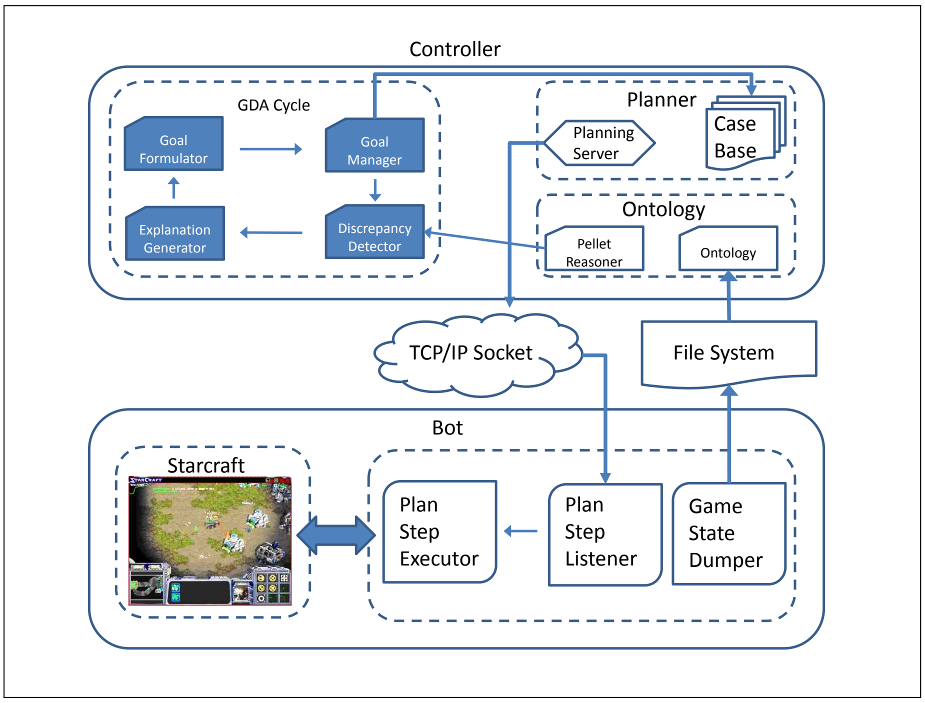


Figure 6.2: *LUiGi-H* Overview

The controller’s perspective of the game is different than the bot’s in a few ways.

The controller's game state data is only updated when the Pellet reasoner finishes. The Pellet reasoner is one of a few easily available reasoners for semantic web ontologies. However, the controller's game state data includes more abstract notions such as "I control region x right now". The controller also knows all current actions being executed. As a result, the controller has an overall view of the match but at the loss of some minute details, such as the exact movements of every unit at every frame of the game. This level of detailed information is perceived by the bot but at cost of only having a narrow, instant view of the game. The bot receives actions from the controller, it only receives a single action per plan at a time (when that action finishes, successfully or not, the bot requests the next action of the plan). The bot can execute multiple actions together independently, without knowing which action is going to come next. If the controller decides an action should be aborted while the bot is executing it, it sends a special message to the bot instructing it to stop executing that action. Example actions might be ”Produce 3 Seige Tanks” or ”Move units 53, 55, and 56 to region4”.

6.3 Discussion

The results of *LUiGi* demonstrate the capability of using an ontology within discrepancy detection and are shown in section 9.2.3. By using an ontology, more general planning actions like ”produce units”, ”move units”, ”attack region”, ”defend region”, etc, can be defined with atomic facts as preconditions and effects, like STRIPS operators. The ontology we used is described in Appendix **??**. For example, ”produce units” can have a single high level fact (compound expectation) of ”I control my base region” using the controlRegion class, as a precondition. Similarly, ”attack region” can have a single high level fact of ”controlRegion” as an effect. Wide variations can exist in units' specific x,y coordinates, number of troops remaining, etc. and still be useful (because of reasoning with the ontology) an effect of the planning action can be represented as a single high level fact. Therefore, such use of an ontology described here enables STRIPS like planning actions with preconditions and effects at a high level to be grounded in a complex domain like Starcraft, where many different game states could mean an action failed or succeeded. As such, the kinds of expectations mentioned in related works, such as using a single entire state as an expectation for the result of an action, would not enable this kind of high level strategic planning.

Chapter 7

A Hierarchical Model for GDA

7.1 LUiGi-H

LUiGi-H is an extension of *LUiGi* that uses hierarchical plans and hierarchical expectations. *LUiGi-H* uses Case-based Reasoning for its approach to plans (using a case-base of plans). Case-based reasoning (CBR) has been shown to be an effective method in GDA research. CBR alleviates the knowledge engineering effort of GDA agents by enabling the use of episodic knowledge about previous problem-solving experiences. In previous GDA studies, CBR has been used to represent knowledge about the plans, expectations, explanations and new goals (e.g..,(Dannenhauer and Muñoz-Avila, 2013; Jaidee and Muñoz-Avila, 2013; Weber, 2012)). A common trait of these works is a plain (non-hierarchical) representation for these elements.

LUiGi-H uses episodic GDA knowledge in the form of hierarchical plans that reason on stratified expectations and explanations modeled with ontologies (as described in Chapter 6). Hierarchical representations enable modeling of stronger concepts thereby facilitating reasoning of GDA elements beyond object-level problem solving strategies on top of the usual (plain) plan representations.

Crucially, both *LUiGi-H* and its baseline *LUiGi* include the same primitive plans. That is, they have access to the same space of sequences of actions that define an automated player's behavior. Hence, any performance difference between the two is due to the enhanced reasoning capabilities; not the capability of one performing actions that the other one could not. For planning from scratch, HTN planning has been shown to be capable of expressing strategies that cannot be expressed in STRIPS planning (Erol et al., 1994). But in this work, plans are not generated from scratch (our systems don't even assume STRIPS operators); instead, plans are retrieved from a case library (those expressiveness results do not apply here).

It is expected that *LUiGi-H* will require increased computation time due to higher level expectations. We test the performance of both *LUiGi-H* and *LUiGi* on the real-time strategy game: Starcraft. Hence, both systems experience a disadvantage if the computation time during reasoning (i.e., planning, discrepancy detection, goal-selection, etc.) is too large. Increased computation time manifests as a delay in the issuing of macro-level strategy (i.e., changing the current plan) to the game-interfacing component of the agent.

7.2 Example

Figure 7.1 shows a hierarchical plan or h-plan used by *LUiGi-H* . This plan, and every plan in the case base, are composed of the primitive actions found in Table 7.1 at the lowest level of the h-plan (we refer to the lowest level as the 0-level plan). The h-plan achieves the *Attack Ground Surround task*. For visualization purposes, in Figure 7.2, we divide the h-plan into two bubbles A and B. Bubble A achieves the two subtasks

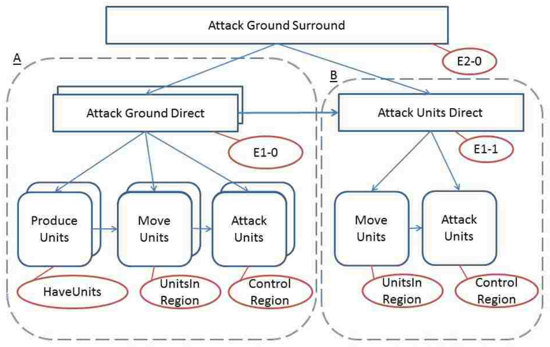


Figure 7.1: High Level Plan: Attack-GroundSurround

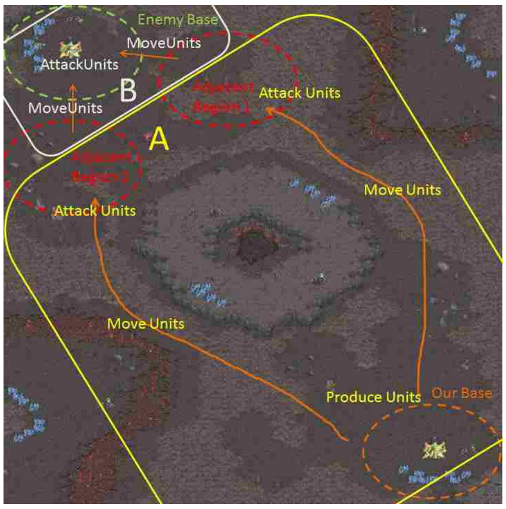


Figure 7.2: AttackGroundSurround on map Volcanis

Attack Ground Direct (these are the two overlapping boxes) while Bubble B achieves the *Attack Units Direct* task. For the sake of simplicity we don't show the actual machine-understandable representation of the tasks. In the representation, the two *Attack Ground Direct* tasks would only differ on the parameters (one is attacking region A while the other one is attacking region B as illustrated in Figure 7.2).

Bubble A contains the two Attack Ground Direct tasks, each of which is composed of the actions: Produce Units, Move Units, and Attack Units. Bubble B contains the task Attack Units Direct which is composed of the actions: Move Units, Attack Units. This h-plan generates an Attack Ground Surround plan for each region surrounding the enemy base. In the example on the map shown in Figure 2, this happens to be two regions adjacent to the enemy base, therefore the plan contains two Attack Ground Direct that are executed concurrently.

Once the execution of both Attack Ground Direct tasks are completed, the agent's units will be in regions adjacent to the enemy base. At this point, the next task Attack

Units Direct is executed, which moves the units into the enemy base and attacks. Reasoning using a more abstract plan such as this one requires representing the notion of surrounding. This is only possible because of *LUiGi-H* 's use rich expectations. Specifically, the expectation labeled *E1-0* in Figure 7.1 represents the condition that all regions that were attacked are controlled by the player attacking (Table 7.2). In the ontology, the explicit notion of Region Surrounded can be inferred for a region if all of that region's adjacent regions are controlled by the agent (represented by Control Region). In this example there are only two Attack Ground Direct because there are only two adjacent regions to the enemy base). In Figure 7.1 each bubble contains the expectation for its corresponding task. For the primitive tasks or actions, the expectations are as shown in Table 1. For the expectations of tasks at higher levels in the plan, such as for *Attack Ground Direct*, the expectation indicates that our units are successfully located in regions adjacent to the enemy base. Only after this expectation is met, then the agent proceeds to *Attack Units Direct* task (denoted by B in Figure 7.1).

Action	**Pre-Expectations**	**Post-Expectations**
Produce Units	1. Control Home Base	1. Our player has the given units requested
Move Units	1. Control Home Base	1. Our units are within a given radius of the destination
Attack Units	None	1. We control the given region
Attack Worker Units	None	1. We control the given region

Table 7.1: Primitive Actions and Corresponding Expectations

Expectation	Description
E1-0	Control all of the regions from Attack Ground Direct
E1-1	Control region from Attack Direct
E2-0	Control same region as in E1-1

Table 7.2: High Level Expectations used in Attack Ground Surround

7.3 GDA in *LUiGi-H*

These four steps of the GDA process are shown in *LUiGi-H* in Figure 6.2. Discrepancy detection plays an important role as the GDA cycle will not choose a new goal unless an anomaly occurs, and the first part of the process is identifying such an anomaly. The baseline *LUiGi* system solved the problem of mapping expectations to primitive plan action such as Produce Units, Move Units, and Attack Units by using an ontology.

The crucial difference between *LUiGi* and *LUiGi-H* is that *LUiGi* performs the GDA cycle on level-0 plans. That, is on the primitive tasks or actions such as Produce Units and their expectations (e.g., *Have Units*). In contrast, *LUiGi-H* reasons on expectations at all echelons of the hierarchy. The next sections describe details of the inner workings of *LUiGi-H* .

7.4 Representation Formalism and Semantics of h-plans

LUiGi-H maintains a library of h-plans. h-plans have a hierarchical structure akin to plans used in hierarchical task network (HTN) planning but, unlike plans in HTN planning, h-plans are annotated with their expectations. In HTN planning only the concrete plan or level-0 plan (i.e., the plan at the bottom of the hierarchy) has expectations as determined by the actions' effects. This tradition is maintained by existing goal-driven autonomy systems that use HTN planners. For example, Muñoz-Avila et al. (2010a) uses the actions' semantics of the level-0 plans to check if the plans' expectations are met but does not check the upper layers. *LUiGi-H* is the first goal-driven autonomy system to combine expectations of higher echelons of a hierarchical plan and case-based reasoning.

These h-plans encode the strategies that *LUiGi-H* pursues (e.g., the one shown in Figure 1). Each case contains one such h-plan. We don't assume the general knowledge to be given to generate HTN plans from scratch. Instead, we assume a CBR solution, whereby these h-plans have been captured in the case library. For example, they are provided by an expert as episodic knowledge. This raises the question about how we ensure the semantics of the plans are met; HTN planners such as SHOP guarantee that HTN plans correctly solve the planning problems but require the knowledge engineer to provide the domain knowledge indicating how and when to decompose tasks into subtasks (i.e., methods). In addition, the STRIPS operators must be provided. In our work, we assume that the semantics of the plans are provided in the form of expectations for each of the levels in the h-plan and an ontology Ω that is used to define these expectations.

We define a task to be a symbolic description of an activity that needs to be performed. We define an action or primitive task to be a code call to some external procedure. This enable us to implement actions such as "scorched earth retreat U to Y" (telling unit U to retreat to location Y while destroying any buildings or units along the way) and the code call is implemented by a complex procedure that achieves this action while encoding possible situations that might occur without worrying about having to declare each action's expectations as *(preconditions, effects)* pairs. This flexibility is needed for constructing complex agents (e.g., an Starcraft automated player) where a software library is provided with such code calls but it would be time costly and perhaps unfeasible to declare each procedure in such library as an STRIPS operator. We define a compound task as a task that it is not defined through a code call (e.g., compound tasks are decomposed into other tasks, each of which can be compound or primitive).

Formally, an h-plan is defined recursively as follows.

Base case. A level-0 plan π_0 consisting of a sequence of primitive tasks. Each primitive task in the level-0 plan is annotated with an expectation. *Example:* In Figure 7.1 the level-0 plan consists of 8 actions: the produce, move, attack sequence is repeated twice (but with different parameters; parameters are not shown for simplicity) followed by the move and attack actions. Each task (shown as a rectangle) has an expectation (shown as an ellipse).

The base case ensures that the bottom level of the hierarchy consists exclusively of primitive tasks and hence can be executed.

Recursive case Given a plan π_k of level k (with $k \geq 0$), a level$-k+1$ plan, π_{k+1}, for pi_k consists of a sequence π_{k+1} of tasks such for each task t in π_{k+1} either:

(d1) t is a task in π_k, or

(d2) t is decomposed into a subsequence $t_1 ... t_m$ of tasks in π_k. *Example:* In Figure 1, the task Attack Ground Direct is decomposed into the produce, move, attack primitive tasks.

Conditions (d1) and (d2) ensure that each task t in level $k+1$ either also occurs in level k or it is decomposed into subtasks at level k.

Finally, we require the each task t in the π_{k+1} plan to be annotated with an expectation e_t such that:

(e1) if t meets condition (d1) above, then t has the same expectation e_t for both π_k and π_{k+1}.

(e2) if t meets condition (d2) above, then t is annotated with an expectation e_t such that $e_t \models_\Omega e_m$, where e_m is the expectation for t_m. That is, e_m can be derived from e_t using the ontology Ω or loosely speaking, e_t is a more general condition that e_m. *Example:* The condition *control region* can be derived from condition **E1-0** (Table 7.2).

An h-plan is a collection $\pi_0, \pi_1, ..., \pi_n$ such that for all k with $(n-1) \geq k \geq 1$, then π_{k+1} is a plan of level $(k+1)$ for π_k. *Example:* the plan in Figure 1 consists of 3 levels. The level-0 plan consists of 8 primitive tasks starting with *produce units*. The level-1 plan consists of 3 compound tasks: *Attack ground direct* (twice) and *attack unit direct*. The level-2 plan consists of a single compound task: *Attack Ground surround*.

A case library consists of a collection $hp_1, hp_2, ..., hp_m$ where each hp_k is an h-plan.

GDA with h-plans. Because *LUiGi-H* uses h-plans the GDA cycle is adjusted as follows: discrepancies might occur at any level of the hierarchy of the h-plan. Because each task t in the h-plan has an expectation e_t, then the discrepancy might occur at any level-k plan. Thus the cycle might result in a new task at any level k. This is in contrast to systems like HTNbots-GDA (Muñoz-Avila et al., 2010a) where discrepancies can only occur at level-0 plans. When a discrepancy occurs for a task t in a level k-plan, an explanation is generated for that discrepancy, and a new goal is generated. This new goal repairs the plan by suggesting a new task repairing t while the rest of the k-level plan remains the same. At the top level, say n, this could mean retrieving a different h-plan. This provides flexibility to do local repairs (e.g., if unit is destroyed, send a replacement unit) or changing the h-plan completely.

Execution of level-0 plans The execution procedure works as follows: each action t_i in the level-0 plan is considered for execution in the order that it appears in the plan. Before executing an action the system checks if the action requires resources from previous actions. If so it will only execute that action if those previous actions' execution is completed. For example, for the level-0 plan in Figure 1, the plan will begin executing *Produce Units* but not *Move Units* since they share the same resource: the units that the former produce are used by the latter. The second *Produce Units* action will start if it has more than one fabric. Otherwise, it will need to wait until the first *Produce Units* is completed. The other levels of the h-plan are taken into account when executing the level-0 plan. For example, the action *Move Units* in the portion B of the plan will not be executed until all actions in the portion A are completed because the compound task *Attack Units Direct* occurs after the compound task *Attack Ground Direct*. As a result of this simple mechanism, some actions will be executed in parallel while still committing to the total order of the h-plan.

7.5 Basic Overview of *LUiGi-H* 's Components

Here we give a brief overview of each component; we also go into more detail for the Planner in Section 7.5.1. The ontology is the same ontology as described in Chapter 6.

The Planner. As explained in Section 7.5.1, an expert-authored case base is composed of h-plans that encode high-level strategies. Actions are parametrized, for example, the action Produce Units takes a list of pairs of the form (unit-type, count) and the bot will begin to produce that number of units of each type given. All expectations for tasks in the current h-plan are inferred using the ontology Ω, which include all facts in the current state and new facts inferred from the rules in the ontology.

Ontology: The ontology represents the current state of the game at any given point in time. It is refreshed every n frames of the Starcraft match, and contains facts such as regions, unit data (health, location, etc.), player data (scores, resources, etc.). The state model is represented as a semantic web ontology.

Bot: Component that directly interfaces with Starcraft to issue game commands. This component dumps game state data that is loaded into the ontology and listens for actions from the Goal Reasoner.

Game State Dumper: Component within the Starcraft Plan Executor that outputs all of the current game state data to a file which is then used to populate the ontology of the State Model of the controller.

Plan Action Listener: The bot listens for actions from the controller, and as soon as it receives an action it begins executing it independently of other actions. It is the job of the controller to ensure the correct actions are sent to the bot. The bot only has knowledge of how to execute individual actions.

7.5.1 Planner

While actions in a level-0 plan are the most basic tasks in the context of the h-plans, these actions encode complex behavior in the context of the Starcraft games. For example, the action Produce Units takes multiple in-game commands to create the desired number and type of units (i.e., 5 Terran Marines). These include commands to harvest the necessary resources, build required buildings, and issue an individual command to build each unit. Each action is parametrized to take different arguments. This allows general actions to be used in different situations. For example, Produce Units is used to produce any kind of units, while Move Units is used to move any units to any region. Table 7.3 list the plans in our case base.

Name	Description	Actions
Attack Ground Direct	Produce ground units and attack the enemy base directly	Produce Units (marine, x) Move Units (enemy base) Attack Units (enemy base)
Attack Air Direct	Produce air units and attack the enemy base directly	Produce Units (marine, x) Move Units (enemy base) Attack Units (enemy base)
Attack Both Direct	Produce air units and attack the enemy base directly	Produce Units (marine, x) Produce Units (wraith, x) Move Units (enemy base) Attack Units (enemy base)
Attack Ground Surround	Calculates the location of each region surrounding the enemy base, send units to that location, and then attacks the enemy	Attack Ground Direct (xR) —Produce Units —Move Units —Attack Units Attack Ground Direct (enemy base)
Attack Air Surround	Calculates the location of each region surrounding the enemy base, send units to that location, and then attacks the enemy	Attack Air Direct (xR) —Produce Units —Move Units —Attack Units Attack Ground Direct (enemy base)
Rush Defend Region	Take all units from a previous plan and defend the home base	Acquire Units (unit ids list) Move Units (home base) Attack Region (enemy base)
Attack And Distract	Attack directly with ground units while at the same time attacking from behind with air units which focus specifically on killing worker units	Attack Air Sneak Attack Ground Direct
Attack Air Sneak	Fly units directly to nearest corner of the map in regards to the enemy base before sending to enemy base	Produce Units (wraith, x) Move Units (nearest corner) Move Units (enemy base) Attack Worker Units (enemy base)

Table 7.3: Plans

7.6 Discussion

To the best of our knowledge, *LUiGi-H* is the first agent to use episodic hierarchical plan representation in the context of goal-driven autonomy where the agent reasons with GDA elements at different levels of abstraction. Nevertheless there are a number of related works which we will now discuss.

Other GDA systems include GRL (Jaidee and Muñoz-Avila, 2013) and EISBot (Weber, 2012). As with all GDA systems, their main motivation is for the agents to autonomously react to unexpected situations in the environment.

The most closely related works are the one from Muñoz-Avila et al. (2010a) and Molineaux et al. (2010); Molineaux and Aha (2013); Shivashankar et al. (2013), which describe the HTNbots and the ARTUE system respectively. Both HTNBots and ARTUE use the HTN planner SHOP (Nau et al., 1999). SHOP is used because it can generate plans using the provided HTN domain knowledge. This HTN domain knowledge describes how and when to decompose tasks into subtasks. Once the HTN plan is generated, HTNBots and ARTUE discard the k-level plans ($k \geq 1$) and focus their GDA process on the level-0 plans (i.e., the sequence of actions or primitive tasks). That is expectations, discrepancies, explanations, all reason at the level of the actions. There are two main difference versus our work. First, in our work we don't require HTN planning knowledge. Instead, *LUiGi-H* uses episodic knowledge in the form of HTN plans. Second, *LUiGi-H* reasons about the expectations, discrepancies and explanations at all levels of the HTN plan; not just at the level 0. As our empirical evaluation demonstrates, reasoning about all levels of the HTN plans results in better performance of the GDA process compared to a GDA process that reasons only on the level-0 plans.

Other works have proposed combining HTN plan representations and CBR. Included in this group are the PRIAR (Kambhampati and Hendler, 1992) Caplan/CbC system (Muñoz et al., 1995), Process manufacturing case-based HTN planners (Chang et al., 2000) and the SiN system (Muñoz-Avila et al., 2001). None of these systems perform GDA. They use CBR as a meta-level search control to adapt HTN plans as in PRIAR or to use episodic knowledge to enhance partial HTN planning knowledge as in SiN.

LUiGi-H is a GDA agent that combine's CBR episodic knowledge, h-plan knowledge and ontological information that enables it to reason about the plans, expectations, discrepancies, explanations and new goals at different levels of abstraction.

We compared *LUiGi-H* against the ablated version, *LUiGi* and the results are shown in 9.2.4. Both agents use the same case base for goal formulation and have access to the same level-0 plans. In our experiments, *LUiGi-H* outperforms *LUiGi* demonstrating the advantage of using episodic hierarchical plan representations over non-hierarchical ones for GDA tasks. We noted one match where *LUiGi-H* lost because of a delay in ontology reasoning time that caused discrepancy detection to respond too slowly to an attack on *LUiGi-H* 's base.

Chapter 8

Metacognitive Expectations

Over the years the notion of expectation has played a central role in the monitoring of plan execution (Pettersson, 2005; Schank, 1982) and in managing comprehension, especially natural language understanding (Schank and Owens, 1987). A common trait among these expectations is that they are defined as a function of the observed states and the actions executed so far. That is, they are defined in terms of the *ground level* environment; in terms of objects and events in the world (including the inferred objects of Chapter 6).

In this chapter we introduce a new class of expectation: expectations at the metacognitive level or *metacognitive expectations*. When cognitive processes compute expectations about the ground level, we consider the processes a kind of mental action that results in mental objects (i.e., the world-related expectations). These actions and objects then are at the *object level*. Analogously, processes at the *metalevel* can compute expectations about the object level behaviors. Metacognitive agents operate at three distinct levels (Cox and Raja, 2011):

- The ground level, in which an agent executes actions and makes observations of the state of the world.

- The object level (cognition), in which the agent reasons about the actions performed at the ground-level, the physical objects there and states of those objects.

- The metalevel (metacognition), in which the agent introspectively reasons about the cognition performed at the object level.

Under this perspective, agents such as GDA agents that reason about expectations at the object level (e.g., GDA agents) can be seen as reasoning at the cognitive levels. However, we do not know of any other work that reasons with explicit expectations at the metalevel[1].

We note that humans certainly have and exploit expectations concerning their own mental state (Dunlosky et al., 2013). Cognitive psychology research has long demonstrated the potential benefit and general characteristics of reasoning about mental state and mechanisms (Flavell, 1979; Flavell and Wellman, 1977). For example game show contestants can often decide before they retrieve a memory item whether or not they know answers to given questions and demonstrate this by striking buzzers before competitors. Phychologists examined this phenomena and found it to be a robust effect (Reder and Ritter, 1992). More recently research has shown that even very young students can assess how well they have learned material, can predict memory and problem-solving performance, and can use such metacognitive expectations to choose learning and study strategies (Dunlosky et al., 2013). However not all metacognition is necessarily helpful; it can actually lead to detrimental performance (Wilson and Schooler, 1991).

[1]The emphasis here is on the explicitness and formalism of these expectations. MetaAQUA (Cox, 1996; Michael and Ashwin, 1999; Lee and Cox, 2002) reasoned with implicit meta-level expectations.

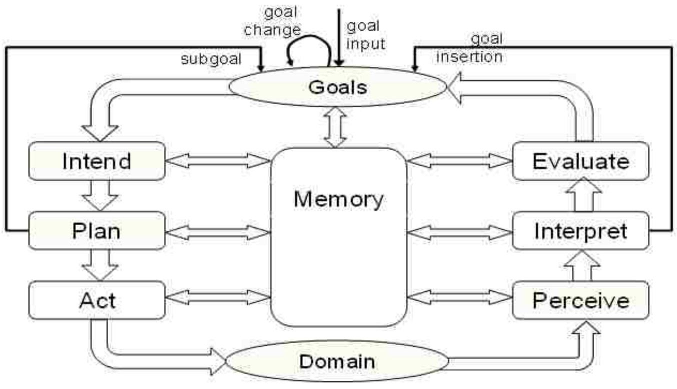


Figure 8.1: Schematic action-perception cycle for both object and meta-levels of the MIDCA Cognitive Architecture (used with permission from Michael T. Cox, see Cox et al. 2016 for more on MIDCA)

8.1 The MIDCA Metacognitive Architecture

The *Metacognitive Integrated Dual-Cycle Architecture (MIDCA)* Cox et al. (2016) is a three level architecture composed of the ground level (where actions and perceptions occur), the object level (where problem-solving and comprehension processes exist), and the metalevel (which introspectively monitors and controls the object-level cognition). The architecture integrates both planning and interpretation algorithms; our current implementation incorporates the SHOP2 hierarchical network planner (Nau et al., 2003) and a custom-made heuristic planner (Bonet and Geffner, 2001). In addition to simulation of standard planning domains, it also includes an ROS interface to physical robots (i.e., a Baxter humanoid robot) and the Gazebo physics simulator.

At the object level, the cycle achieves goals that change the environment. At the metalevel, the cycle achieves goals that change the object level. That is, the metacognitive "perception" components introspectively monitor the processes and mental state changes in cognition. Monitoring occurs by reasoning over traces of object-level behavior (e.g., decision making and inferences). The "action" component consists of a metalevel controller that mediates reasoning over an abstract representation of the object-level cognition. It can manage the activity at the object level by changing the object-level goals or by changing MIDCA's domain knowledge (e.g., SHOP action models).

8.2 Formalizing Metacognitive Expectations

We define a model of the metalevel, M_Σ, in an analogous manner of the action model used in planning: $\Sigma = (S, A, \gamma)$, where S is the set of states that the agent can visit, A is the collection of actions that the agent can take and γ is the state-action transition function $\gamma : S \times A \rightarrow S$. Let $M_\Sigma = (S_M, A_M, \gamma_M)$ where S_M is a collection of mental states, A_M is a collection of mental actions, and γ_M is a transition function defined as $\gamma_M : S_M \times A_M \rightarrow S_M$.

A mental state is a vector of variables, $\langle v_1, ..., v_n \rangle$. The variables that comprise a mental state are dependent on the agent; in MIDCA we use a separate variable for each of the following datums: *Selected Goals*, *Pending Goals*, *Current Plan*, *World State*, *Discrepancies*, *Explanations*, *Actions*. Thus our mental state is represented as a vector of length $n = 7$ as follows: $(g_c, \hat{G}, \pi, M_\Psi, D, E, \alpha_i)$ We make no constraints on the type of data for a mental state variable.

A mental action may perform reads or updates to variables in a mental state. MIDCA employs the following mental actions: *Perceive*, *Detect Discrepancies*, *Explanation*, *Goal Insertion* (i.e., formulates the set of goals $\hat{G}$ to be achieved), *Evaluate* (i.e., checks to see if the state M_Ψ entails the current goal g_c, and if so, removes it from $\hat{G}$), *Intend* (i.e., selects a new g_c from $\hat{G}$), *Plan* (i.e., calls a planner to generate a plan π to achieve g_c), and *Act* (i.e., executes the next step α_i from π). MIDCA enforces a strict sequence of mental actions that repeats (e.g., *Perceive* is always followed by *Detect Discrepancies*, *Act* is always followed by *Perceive*).

A mental trace, τ, is a sequence of mental states and mental actions interleaved. Specifically $\tau : \langle s_{M0}, \alpha_{M1}, s_{M1}, \alpha_{M2}, ...\rangle s_{Mn}$ where $s_{M0}, s_{M1} \in S_M$ and $\alpha_{M1}, \alpha_{M2} \in A_M$. When the sequence of mental actions are fixed (as is the case in MIDCA), a trace may be represented as only the mental states $\tau : \langle s_{M0}, s_{M1}, ...\rangle$. However, for the purpose of demonstrating the importance of the interaction of mental actions and mental states, we will use the representation of interleaved mental actions and mental states when describing metacognitive expectations.

We define a metacognitive expectation, $EX_M(s_{M_i}, \alpha_{M_i}, s_{Mi+1})$, over a segment of a mental trace τ. The length of the segment is always three: prior mental state, s_{M_i}, mental action, α_{M_i}, and subsequent mental state, s_{Mi+1}. Since a mental action can read the value of a mental state and update that variable given its original value, the mental state before and after a mental action are relevant to the expectation (thus the τ segment length of three). The metacogntive expectation is a function, $EX_M : (s_{M_i}, \alpha_{M_i}, s_{Mi+1}) \rightarrow \{true, false\}$. If the output is true, the expectation is met. Otherwise, the expectation is not met.

We now give an example of a metacognitive expectation that represents the following notion: *when the agent encounters a discrepancy, then the agent will generate an explanation for this discrepancy*. Formally this is expressed as follows: $EX_M(s_{M_i}, Explanation, s_{Mi+1})$ is a function that checks $v_{explanations} \neq \emptyset$ in s_{Mi+1} whenever $v_{discrepancies} \neq \emptyset$ in s_{M_i}. For example, at the object level when MIDCA detects a discrepancy between the expected outcome of an action and the observed state, then an explanation for this object-level discrepancy must be generated. If the agent executes the action to move east and MIDCA observes that the agent remains in the same place, MIDCA might generate as an explanation that the agent is stuck and generate a new goal to become $\neg$stuck. In our experiments at some point during execution, strong winds will permanently affect the agent's move-east action, causing it to move not only one cell east but also an additional three cells to the east. When this happens, MIDCA cannot generate any plausible explanation for this discrepancy at the object level. This triggers a discrepancy with the metacognitive expectation because no explanation at the ground-level was generated. In the next section we describe the MIDCA procedure and indicate how MIDCA deals with this discrepancy of metacognitive expectations.

8.3 Reasoning with Metacognitive Expectations

In the MIDCA architecture, mental traces are constructed when cognitive processes are executed at the object level. Algorithm 3 illustrates the procedure for capturing mental states and process execution. The gamma function is an implementation of γ_M described above. We assume here that the application of any cognitive process to an input state will output a result s_M in the appropriate format (i.e., a 7-tuple mental state). The input, output and time of invocation are recorded in self-referential object form for the process (Lines 3-6). The function $now(t)$ in Line 5 is a temporal predicate as defined in active logic (Anderson et al., 2008), a kind of step logic. The process and then the output state are subsequently appended to the mental trace τ. Note that this assembles the mental trace in opposite order compared to the description in the previous section. However Algorithm 4 below will reverse the sequence. Finally Line 9 returns the output to complete the procedure.

After initialization of the basic metacognitive state (Line 8), cognition performs one phase (determined by Lines 9-12) of its action-perception cycle (i.e., executes the switch-case control structure in Lines 14-32) and then invokes metacognition (Line 33). Each clause in the switch returns a mental state as defined in the previous section but modifies only selective elements in the state. A t in any element's position indicates that the value remains unchanged. For example, $perceive$ assigns a value to the fourth element of the state (a percept $\vec{p}$ representation of the environment denoted by Ψ) to be unioned into the model of the environment M_Ψ (see Line 16). The remaining six elements are not effected.

Algorithm 3 The $gamma_M$ procedure that creates a mental trace τ. The expression $head|\langle tail\rangle$ prepends the element $head$ to the sequence $\langle tail\rangle$.

```
1: **global** $\tau$
2: **procedure** $\gamma_M(process, args)$
3:     $process.in \leftarrow args$
4:     $process.out \leftarrow apply(process, args)$
5:     **if** $now(t)$ **then**
6:         $process.time \leftarrow t$
7:     $\tau \leftarrow process.self|\tau$
8:     $\tau \leftarrow process.out|\tau$
9:     $return(process.out)$
```

Note the parallel control structure between the cognitive and metacognitive cycles. Each includes a comprehension sequence of $\langle perceive/monitor, interpret, evaluate\rangle$ and a problem-solving sequence of $\langle intend, plan, act\rangle$. In contrast, the former uses γ_M to record a trace whereas the latter does not. Yet functionally they are very similar. The cognitive level interprets the percepts from the external environment; whereas, the metalevel interprets the current trace of the internal object level. Interpretation is actually composed of a GDA three-step process. It detects discrepancies, explains them, and uses the explanation to generate potential new goals to resolve the discrepancy. This is implemented by having $interpret$ add $explain$ to the head of the processes to be executed (Line 18) and then calling $detect$ (Line 19). Explanation then prepends

Algorithm 4 The MIDCA procedure. Note that each invocation of γ_M returns a cognitive 7-tuple. For brevity we indicate with the "t" symbol an element that does not change as a result of the output.

```
 1: global $\tau$, $cycle$
 2: $cycle \leftarrow \langle perceive, interpret, eval, intend, plan, act \rangle$
 3: procedure MIDCA($\Psi$)
 4:     $g_c \leftarrow \emptyset$; $\hat{G} \leftarrow \emptyset$
 5:     $\tau \leftarrow \langle \emptyset \rangle$                      ▷ Initialize trace with $s_{M0}$
 6:     cognition($\emptyset, \phi, cycle$)

 7: procedure COGNITION($M_\Psi, \pi, procs$)
 8:     $\pi_M \leftarrow \phi$; $g_M \leftarrow \emptyset$; $\hat{G}_M \leftarrow \emptyset$
 9:     if $procs = \langle \rangle$ then
10:         $procs \leftarrow cycle$
11:     $next \leftarrow head(procs)$
12:     $procs \leftarrow tail(procs)$
13:
14:     switch $next$ do
15:       case $perceive$
16:         $(t, t, t, M_\Psi \cup \vec{p}, t, t, t) \leftarrow \gamma_M$ (perceive, ($\Psi$))
17:       case $interpret$
18:         $procs \leftarrow explain|procs$
19:         $(t, t, t, t, D, t, t) \leftarrow \gamma_M$ (detect, $(\vec{p}, \pi)$)
20:       case $explain$
21:         $procs \leftarrow insertion|procs$
22:         $(t, t, t, t, t, E, t) \leftarrow \gamma_M$ (explain, $(D, \pi)$)
```

```
23:         case $insertion$
24:             $(t, \hat{G}, t, t, t, t, t) \leftarrow \gamma_M$ (goal-insertion, $(E, \pi)$)
25:         case $eval$
26:             $(t, \hat{G}, t, M_\Psi, t, t, t) \leftarrow \gamma_M$ (evaluate, $(D, E, g_c)$)
27:         case $intend$
28:             $(g_c, t, t, t, t, t, t) \leftarrow \gamma_M$ (intend, $(\hat{G}, \pi)$)
29:         case $plan$
30:             $(t, t, \pi', t, t, t, t) \leftarrow \gamma_M$ (plan, $(M_\Psi, g_c, \pi)$)
31:         case $act$
32:             $(t, t, t, t, t, t, \alpha) \leftarrow \gamma_M$ (act, $(\pi')$)

33:     metacognition($M_\Psi, \emptyset, \pi', \tau, procs$)

34: procedure METACOGNITION($M_\Psi, M_\Omega, \pi, \tau, procs$)
35:     $\tau' \leftarrow$ monitor($M_\Psi, \pi, reverse(\tau)$)
36:     $D_M, E_M, \hat{G}_M \leftarrow$ meta-interpret($\tau', \pi_M$)
37:     $M_\Omega, \hat{G}_M \leftarrow$ meta-evaluate($D_M, E_M, g_M$)
38:     if $\hat{G}_M = \emptyset$ then
39:         cognition($M_\Psi, \pi, procs$)
40:     else
41:         $g_M \leftarrow$ meta-intend($\hat{G}_M, \pi$)
42:         $\pi'_M \leftarrow$ meta-plan($M_\Omega, g_M, \pi$)
43:         controller($\pi'_M$)
44:         metacognition($M_\Psi, M_\Omega, \pi'_M, \tau', procs$)
```

$goal\text{-}insertion$ to the procedure sequence (Line 21) and calls $explain$ (Line 22).

Algorithm 4 Dannenhauer et al. (2017) describes the basic high-level flow of control for the MIDCA architecture. MIDCA initializes the internal state (Lines 4-6) and starts a cognitive cycle (Line 6). The mental trace τ is initialized to the sequence having the empty set as its only member. The empty set represents the initial mental state s_{M0}. It is empty because no perception has occurred and thus no knowledge of the world yet exists.

If an expectation failure occurs (on Line 36, $\hat{G}_M$ will not be empty), it attempts to repair itself; otherwise it calls cognition to continue. Note that the mental trace τ is reversed on Line 35 before passing to the $monitor$ function given that it is constructed in the opposite order by Algorithm 3. Interpretation at the metalevel is performed as a three-step process as is the case with the object-level analogue. It is however performed in a single call on line 36, setting a discrepancy D_M if one exists from any metacognitive expectation, explaining D_M, and then formulating a new goal if necessary to change the object level. Line 37 performs evaluation on any goal from previous metalevel cycles, and then Line 38 checks to see if a discrepancy existed (it will if a new meta-level goal was formulated. If so, it performs goal selection with intend (Line 41), plans for the goal at Line 42 and executes the plan with the metalevel controller on Line 43. Metacognition is called again to perform evaluation on the results (Line 44).

Reconsidering the example at the end of the previous section, when MIDCA fails to generate an explanation at the object level because permanent changes in the wind conditions causes the agent to always be pushed an additional 3 cells whenever it moves east. Since no explanation is generated at the object level, it causes a discrepancy of a metacognitive expectation. In this case Line 36 will evaluate τ', which will contain a sequence of the form $(s_{M_i}, Explanation, s_{Mi+1})$ with $v_{explanations} = \emptyset$ in s_{Mi+1} and

$v_{discrepancies} \neq \emptyset$ in s_{M_i}. This will detect the discrepancy and generate a new goal $\hat{G}_M$. Therefore the condition in Line 38 fails and the else code in Lines 41-44 is performed. Line 41 selects the new goal, g_M, to fix the domain theory (the new goal is generated during meta-interpret on Line 36 and is a member of the resulting $\hat{G}_M$). Line 42 generates a plan to modify the domain description (now the effects of the move east will correctly indicate that it will be pushed 3 additional cells to the right), the controller makes this change in the domain description (Line 43) and the process is re-started again.

8.4 Discussion

This work extends the existing research on anticipatory approaches to intelligent agents within a cognitive architecture that use expectations to focus problem solving on salient aspects of a problem. We introduce metacognitive expectations, a novel class of expectations that applies the approach to the problem-solving cycle itself rather than just the environment and actions executed within it. Using a persistent domain, NBeacons (Section 9.1.6), that changes over time, we provide empirical results (Section 9.2.5) showing the benefit of metacognition made available by metacognitive expectations.

Chapter 9

Evaluation

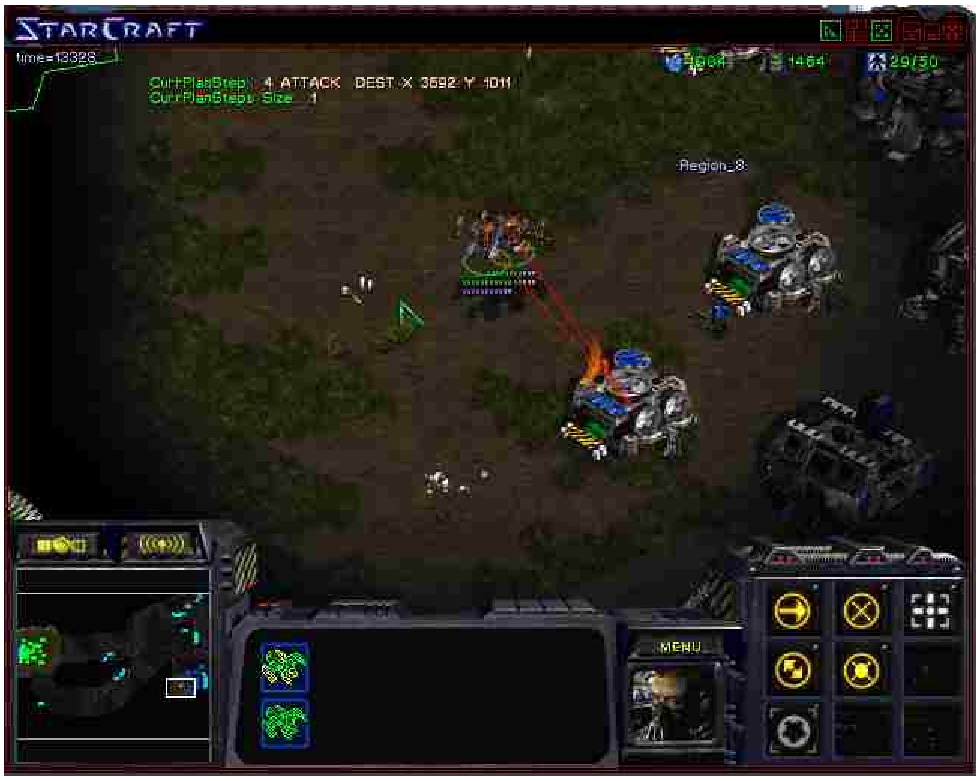
9.1 Simulated Domains

The following domains were used to test the approaches described in Part II. The qualities of these domains are shown in Table 9.1

Domain Name	**Dynamic**	**Partially Observable**	**Deterministic Actions**
Starcraft	Yes	Yes	No
Marsworld[1]	Yes	No	Yes
Arsonist	Yes	No	Yes
Marsworld[2]	Yes	Yes	Yes
BlocksCraft	Yes	Yes	Yes
NBeacons	Yes	No	Actions' effects may change over time

Table 9.1: Properties of Simulated Domains

All of these domains have properties that make planning and execution difficult, and in many cases present an environment ideal for observing the benefit of self-monitoring strategies.



```
time=13328
CurrPlanStep: 4 ATTACK  DEST X 3692 Y 1011
CurrPlanSteps Size  1
```

Figure 9.1: Screenshot of Starcraft Gameplay

9.1.1 Starcraft

Real-Time Strategy (RTS) is a large genre of video games with many commercially successful titles including Starcraft, which has sold millions of copies and enjoyed a professional gaming scene, particularly in South Korea. In general, the objective of real time strategy games is to build up an army and attack with that army to defeat an opponent attempting to do the same. The real-time quality of these games is that speed is important: the faster the player can issue commands, the faster it can build, move, and attack with his/her army.[1]. The strategy component of these games lie in both economic and combat decision making. Players start with just a few units and nearby resources.

[1]Some actions have a minimum wait time: it might take 3 seconds for a soldier to be produced once the player gives the command

Only after using initial workers to harvest these resources, can the players produce more units and units of various types. Once the player has some units with which they wish to battle, they must move them into proximity of the enemy and engage. A player has won when either his/her opponent has resigned or all of the opponents troops and buildings are destroyed.

Starcraft has three factions: Terran, Protoss, and Zerg. Players choose which faction they wish to play as, and each has its own unique play style and units. For example, Zerg units have an ability to burrow into the ground to avoid detection and almost all Terran buildings can be airlifted and fly (at a slow pace) around the map (neither Zerg nor Protoss can move their buildings).

At the beginning of a match, there is a preset number of resources on the map. Usually each player starts far away from another player, and near some resources. The resources in the game are minerals and vespene gas. Minerals must be harvested by worker units, and players spend these resources to build buildings, produce fighting or defensive units, and to pay for research upgrades. Generally the more powerful a unit, the more resources it costs. Thus, an important part of the game is making sure one has a steady stream of resources being harvested. There are also multiple strategies that involve preventing a player from acquiring these resources (i.e., killing or harassing enemy worker units) to slow down the economy of that player.

Each faction has many unit and building types for the player to choose. To give some perspective, Terran has ten different types of ground units, five different kinds of air units, twelve different kinds of buildings, with an additional six building add-ons the player can build once a building is complete. For example, when a player builds a Terran Factory, they can only build vultures (a type of fast ground unit). However, if they build a machine shop add-on, the factory will then be able to produce seige tanks (slow moving but powerful damage unit with long-range). There are also units that can only be produced by if the player has multiple buildings. Continuing with the Terran factory example, if the player also has an armory (a building used to do upgrades on units) then a factory can produce goliaths (a versatile ground unit that can also attack air units). To add even more complexity, some buildings allow the player to spend resources to perform research upgrades: these generally improve some quality of units, such as giving machine units like vultures and tanks increase damage against the opponent.

The order in which you construct buildings and produce fighting units is crucial in the beginning of a match. There is a trade-off between constructing the buildings and harvesting the resources that are needed to build more powerful units in the future versus building weaker units more quickly. A rush attack is when one player builds many cheap units to attack the enemy, before he has produced any defensive capabilities. It is important to note that if a rush fails, the player who rushed may then be at a disadvantage, having less resources then the defending player who was planning for the long term.

Another important aspect of Starcraft is “fog of war”: each players visibility of the map is determined from the radius of its units. From the players perspective, a region is unknown if it is enveloped in the fog of war. This makes Starcraft a partially observable domain. Unless your units are near enemy units, you can not see them. Scouting (sending a troops to see what the enemy is doing, is an important strategic aspect).

Starcraft has an active research community and the game has enjoyed world-wide popularity (including televised professional matches) and remains a challenging domain for automated computer agents. Like games such as chess and go, there are professional human experts for which to test A.I. against. Evidence of the difficulty of the domain for A.I. comes not only from characteristics of the game (massive state space, stochastic actions, partial visibility, etc.) but also from three years of competitive entries in tournaments (i.e., AIIDE Annual Starcraft Competition) in which the best automated entry has performed poorly against a human expert. Figure 9.1 shows a screenshot of the game where the player has issued an attack command to the highlighted units in the middle of the screen: they are attacking an enemy bunker that has begun to catch fire.

9.1.2 Marsworld[1]

Marsworld[1] is a domain involving a rover agent in a tile-grid environment which is tasked with activating beacons. The underlying idea behind beacon activations is to create a signal. Marsworld[1] 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*[1] mud causes the agent to be stuck. This difference allows us to examine obstacles that require choosing new goals. The other difference between *Marsworld*[1] 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.

9.1.3 Arsonist

The second domain is a slight variant of the *Arsonist* domain from 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.

Execution cost of a plan in *Marsworld*^{1} 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*^{1} 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.

9.1.4 Marsworld^2

Marsworld^2 is a modified version of Marsworld^1 Dannenhauer and Muñoz-Avila (2015b). Modifications were needed to make the domain partially observable; in that work the authors only deal with dynamic but not partially observable domains. It consists of a square grid of a 100 tiles, with randomly generated objects: beacons and piles of wood. The agent also begins with flares in its inventory. The agent has one overarching goal: signal a nearby agent for assistance. The agent creates a signal in one of three ways: activate a specified number of beacons, light a specified number of fires using wood, or drop and light a specified number of flares. When the agent is in the same tile as a beacon or a pile of wood, it can activate the beacon or create a fire. If the agent is in an empty location, it can drop and light a flare. Each of these may fail: beacons deactivate and fires and flares can be permanently extinguished. Only after the target number of activated beacons, fires, or flares have been reached can the agent signal for assistance. The agent is endowed with 7 plan actions: moveup, movedown, moveright, moveleft, activatebeacon, makefire, dropflare. The agent can sense anything in its current tile and any adjacent tiles (N,S,E,W) with a cost of 0. When an object is no longer within view, the agent can check on the object with a sensing action at a cost of 1. Hence, an agent can perform enough sensing to know everything it has seen, but at a high cost (i.e., the cost for each sensing action required to view everything in its belief state).

9.1.5 BlocksCraft

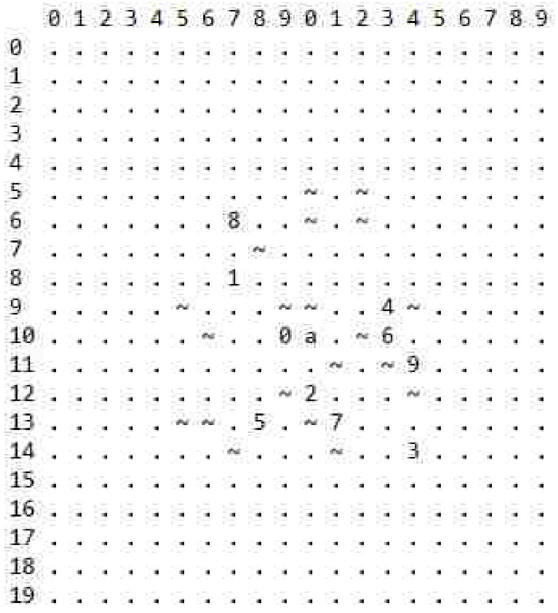
The **blockscraft** domain is inspired by the popular sandbox game Minecraft. In this domain the agent's goal is to build a 10-block tower by picking up and stacking blocks. Blocks can only be stacked on the ground or on top of blocks of the same type. The agent does not know what blocks will be available to it over the course of its execution. In the game Minecraft, often the player will dig for blocks and uncover different types of blocks, only known after acquiring them. The agent always has 3 blocks in a nearby quarry to choose from, and only after it uses a block will a new one become available. In our experiments there are three different types of blocks and each new block has a randomly selected type. Blockscraft is dynamic due to external agents, unseen to our agent, that may remove blocks from a tower as well as building their own towers. This is akin to the online multiplayer aspect of Minecraft, where many players can modify a shared world. Blockscraft is partially observable in that our agent only has a top-down view of the blocks. The top-down view enables sensing actions of cost 0 for the top two blocks of each tower it has built. Any other block (those under the top two) can be sensed with a cost of 1.

9.1.6 NBeacons

We performed an ablation study of three agents in the NBeacons domain. NBeacons is a modification of the Marsworld[1] domain used in Dannenhauer and Muñoz-Avila (2015b). The motivation behind NBeacons is to test robust agents in environments that change over time.

The NBeacons domain is unbounded, with an area of interest to the agent that is a square grid of 100 tiles. Scattered among the area of interest are a total of 10 beacons and 20 sand pits. Beacons and sand pits cannot be in the same tile. Beacons are deactivated until the agent, which must be on the same tile as the beacon, performs an activate-beacon action. Other actions available to the agent include navigation (move east, west, north, south) and push actions (to free an agent that has fallen into a sand pit). There are five push actions an agent must execute before it will become free to move. Figure 9.2 shows a part of an NBeacons domain and is centered around the area of interest. Wind may push the agent outside the area of interest, in which case the agent will navigate back.

The primary difference of NBeacons compared to Marsworld[1], is the addition of wind. After a period of time, wind blows causing the agent to be instantaneously pushed extra tiles whenever it attempts to move in a specific direction. If the agent would be pushed through any tiles containing sand pits, the agent will become stuck in the sand pit. In our experiments, wind begins to take effect after 500 ticks at a strength of 3 (pushing the agent 3 extra tiles) and then increases to a strength of 4 after 1500 ticks. Wind causes the effects of the agent's move action to be permanently altered for the rest of its execution. Due to wind, an agent will find itself in a location other than what it expected, sometimes ending up in a sand pit. Wind always blows in one direction.



```
   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
0  . . . . . . . . . . . . . . . . . . . .
1  . . . . . . . . . . . . . . . . . . . .
2  . . . . . . . . . . . . . . . . . . . .
3  . . . . . . . . . . . . . . . . . . . .
4  . . . . . . . . . . . . . . . . . . . .
5  . . . . . . . . . . ~ . ~ . . . . . . .
6  . . . . . . . 8 . . ~ . ~ . . . . . . .
7  . . . . . . . . ~ . . . . . . . . . . .
8  . . . . . . . 1 . . . . . . . . . . . .
9  . . . . . ~ . . . ~ ~ . . 4 ~ . . . . .
10 . . . . . . ~ . . 0 a . ~ 6 . . . . . .
11 . . . . . . . . . . . ~ . ~ 9 . . . . .
12 . . . . . . . . . ~ 2 . . . ~ . . . . .
13 . . . . . ~ ~ . 5 . ~ 7 . . . . . . . .
14 . . . . . . . ~ . . . ~ . . 3 . . . . .
15 . . . . . . . . . . . . . . . . . . . .
16 . . . . . . . . . . . . . . . . . . . .
17 . . . . . . . . . . . . . . . . . . . .
18 . . . . . . . . . . . . . . . . . . . .
19 . . . . . . . . . . . . . . . . . . . .
```

Figure 9.2: NBeacons Example Scenario: **a** represents the agent, integers between 0-9 are the locations of beacons, and $\sim$ represents a sand pit

Without wind, an agent never ends up in a sand pit because it will plan around them. All agents perform heuristic planning with the heuristic straight-line distance. Agents are given goals to activate specific beacons. Beacons are deactivated after they are activated to ensure an agent always has an available goal.

All three agents are implemented using the MIDCA Cognitive Architecture, albeit with ablated components. The baseline agent, a re-planning only agent, checks to see if the goal was reached at the end of execution of its plan, and if not, re-plans. The second agent, a goal driven autonomy agent, uses expectations to determine when a discrepancy of the world occurs, followed by explanation, and finally goal formulation. Expectations use a state comparison approach described in Dannenhauer and Muñoz-Avila (2015b) and used in Molineaux et al. (2010). Explanation is implemented as a mapping from discrepancies to goals. Thus our GDA agent generates a goal to become free after it explains that the agent is stuck in a sand pit. When the GDA agent finds it has been blown off course (but not in a sand pit), it inserts the same beacon-activation goal it previously had, which triggers immediate planning in the agent's new state.

The third agent contains the GDA mechanism of the previous agent (GDA at the cognitive level) along with metacognition enabling it to reason with meta expectations. These expectations include the expectation mentioned in the example earlier: an agent detecting a discrepancy should also have generated an explanation. When the cognitive GDA component, Explanation, fails to produce an explanation after observing the discrepancy of being in an unexpected location, the metacognitive expectation is triggered and the agent generates a goal to update its domain model. Since the focus of this work is on modeling expectations and using them to detect any anomalies across any area of cognition, we do not go into detail regarding operator learning. Research in learning action models for planning domains has a long history (Čertickỳ, 2014; Wang, 1995, 1996; Yang et al., 2007). In this work, we use an oracle-like function that updates the appropriate action given the distance the agent was pushed by the wind. Unlike the other two ablated agents, the metacognitive agent is able to update its model for the action affected by wind (i.e., when wind is blowing east, the move-east action is updated).

Our hypothesis is that the agent equipped with metacognitive expectations will achieve the lowest execution cost, followed by the GDA agent, and finally the replanning agent. We expect that the GDA agent will immediately respond to discrepancies while the replanning agent waits for its current invalid plan to run out. The metacognitive agent will detect an explanation failure of its gda components and update its effects of the move action to take into account the new presence of the wind. Thus the gda agent will be blown unexpectedly into sand pits while the metacognitive agent will avoid them once it has learned from previous encounters with the wind.

	Goals	Obstacles	Observability
Marsworld^1	Activate 3 Beacons	Beacon failures, mud	Full
Marsworld^2	Activate n beacons Have n fires lit, Have n flares lit	Beacon failures, fires and flares may be extinguished by wind	Partial - with ability to query the state for already-seen atoms
NBeacons	Activate n beacons	No beacon failure; wind pushes agent extra tiles in the east direction	Full

Table 9.2: Differences in Marsworld-based domains

To help summarize the differences between Marsworld^1, Marsworld^2, and NBeacons, the following Table 9.2 summarizes their differences:

9.2 Results

9.2.1 Expectations in Dynamic & Fully Observable Domains

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.

Figures 9.3 and 9.4 show the average results of 5 runs from the GDA agents on

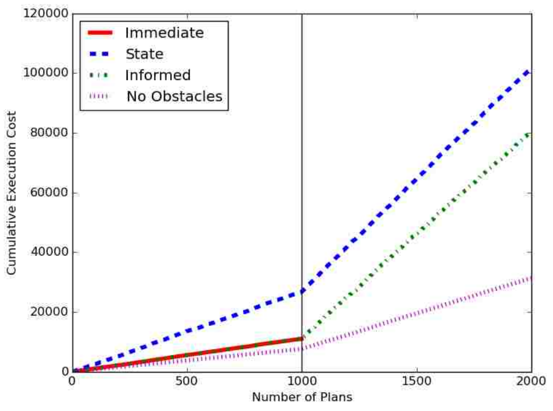


Figure 9.3: Cumulative Execution Cost in Marsworld[1]

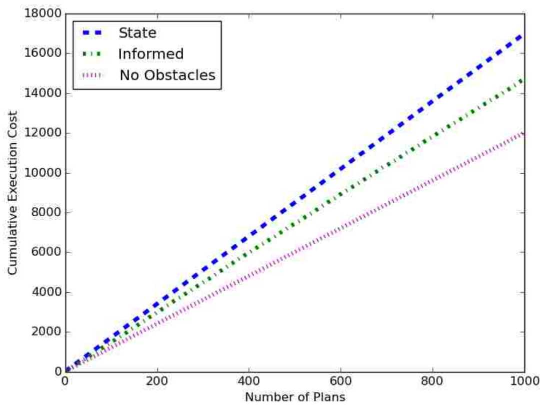


Figure 9.4: Cumulative Execution Cost in Arsonist

Marsworld[1] 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[2].

In the first domain (Figure 9.3), *Marsworld*[1], 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. However, the state agent ended up taking much longer because it triggered false anomalies. This is due to mud obstacles 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 9.5 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

[2]Execution cost when their are no obstacles is not possible to achieve because obstacles do in fact exist, but is helpful to include to give a sense of what plan execution would look like alone

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 9.6). As the probability of fire increases (x-axis) the immediate expectations agent fails to complete more plans (y-axis).

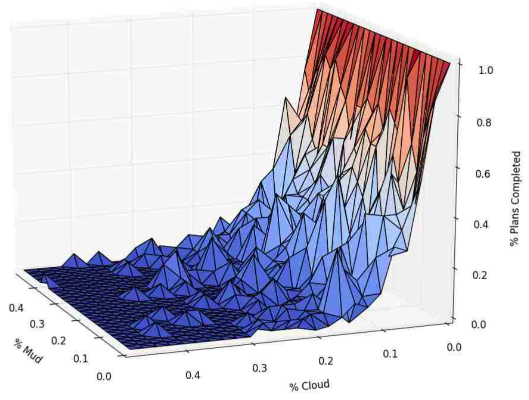


Figure 9.5: Plan Success vs. Obstacles (Immediate Expectations agent) in Marsworld[1]

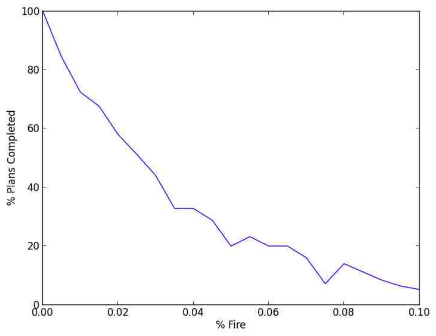


Figure 9.6: Plan Success vs. Obstacles (Immediate Expectations agent) in Arsonist

9.2.2 Expectations in Dynamic & Partially Observable Domains

In Figures 9.7 and 9.8, the first bar of each agent (green) is the percentage of goals achieved. The second bar (red) is the percentage sensing cost out of the maximum sensing cost. The maximum sensing cost is the sensing cost of all atoms in the state (as shown by our upper bound: complete expectations). The third bar (purple) shows the normalized total of actions executed by each agent. In Figure 9.7 the chance of failure per action executed was 20% for beacons, fires, and flares each. In Figure 9.8, the chance that a block would be removed was 10% and the chance that a block would be added was 30%. Chances for discrepancy to occur are computed per every planning action executed by the agent.

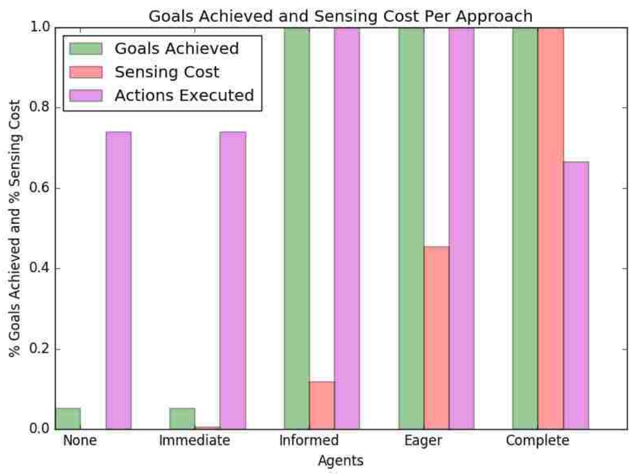


Figure 9.7: Sensing Cost per Failure Rate in Marsworld[2]

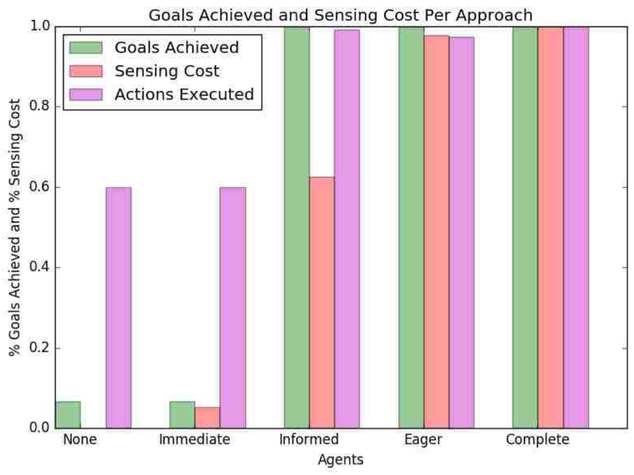


Figure 9.8: Sensing Cost per Failure Rate in Blockscraft

Looking at Figure 9.7, we see that agents using *none* and *immediate* expectations were unable to achieve most of their goals. Agents using *informed* and *eager* were able to achieve all of their goals. However, *informed* expectations incurred in significantly less sensing costs than *eager*, and less than the upper bound shown by *complete*. The *none* and *immediate* expectations agents do not become aware of failures outside their limited view and thus fail to switch goals, reaching their (falsely believed) goal with less actions (compared to *informed* and *eager*). In these experiments we turned off the sensing checking of the goals (Line 42 of the Algorithm) as the *none* and *immediate* expectation agents were taking too long to complete their goals while the *informed*, *eager*, and *complete* agents are guaranteed to satisfy the goals they believe they achieved. We did not implement goal regression in our algorithm because in the domains we tested our agent cannot generate a complete grounded plan from the outset due to partial observability.

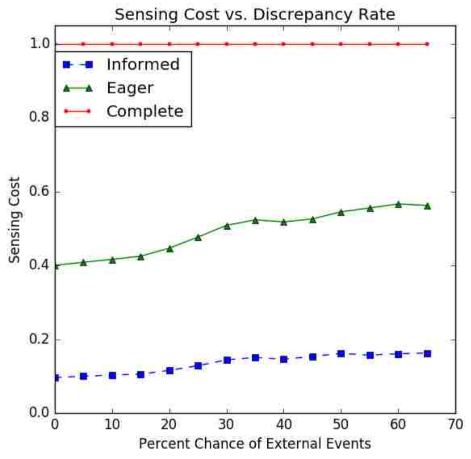


Figure 9.9: Sensing Cost per Failure Rate in Marsworld[2]

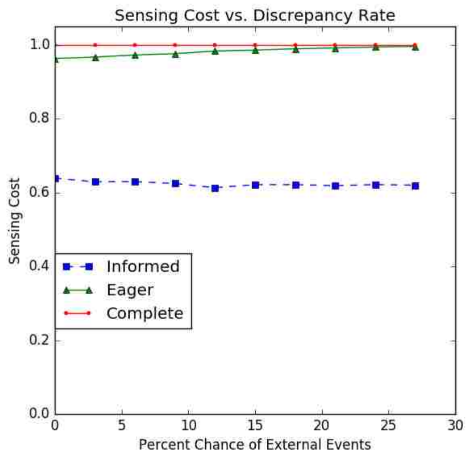


Figure 9.10: Sensing Cost per Failure Rate in Blockscraft

Figure 9.8 shows results from the blockscraft domain. We see similar results to those in marsworld. The none and immediate fail to achieve goals most of the time because even a change in a single block will go unnoticed. We see a cost gap in the *informed* and *eager* expectations as a result of the *eager* expectations agent sensing everything it has ever seen (including in this case the other towers under construction by other agents), while *informed* only keeps track of those atoms in the state that are related to its previous actions. Thus it performs sensing only on the blocks the agent itself has stacked.

Figures 9.9 and 9.10 show the total sensing (relative to complete expectations) varied by the chance of failure in each domain. Each data point is the total sensing cost performed out of the maximum possible sensing cost over 100 runs. In Figure 9.9 the chance for each of beacons, fire, and flares to fail varied from 0% to 65% in increments of 5%. Figure 9.10 shows the results of blockscraft where the chance that blocks were removed had a failure rate that varied from 0% to 27% in increments of 3%. In Figure 9.10, only the chance that blocks were removed was varied, the chance for blocks being added was held at 30%. We did not test with values close to 100% failure rate in both domains because at some point the environment changes so frequently it is not possible to achieve any goals, even with perfect sensing. By stopping at 65% and 27% respectively, we are able to see what is happening while still enabling agent's to achieve all of their goals. In both of these figures we see that informed expectations is performing substantially less overall sensing than eager and complete expectations. In blockscraft the difference between eager and complete is negligible because both agents basically see the same blocks regardless if used by the own agent or by the external agents.

9.2.3 Scenario Demonstrations of High Level Expectations in Starcraft

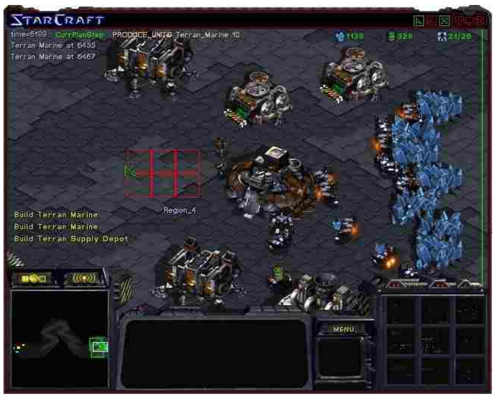
We now describe three scenarios where *LUiGi* (see chapter 6) was able to successfully detect a discrepancy and choose a new goal using the ontology.

Scenario 1: Early rush attack

In this scenario LUIGi gets rushed by the enemy early in the game. A discrepancy is detected soon after enemy units enter LUIGi's starting region. The discrepancy was that LUIGi does not control its base region because the region is contested and an explanation trace similar to Figure 6.1 is generated. LUIGi sends the explanation to the goal formulator component which chooses a new goal to defend the region. As part of the new goal, LUIGi recalls all units (including those in other regions) and uses them to attack the enemy forces in its base region. Figure 9.11a shows LUIGi in the process of producing troops while controlling the region. Figure 9.11b shows LUIGi pursuing a newly generated goal to defend the region after detecting and explaining the discrepancy of not controlling the region (i.e., the blue units are enemy units).

Scenario 2: Units do not reach destination

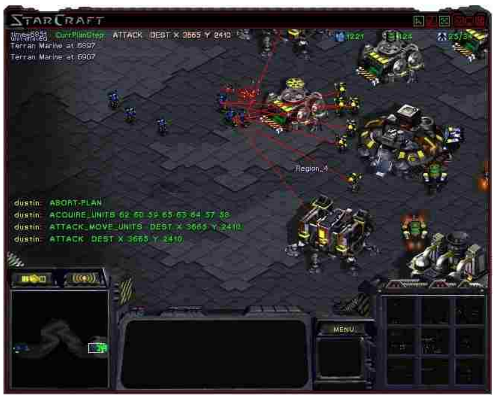
In a second scenario, LUIGi successfully builds ground units and sends them to attack the enemy base. However, the enemy has set up a defense force at its base region's perimeter, which destroys LUIGi's ground units before they make it to the enemy region. The expectation that is violated is a primitive expectation (e.g., $\langle$unit5, isInRegion, region8$\rangle$) and the discrepancy is that LUIGi expects unit5 to be in the region region8. The explanation is simply the negation of the expectation. LUIGi chooses a new goal



```
time=5129  CurrPlanStep  PRODUCE_UNITS Terran_Marine 10
Terran Marine at 6455
Terran Marine at 6467

Build Terran Marine
Build Terran Marine
Build Terran Supply Depot
```

(a) No Discrepancy



```
time=5851  CurrPlanStep  ATTACK  DEST X 3665 Y 2410
Terran Marine at 6897
Terran Marine at 6907

dustin:  ABORT-PLAN
dustin:  ACQUIRE_UNITS 62 60 59 65 63 64 57 58
dustin:  ATTACK_MOVE_UNITS  DEST X 3665 Y 2410
dustin:  ATTACK  DEST X 3665 Y 2410
```

(b) Discrepancy due to enemy Rush Attack

Figure 9.11: Screenshots of LUIGi building an initial army

to build units that fly in order to bypass the units defending the enemy's base. However, there are multiple valid goals worth pursuing in this situation, such as building different units capable of defeating the enemy units defending the enemy base's perimeter, or taking a different route leading into the enemy base.

Scenario 3: Units reach destination but do not defeat enemy

In a third scenario, LUIGi's units were produced and moved to the enemy region successfully, but were not able to defeat the enemy in its base region. The expectation that LUIGi controlled the enemy base region was unmet after LUIGi's units finished executing the plan step to attack the enemy units in the base and LUIGi's units were killed. The corresponding explanation, informally, was that LUIGi had no units in the region when the plan step of attacking had finished. While the expectation is the same as scenario 1, the explanation is different (i.e., the traces are different). As a result LUIGi chooses a different goal than in Scenario 1. In this case, LUIGi chooses a goal that does not involve directly attacking the enemy but instead to secure and defend other locations that contain valuable resources. This type of strategy gives LUIGi an economic advantage over its opponent yielding a more substantial army later in the match.

9.2.4 LUiGi-H vs. LUiGi in Starcraft

In order to demonstrate the benefit of h-plans, we ran *LUiGi-H* against the baseline *LUiGi* . Matches occurred on three different maps: Heartbreak Ridge, Challenger, and Volcanis. Heartbreak Ridge is one of the most commonly used maps for Starcraft (it is one of the maps used in AIIDE's annual bot tournament), while Challenger and Volcanis are common well-known maps. Data was collected every second, and the Starcraft match was run at approximately 20 frames per second (BWAPI function call of setLocalSpeed(20)). The performance metrics are:

- **kill score.** Starcraft assigns a weight to each type of unit, representing the resources needed to create it. For example, a marine is assigned 100 points whereas a siege tank is assigned 700 points. These points are pre-assigned values the game designers used in computing a player's score after the match finishes. The kill score is the difference between the weighted summation of units that *LUiGi-H* killed minus the weighted summation of units that *LUiGi* killed.

- **razing score.** Starcraft assigns a weight to each type of structure, representing the resources needed to create it. For example, a refinery[3] is assigned 150 points whereas a factory[4] is assigned 600 points. The razing score is the difference between the weighted summation of structures that *LUiGi-H* destroyed minus the weighted summation of structures that *LUiGi* destroyed.

[3]A refinery is a building that allows to harvest gas, a resource needed to produce certain kinds of units. For instance, 100 gas units are needed to produce a single siege tank.

[4]A factory is building that allows the production of certain kinds of units such as siege tanks provided that the required resources have been harvested.

- **total score.** The total score is the summation of the kill score plus the razing score for *LUiGi-H* minus the summation of the kill score plus the razing score for *LUiGi* .

In addition to these performance metrics, the unit score is computed. The unit score is the difference between the total number of units that *LUiGi-H* created minus the total number of units that *LUiGi* created. This is used to assess if one opponent had an advantage because it created more units. This provides a check to ensure that a match wasn't won because one agent produced more units than another.

We show our results in Figure 2 below.[5]

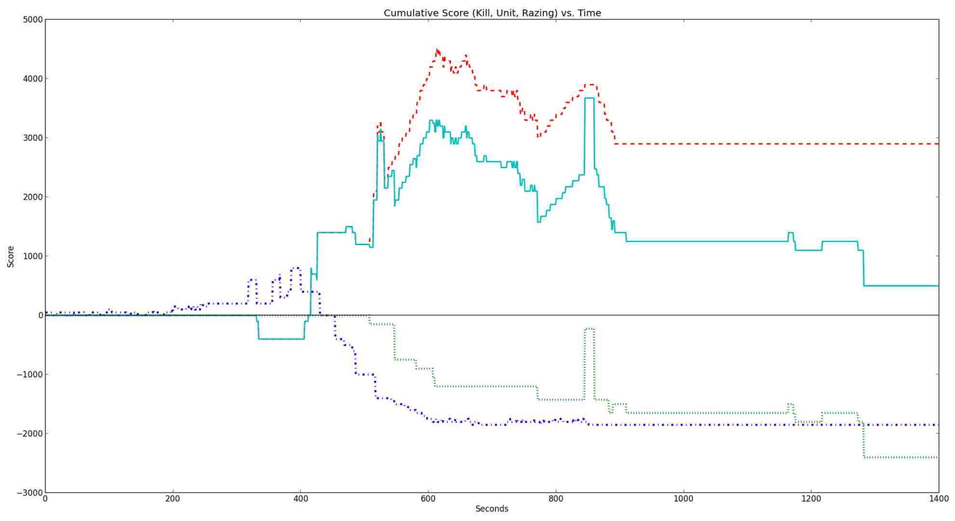


Figure 9.12: *LUiGi-H* vs. *LUiGi* on Heartbreak Ridge

The red dashed line shows the kill score, the blue dot-dashed line shows the unit score and the green dotted line is the razing score. The total score, which is the sum of the kill score and razing score is shown as the unbroken cyan line. All lines show

[5]We plot results for a single run because difference in scores between different runs were small.

the difference in cumulative score of *LUiGi-H* vs. *LUiGi* . A positive value indicates *LUiGi-H* has a higher score than *LUiGi* .

From Figure 9.12 we see that *LUiGi-H* ended with a higher total score than *LUiGi* , starting around the 400 second mark. In Figure 9.12, the difference in the blue dot-dashed line (unit score) shows that in this match the *LUiGi* system produced far many more units than the *LUiGi-H* system. Despite producing significantly fewer units *LUiGi-H* system outperformed *LUiGi* as can be seen by the total score line (cyan unbroken). *LUiGi-H* scored much higher on the kill score, but less on the razing score. A qualitative analysis revealed that *LUiGi* had slightly more units end game, shown in the graph by the much higher unit score (blue dot dashed), which caused its razing score to be higher than *LUiGi-H* . We expect that as the unit score approaches zero, *LUiGi-H* will exhibit higher kill and especially razing scores. *LUiGi-H* won both this match and the match shown in Figure 9.13, on the map Challenger.

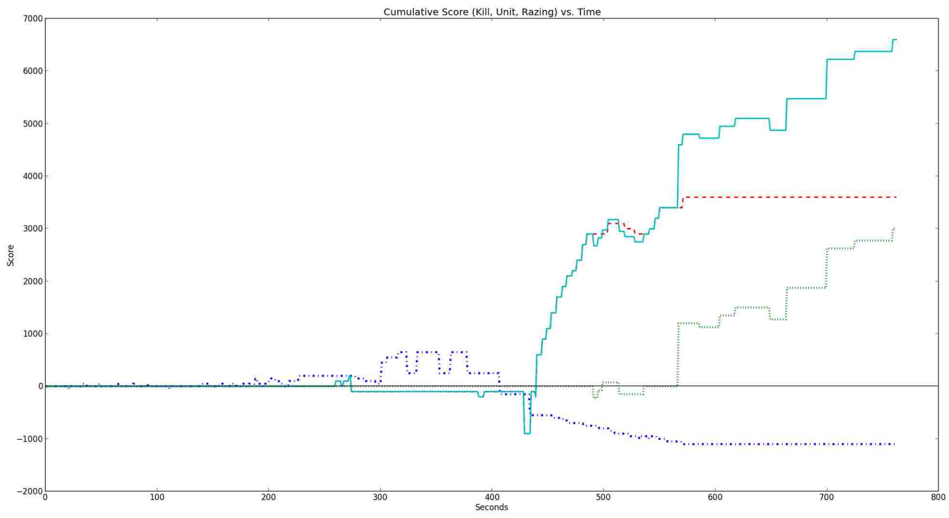


Figure 9.13: *LUiGi-H* vs. *LUiGi* on Challenger

Figure 9.13 shows *LUiGi-H* vs. *LUiGi* on the Challenger map. *LUiGi-H* produces slightly more units in the beginning but towards the end falls behind *LUiGi* . This graph shows a fairer matchup in unit strength. Both the razing score and kill score show *LUiGi-H* outperforming the ablation: *LUiGi* .

LUiGi-H used h-plans with multiple levels of expectations which allowed a more coordinated effort of the primitive actions of a plan. In the situation where *LUiGi-H* and *LUiGi* were executing plans composed of the same primitive actions, in the event of a discrepancy, *LUiGi-H* would trigger discrepancy detection that would reconsider the broader strategy (the entire h-plan of which the primitive actions were composed from) while *LUiGi* would only change plans related to the single level-0 plan that was affected by the discrepancy. This allows *LUiGi-H* more control in executing high level strategies, such as that depicted in the example in Figure 1.

A non-trivial task in running this experiment was ensuring that each bot produced roughly the equivalent strength of units (shown in the graph as unit score). While we were unable to meet this ideal in our experiments precisely, including the unit score in the graphs helps identify the chances that a win was more likely because of sheer strength vs. strategy.

We leave out the result from Volcanis due to a loss from a delay due to the reasoning over the ontology. The average time taken by each agent to reason over the ontology is about 1-2 seconds. This is the crucial part of the discrepancy detection step of the GDA cycle. A delay in the reasoning means that discrepancy detection will be delayed. During the match on Volcanis, at the first attack by *LUiGi* on *LUiGi-H* the reasoning hangs and causes discrepancy detection to respond late and fail to change goals before a building is destroyed. This causes *LUiGi-H* a big setback in the beginning of the match and results in a loss of the game. This issue is due to the fact that at any given point in time there are a few hundred atoms in the state (and thus ontology), with greater numbers of atoms during attacks (because the agent now has all the atoms of its enemy units which it can now see). Optimizing the ontology for both reasoning and state space is one possibility for future improvement: an improvement in reasoning time would increase the rate of discrepancy detection. This also demonstrates that even though the GDA cycle is being performed every few seconds while the bot is issuing a few hundred actions per minute, GDA is still beneficial due to the ability to generate and reason about high level strategies.

9.2.5 Metacognitive Expectations for Long Duration Missions

Figure 9.14 shows the results of the three agents averaged over 10 runs. Each run places randomly the beacons and the sand pits with the agent starting in the center (see Figure 9.2 for an example). The x axis is the cumulative number of beacon-activation goals achieved and the y axis is the cumulative execution cost.

After 500 actions, wind begins blowing east with a strength of 3 (the agent will be pushed an additional 3 tiles), followed with another increase in wind to 4 after 1,500 actions. Prior to wind, the performance of all the agents is the same because they are able to plan around all sand pits (roughly when y = 500). However, once wind is introduced, the agents may be blown into a sand pit and must execute a series of push actions to free themselves, thus raising their execution costs to activate beacons.

All agents exhibit a similar performance for the first 75 goals, which takes about 800 actions. Although wind is introduced after 500 actions, it takes some time to see the benefit because not all actions are affected by wind. (When wind is blowing east, only the move-east action is affect. Thus goals that do not require navigating east will not see any impact from wind).

As the agent's continue to operate, we see that the GDA agent performs better than the re-planning agent, and the metacognition agent performing better than the GDA agent. This experiment shows that even when the effects of a single action change in the environment, metacognition improves performance by identifying a cognitive failure and addressing a limitation in the agent's own knowledge.

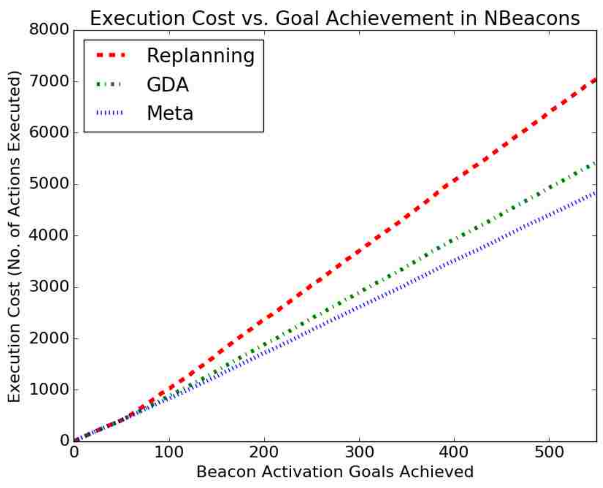


Figure 9.14: Average Execution Cost in NBeacons

Part III

Conclusions

Chapter 10

Future Work

A core tenet of robust autonomy is detecting expectations. The contributions of chapters 4-9 open up multiple avenues for future research. This chapter discusses these areas, particularly expanding the idea of metacognitive expectations.

The first area of future work involves informed expectations combined 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.

Chapter 5 introduced the guiding sensing problem, for which informed expectations performs best but does not guarantee minimal sensing costs (i.e., Condition 3 of the guiding sensing problem definition is not met). For future work, we would like to explore solutions in the context of GDA agents, that either meet Condition 3 of the guiding sensing problem or provide a better approximation than informed expectations. We would also like to explore agents that can decide whether or not to perform sensing between each plan action and at the time of believed goal achievement. Essentially, these agents can vary the frequency of sensing (e.g., instead of sensing per each plan action, perform sensing every n actions). As such, Informed and Eager expectations would no longer guarantee that a believed to be true goal is actually true. Since goals must then be confirmed by additional sensing at the time of presumed goal completion, future work is needed to identify for which, if any, sensing frequencies there exists an improvement in minimizing overall sensing cost.

Building off of chapters 6 and 7, an interesting area of future research is to explore reasoning with GDA elements as ontologies might change over time. For example, imagine a game that allows players to change the terrain over time (e.g., constructing a bridge between two regions previously disconnected). Such a capability to reason with these changing elements would be of particular interest for GDA agents that interact for long-durations in an environment (e.g., an agent interacting in a persistent world).

For future work we will study automated generation of metacognitive expectations. We will explore two potential ways to do this. First, to express the mental actions in a declarative language that enables inference over the outcomes of these actions. This will enable agents, akin to how expectations are generated at the ground level by GDA systems (e.g., (Dannenhauer and Muñoz-Avila, 2015b)), to generate the metacognitive expectations automatically. Second, to learn expectations from traces of mental states. This will enable us to side-step the potentially difficult task of expressing mental actions such as ”PLAN” in a declarative language. On the other hand it may require metacognitive agents that are capable of reasoning with multiple plausible expectations.

10.1 Towards Domain-Independent Metacognitive Expectations

Chapter 8 introduces metacognitive expectations followed by an evaluation in Section 9.2.5. The meta expectation used in the evaluation is the $EX_M(s_{M_i}, Explanation, s_{Mi+1})$, which is a function that checks $v_{explanations} \neq \emptyset$ in s_{Mi+1} whenever $v_{discrepancies} \neq \emptyset$ in s_{M_i} (recall this expectation is informally described as: *when the agent encounters a discrepancy, then the agent will generate an explanation for this discrepancy*). Since the $EX_M(s_{M_i}, Explanation, s_{Mi+1})$ expectation is not concerned with the values of $v_{discrepancies}$ or $v_{explanations}$ but instead whether or not they are empty, we imply that this is a domain-independent metacognitive expectation. Domain-independent metacognitive expectations should be able to be applicable to any domain with the only requirement being the agent has a cognitive level explanation process. Hence we describe here another domain, the Seasonal Construction Domain, which may demonstrate that this metacognitive expectation for explanation is useful and truly domain-independent.

10.1.1 Seasonal Construction Domain

The proposed seasonal construction domain is an extension of the arsonist domain (Paisner et al. (2013), see Section 9.1.3). In the arsonist domain, an unseen arsonist lights blocks on fire and prevents the agent from building towers. In order to deal with the unexpected fires, the agent must put out the fires before continuing (it is not possible for the agent to stack a block on top of a block that is currently on fire). The difference of the seasonal construction domain from the arsonist domain, is that over time (as seasons change from winter to summer) fire becomes more aggressive and spreads more easily. Specifically, now when an agent tries to put out a fire, the agent must put out the base of the fire or the fire will remain. Specifically, when two blocks are on fire, the agent must put out the bottommost block first, and then continue putting out the rest of the fires. Otherwise attempts to put out the fire will fail. However, since the agent is operational in all seasons, this change will take place over time (similar to how wind changed over time in the nbeacons domain, see Section 9.1.6). Fires are initially started by the unseen arsonist from the original arsonist domain.

We hypothesize that the same three agents from NBeacons domain experiment (Section 9.2.5) will demonstrate relatively similar results in the seasonal construction domain. All agents will operate effectively during the winter construction season. As fires change to become more aggressive in the summer, the first agent (a replanning only agent) will attempt to put out fires and fail when those fires having the blocks underneath them are on fire. Figure **??** shows the execution of putoutfire for a topmost block when both blocks are on fire. In winter, fire is put out but in summer, the fire remains because the block underneath (block A) is on fire. The agent will then replan but that plan is likely to include the same putoutfire action, and the agent will become stuck trying to create plans with the no longer effective putoutfire action (because the action is targeting the wrong block). Agent one will perform goal reasoning after it finishes executing the plan and realizes the goal hasn't been met.

The second agent will perform goal driven autonomy when detecting that the putoutfire(B) action fails. In the case, the agent will perform replanning immediately, except it will still not learn that the putoutfire action is no longer compatible. The resulting

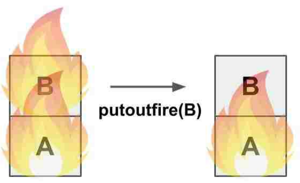


Figure 10.1: Result of putoutfire(B) during Winter

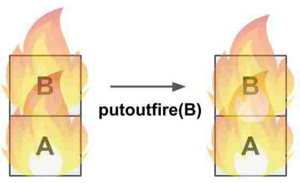


Figure 10.2: Result of putoutfire(B) during Summer

performance of agent two (GDA agent) will likely be similar to agent one because the agent will generate the same plans.

The third agent, equipped with the metacognitive expectation $EX_M(s_{M_i}, Explanation, s_{Mi+1})$, will perform metacognition when the gda agent fails to explain the discrepancy of the failure of the putoutfire(B) action. In this case, the metacognition will produce an explanation that the action putoutfire(B) is no longer applicable. In this case, the agent will generate a learning goal to update its planning operator putoutfire. Figure 10.3 shows the original putoutfire action followed by the operators to be learned in Figures 10.4 to 10.6. Because of the nature of the change in fire behavior, two new operators would need to be learned in order to cover the new situations for fire behavior. (Note: The operator in Figure 10.5 is included for clarity for the reader, this operator would not need to be included in an implemented planner since the effects are empty). Figure 10.4 represents the putoutfire operator for the situation when a block is on fire and is on top of another block which is not on fire. Figure 10.5 represents the putoutfire operator when the target block is on fire and the block underneath it is also on fire. Figure 10.6 represents the situation when the targeted block is on the table and is on fire.

Since the third agent, equipped with metacognitive expectations would be able to detect and update its operators with the ones shown in Figures 10.4 to 10.6, the agent will be able to resume achieving its goals. We hypothesis that only the this agent will be able to recover from the change in fire, and that is recovery process is triggered via the metacognitive expectation described earlier. Thus, it seems that the same metacognitive expectation is applicable in both nbeacons and the seasonal construction domain. The following section proposes other metacognitive expectations to be explored in future research.

10.1.2 Potential Domain-Independent Metacognitive Expectations

We believe that a rich area for future work lies in discovering other domain-independent metacognitive expectations. The following table lists other possible meta expectations and corresponding informal descriptions.

The first metacognitive expectation regards planning as a mental action that should always produce a plan given an unachieved goal. Therefore the size of the plan and the presence of the goal are the only requirements for this expectation. The second mental action involves a sequence of two mental actions: Acting and Perception. During acting, the agent will execute one or more actions, and the effects of those actions should be perceived by the agent in the subsequent perception mental action. The third proposed

```
operator(**putoutfire**,
args = [(blk, BLOCK)],
preconditions = [
   condition(onfire, [blk])],
results = [
   condition(onfire, [blk], negate = True)])
```

Figure 10.3: Original *putoutfire* operator definition

```
operator(putoutfire_1,
args = [(blk, BLOCK), (blk2, BLOCK)],
preconditions = [
   condition(onfire, [blk])],
   condition(on, [blk], [blk2]),
   condition(onfire, [blk2], negate = True)],
results = [
   condition(onfire, [blk], negate = True)])
```

Figure 10.4: First learned operator for a single fire

```
operator(putoutfire_2,
args = [(blk, BLOCK), (blk2, BLOCK)],
preconditions = [
   condition(onfire, [blk])],
   condition(on, [blk], [blk2]),
   condition(onfire, [blk2])],
results = [ ])
```

Figure 10.5: Second learned operator for two fires

```
operator(putoutfire_3,
args = [(blk, BLOCK)],
preconditions = [
   condition(onfire, [blk])],
   condition(ontable, [blk])],
results = [
   condition(onfire, [blk], negate = True)])
```

Figure 10.6: Third learned operator for a single block

#	**Mental Action**	**Informal Description of Expectation**
1	Planning	Given an unachieved goal, the mental action for planning will generate a plan of size $>= 1$
2	Acting, Perception	Given an action and a current world state, the perceived world state immediately followed an action will consist of the results of that action
3	Evaluation, Intend	Given an empty set of current goals the following goal selection (Intend) mental action will select a new goal. More specifically: the goal set following an intend mental action should have size $>= 1$

Table 10.1: Proposed Metacognitive Expectations for Future Evaluation

metacognitive expectation is that goal selection (Intend) will ensure the set of goals for the agent to pursue is non-empty.

It is important to note that these expectations trigger the identification of an anomaly. However, the cause of failure may not be known. For example, when planning fails, it could be due to incorrect domain knowledge or it could be due to an unachievable goal. Another example is that when perception fails to observe a world state containing the effects of the recently executed actions, the problem may lie in the agents action execution components (i.e., perhaps the agent's robotic arm is broken) or in the agent's perception components (perhaps the agent's camera sensor is no longer functioning properly). Cox (1996) presents a taxonomy of these kinds of failures and their causes at the cognitive level. The work of Chapter 8 presents explicit notions of expectations about the cognitive level. The violation of these expectations acts as a trigger for diagnosis followed by corrective measures. There may be varied responses for the same expectation failure; deciding the appropriate response is itself a significant area for future research.

Chapter 11

Summary

The road to robust autonomy is paved by improved self-monitoring. Goal reasoning and metacognitive approaches have both been shown to increase robustness of agents. Goal reasoning addresses robustness at a higher level than agents who only re-plan in response to an anomaly. This is warranted in domains where the agent's goal may become invalid and attempts at re-planning are futile. Metacognitive approaches increase robustness by dealing with anomalies that are internal to the agent. We have only begun to scratch the surface of metacognitive agents in Chapter 8 and there remains considerable and fruitful future research in that area (as mentioned in Chapter 10). Both goal reasoning and metacognitive approaches rely on a strong discrepancy detection and this thesis contributes multiple approaches for better expectations.

In Chapter 6 we presented abstract concepts to be used as expectations that enable high level planning in the complex environment of the Real-Time Strategy game Starcraft: Broodwar. By introducing the concepts of *controlled-region* and *contested-region*, along with the axioms and rules to allow their inference, we were able to create high level actions leading to high level plans. High level planning in Starcraft is needed because the current state of the art involves agents that primarily focus on the most efficient production of combat units and optimizing local fighting strategies. However, human players still easily defeat these agents, in part because they are skilled at using multiple groups of units in higher-level strategies. The execution of such plans, and response when they fail (as they are likely to do at least some of the time given the adversarial nature of the game) requires high level expectations that encode many possible game states into a single expectation. We believe this approach extends to other complex and large domains where many combinations of states represent an anomalous event or failed action. Using semantic inference over these expectations enable higher level behavior.

We went on to show that hierarchical plans based off of the plans in Chapter 4 needed their own version of hierarchical expectations in Chapter 5. An agent using a case-base of hierarchical-plans with corresponding hierarchical-expectations was able to execute plans at a higher level and defeat the agent using only flat plans.

In Chapter 6 we introduced informed expectations, which bridge the gap between immediate expectations and state expectations. Immediate expectations fail to identify discrepancies from past actions, thereby failing to guarantee an agent will identify all relevant changes in the environment leading to goal failure. State expectations, the sequence of states corresponding to the results of each action of the plan, cover too wide an area for discrepancy detection. In large, complex domains, they falsely trigger discrepancy detection when it is not needed should anything irrelevant change. Immediate expectations accumulates the effects of actions as they are propagated through the plan, resulting in only those effects that are necessary in the future states or the goal state. Informed expectations rely on the assumption that the effects of actions are relevant to the rest of the plan and the goal. In situations where 1 or more action's effects may be irrelevant to future plan actions or the goal, goal regression can be used to trim these unnecessary actions. This may happen in situations where actions are learned over time.

Chapter 7 extends the results of informed expectations in Chapter 6 for domains where exploration is required to find objects needed to achieve the goal. In these kinds of domains, an agent is unable to plan far ahead and is constantly changing its goal depending upon what is available in the environment. In such partially observable worlds, goal regression is not applicable due to short plans. Informed expectations keep track of the accumulated action effects which minimizes the sensing costs while enabling discrepancy detection to detect if a change in the environment warrants an agent changing its goal.

Chapter 8 presents a formalism for metacognitive expectations used within the MIDCA cognitive architecture. Instead of expectations of the world, meta expectations are about cognition. The formalism for meta expectations defines functions which should evaluate to true given the values of mental variables. Most cognitive processes (i.e., planning, goal selection, perception, etc.) make use of a fairly small set (i.e., less than ten) core mental variables. The values of these mental variables make up the current mental state of the agent at any given point in time and cognitive processes change the values of the mental variables. For example, planning will change the mental variable Π (representing the agent's current plan) from a null value to recently made plan, to achieve the agents current goals (stored in the mental variable G). Meta expectations describe expectations that are concerned with these variables and define conditions that represent anomalous behavior.

The primary contribution of Chapter 8 and the meta expectations is the formalism and a mental expectation regarding the cognitive explanation process. This meta expectation is fairly general (in that it only cares whether an explanation was produced;

it is domain independent) and we believe there is a small set of meta expectations that are fairly general and useful in any domain (in a similar way that domain-independed heuristics are useful in hueristic search planners). A major area for future research is identifying more of the meta expectations that are part of this core set.

Chapter 9 provides details on the experimental domains and corresponding results of the approaches described in Chapter 4-8. Briefly, informed expectations outperforms all other expectation approaches in fully observable and dynamic environments (except goal regression for which is equal to informed expectations barring domain models for actions with irrelevant effects, in which case goal regression is ideal). In dynamic and partially observable environments where exploration is needed to achieve goals, informed expectations performs best (in these cases goal regression is not available because planning is not able to generate plans for far in the future). Additionally, inferred concepts enable the means to execute high level plans in complex real-time strategy games such as Starcraft: Broodwar and hierarchical approaches outperform flat plans. Finally, with meta expectations, an agent can identify when its own cognitive explanation processes have failed, leading to detection of anomalies in own system. The use of meta expectations allows an agent to update its internal domain model in domains that change over time, leading to increased performance (goals achieved).

Expectations can be easily overlooked as part of a larger autonomous system but play an immense role in robust autonomy. This work contributes multiple approaches to expectations for self-monitoring agents (agents that perform their own monitoring of their actions or internal processes). This work also sets the foundation for multiple areas of future research. The guiding sensing problem (Chapter 5) is not likely to see a better knowledge-free solution than informed expectations (although we do not prove this here). Improvements are likely to come from using additional knowledge. This research also opens up a large area of future research in meta cognition. We have started the effort by formalizing cognitive processes and the representation of meta expectations, and define the first concrete domain-independent meta-expectation. Applying data mining approaches to learning domain-specific meta expectations as well as identifying other domain-independent meta expectations seem to be fruitful and important areas for future research. A core set of domain-independent meta expectations could signal anomalous behavior in agents operating in any domains.

Appendix A

Ontology for Starcraft: Broodwar

This ontology was originally created as part of a group project during the Semantic Web Topics course during the Spring semester of 2013. Our group consisted of myself, Will West, and Kostas Hatalis. The ontology was then modified and the version used in the *LUiGi* and *LUiGi-H* agents (Chapters 6 and 7) is described here. The Description Logic (DL) Expressivity is ALCROIQ(D).

Boolean values ”true” and ”false” are of type: http://www.w3.org/2001/XMLSchema#boolean. These values are only used in the *UnderAttackUnit* and *NotUnderAttackUnit* classes.

Classes

Ability

Academy

Academy $\sqsubseteq$ BasicStructure

Addon

Addon $\sqsubseteq$ Unit

AdvancedStructure

AdvancedStructure $\sqsubseteq$ Structure

DisjointUnion Armory Factory ScienceFacility Starport

Armory

Armory $\sqsubseteq$ AdvancedStructure

Barracks

Barracks $\sqsubseteq$ BasicStructure

BasicStructure

BasicStructure $\sqsubseteq$ Structure

Battlecruiser

Battlecruiser $\sqsubseteq$ FlyingVehicle

Bunker

Bunker $\sqsubseteq$ BasicStructure

Chokepoint

Chokepoint $\equiv$ Chokepoint $\sqcap =$ connectedTo Region

Chokepoint $\sqsubseteq$ Thing

CommandCenter

CommandCenter $\sqsubseteq$ BasicStructure

ComsatStation

ComsatStation $\sqsubseteq$ Addon

ContestedChokepoint

ContestedChokepoint $\sqsubseteq$ Chokepoint

ContestedRegion

ContestedRegion $\equiv$ = containsUnitOf Player

ContestedRegion $\sqsubseteq$ KnownRegion

ContestedRegion $\sqsubseteq \neg$ ControlledRegion

ControlTower

ControlTower $\sqsubseteq$ Addon

ControlledRegion

ControlledRegion $\sqsubseteq$ KnownRegion

ControlledRegion $\sqsubseteq \neg$ ContestedRegion

CovertOps

CovertOps $\sqsubseteq$ Addon

Dropship

Dropship $\sqsubseteq$ FlyingVehicle

Enemy

Enemy $\sqsubseteq$ Player

EnemyUnit

EnemyUnit $\equiv$ Unit $\sqcap$ $\exists$ isOwnedBy $\{$player1$\}$

EnemyUnit $\sqsubseteq$ Unit

EngineeringBay

EngineeringBay $\sqsubseteq$ BasicStructure

Factory

Factory $\sqsubseteq$ AdvancedStructure

FightingUnit

FightingUnit $\sqsubseteq$ Unit

DisjointUnion FlyingVehicle GroundVehicle SCV Soldier

FightingUnitAbility

FightingUnitAbility $\sqsubseteq$ Ability

Firebat

Firebat $\sqsubseteq$ Soldier

FlyingVehicle

FlyingVehicle $\sqsubseteq$ FightingUnit

DisjointUnion Battlecruiser Dropship ScienceVessel Valkyrie Wraith

FriendlyUnit

FriendlyUnit $\equiv$ Unit $\sqcap$ $\exists$ isOwnedBy $\{$player0$\}$

FriendlyUnit $\sqsubseteq$ Unit

Ghost

Ghost $\sqsubseteq$ Soldier

Goliath

Goliath $\sqsubseteq$ GroundVehicle

GroundVehicle

GroundVehicle $\sqsubseteq$ FightingUnit

DisjointUnion Goliath Tank Vulture

KnownRegion

KnownRegion $\sqsubseteq$ Region

KnownRegion $\sqsubseteq \neg$ UnknownRegion

DisjointUnion ContestedRegion ControlledRegion

MachineShop

MachineShop $\sqsubseteq$ Addon

Marine

Marine $\sqsubseteq$ Soldier

Match

Medic

Medic $\sqsubseteq$ Soldier

MissileTurret

MissileTurret $\sqsubseteq$ BasicStructure

NotUnderAttackUnit

NotUnderAttackUnit $\equiv$ Unit $\sqcap$ $hasValue$ isBeingAttacked ”false”

NotUnderAttackUnit $\sqsubseteq$ Unit

NotUnderAttackUnit $\sqsubseteq \neg$ UnderAttackUnit

NuclearMissile

NuclearMissile $\sqsubseteq$ FlyingVehicle

NuclearSilo

NuclearSilo $\sqsubseteq$ Addon

PhysicsLab

PhysicsLab $\sqsubseteq$ Addon

Player

Player $\sqsubseteq$ Thing

Refinery

Refinery $\sqsubseteq$ BasicStructure

Region

Region $\sqsubseteq$ Thing

DisjointUnion KnownRegion UnknownRegion

SCV

SCV $\sqsubseteq$ FightingUnit

ScienceFacility

ScienceFacility $\sqsubseteq$ AdvancedStructure

ScienceVessel

ScienceVessel $\sqsubseteq$ FlyingVehicle

SiegeTank

SiegeTank $\sqsubseteq$ GroundVehicle

Soldier

Soldier $\sqsubseteq$ FightingUnit

DisjointUnion Firebat Ghost Marine Medic

SpellScannerSweep

SpellScannerSweep $\sqsubseteq$ Unit

Starport

Starport $\sqsubseteq$ AdvancedStructure

Structure

Structure $\sqsubseteq$ Unit

DisjointUnion Academy AdvancedStructure Barracks Bunker CommandCenter EngineeringBay MissileTurret Refinery SupplyDepot

StructureAbility

StructureAbility $\sqsubseteq$ Ability

SupplyDepot

SupplyDepot $\sqsubseteq$ BasicStructure

Tank

Tank $\sqsubseteq$ GroundVehicle

Thing

UnderAttackUnit

UnderAttackUnit $\equiv$ Unit $\sqcap$ $hasValue$ isBeingAttacked ”true”

UnderAttackUnit $\sqsubseteq$ Unit

UnderAttackUnit $\sqsubseteq$ $\neg$ NotUnderAttackUnit

Unit

Unit $\sqsubseteq$ Thing

UnknownRegion

UnknownRegion $\sqsubseteq$ Region

UnknownRegion $\sqsubseteq$ $\neg$ KnownRegion

Valkyrie

Valkyrie $\sqsubseteq$ FlyingVehicle

Vulture

Vulture $\sqsubseteq$ GroundVehicle

Wraith

Wraith $\sqsubseteq$ FlyingVehicle

Object properties

builds

$\exists$ builds Thing $\sqsubseteq$ Structure

$\top \sqsubseteq \forall$ builds (FlyingVehicle $\sqcup$ GroundVehicle $\sqcup$ Structure)

DisjointObjectProperties builds trains

canBeBuiltBy

canBeBuiltBy $\equiv$ canBuild^{-}

canBeTrainedBy

$\sqsubseteq$ topObjectProperty

canBeTrainedBy $\equiv$ canTrain^{-}

canBuild

canBeBuiltBy $\equiv$ canBuild^{-}

$\exists$ canBuild Thing $\sqsubseteq$ Player

$\top \sqsubseteq \forall$ canBuild StructureAbility

canTrain

$\sqsubseteq$ topObjectProperty

$\text{canBeTrainedBy} \equiv \text{canTrain}^-$

$\exists$ canTrain Thing $\sqsubseteq$ Player

$\top \sqsubseteq \forall$ canTrain FightingUnitAbility

connectedTo

$\text{connectedTo} \equiv \text{hasChokepoint}^-$

$\exists$ connectedTo Thing $\sqsubseteq$ Chokepoint

$\top \sqsubseteq \forall$ connectedTo Region

contains

$\text{contains} \equiv \text{isInRegion}^-$

containsUnitOf

$\text{containsUnitOf} \equiv \text{hasUnitIn}^-$

$\exists$ containsUnitOf Thing $\sqsubseteq$ Region

$\top \sqsubseteq \forall$ containsUnitOf Player

controls

$\exists$ controls Thing $\sqsubseteq$ Player

$\top \sqsubseteq \forall$ controls ControlledRegion

hasChokepoint

$\text{connectedTo} \equiv \text{hasChokepoint}^-$

$\exists$ hasChokepoint Thing $\sqsubseteq$ Region

$\top \sqsubseteq \forall$ hasChokepoint Chokepoint

hasEnemyUnit

$\exists$ hasEnemyUnit Thing $\sqsubseteq$ Player

$\top \sqsubseteq \forall$ hasEnemyUnit Unit

hasHumanCompetitor

$\sqsubseteq$ topObjectProperty

hasPlayer

$\sqsubseteq$ topObjectProperty

$\top \sqsubseteq \forall$ hasPlayer Player

hasPresenceIn

$\exists$ hasPresenceIn Thing $\sqsubseteq$ Player

$\top \sqsubseteq \forall$ hasPresenceIn Region

hasStructure

hasStructure $\equiv$ hasUnit

$\top \sqsubseteq \leq 1$ hasStructure^{-} Thing

$\exists$ hasStructure Thing $\sqsubseteq$ Player

$\top \sqsubseteq \forall$ hasStructure Unit

hasUnit

hasStructure $\equiv$ hasUnit

hasUnit $\equiv$ isOwnedBy^{-}

$\exists$ hasUnit Thing $\sqsubseteq$ Player

$\top \sqsubseteq \forall$ hasUnit Unit

hasUnitIn

containsUnitOf $\equiv$ hasUnitIn^{-}

$\exists$ hasUnitIn Thing $\sqsubseteq$ Player

$\top \sqsubseteq \forall$ hasUnitIn Region

isConnectedToRegionOne

$\exists$ isConnectedToRegionOne Thing $\sqsubseteq$ Chokepoint

$\top \sqsubseteq \forall$ isConnectedToRegionOne Region

isConnectedToRegionTwo

$\exists$ isConnectedToRegionTwo Thing $\sqsubseteq$ Chokepoint

$\top \sqsubseteq \forall$ isConnectedToRegionTwo Region

isInRegion

contains $\equiv$ isInRegion^{-}

$\top \sqsubseteq \leq 1$ isInRegion Thing

$\exists$ isInRegion Thing $\sqsubseteq$ Unit

$\top \sqsubseteq \forall$ isInRegion Region

isOwnedBy

hasUnit $\equiv$ isOwnedBy^{-}

playedAs

$\top \sqsubseteq \forall$ playedAs Player

topObjectProperty

trains

$\exists$ trains Thing $\sqsubseteq$ Structure

$\top \sqsubseteq \forall$ trains (SCV $\sqcup$ Soldier)

DisjointObjectProperties builds trains

unlocks

$\exists$ unlocks Thing $\sqsubseteq$ Structure

$\top \sqsubseteq \forall$ unlocks Unit

Data properties

hasArmor

$\exists$ hasArmor Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Unit

$\top \sqsubseteq \forall$ hasArmor Datatypehttp://www.w3.org/2001/XMLSchema#int

hasAttackDamage

$\exists$ hasAttackDamage Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ FightingUnit

$\top \sqsubseteq \forall$ hasAttackDamage Datatypehttp://www.w3.org/2001/XMLSchema#int

hasAttackRange

$\exists$ hasAttackRange Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ FightingUnit

$\top \sqsubseteq \forall$ hasAttackRange Datatypehttp://www.w3.org/2001/XMLSchema#int

hasBuildTime

$\exists$ hasBuildTime Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Unit

$\top \sqsubseteq \forall$ hasBuildTime Datatypehttp://www.w3.org/2001/XMLSchema#int

hasCenterX

$\exists$ hasCenterX Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Region

$\top \sqsubseteq \forall$ hasCenterX Datatypehttp://www.w3.org/2001/XMLSchema#int

hasCenterY

$\exists$ hasCenterY Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Region

$\top \sqsubseteq \forall$ hasCenterY Datatypehttp://www.w3.org/2001/XMLSchema#int

hasChokepointCenterX

$\exists$ hasChokepointCenterX Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Chokepoint

$\top \sqsubseteq \forall$ hasChokepointCenterX Datatypehttp://www.w3.org/2001/XMLSchema#int

hasChokepointCenterY

$\exists$ hasChokepointCenterY Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Chokepoint

$\top \sqsubseteq \forall$ hasChokepointCenterY Datatypehttp://www.w3.org/2001/XMLSchema#int

hasChokepointId

$\exists$ hasChokepointId Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Chokepoint

$\top \sqsubseteq \forall$ hasChokepointId Datatypehttp://www.w3.org/2001/XMLSchema#int

hasCurrentHitPoints

$\exists$ hasCurrentHitPoints Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Unit

$\top \sqsubseteq \forall$ hasCurrentHitPoints Datatypehttp://www.w3.org/2001/XMLSchema#int

hasElapsedGameTime

$\top \sqsubseteq \forall$ hasElapsedGameTime Datatypehttp://www.w3.org/2001/XMLSchema#dateTime

hasEnergy

$\exists$ hasEnergy Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ FightingUnit

$\top \sqsubseteq \forall$ hasEnergy Datatypehttp://www.w3.org/2001/XMLSchema#int

hasGasCost

$\exists$ hasGasCost Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Unit

$\top \sqsubseteq \forall$ hasGasCost Datatypehttp://www.w3.org/2001/XMLSchema#int

hasKillScore

$\exists$ hasKillScore Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Player

$\top \sqsubseteq \forall$ hasKillScore Datatypehttp://www.w3.org/2001/XMLSchema#int

hasMapHeight

$\exists$ hasMapHeight Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Match

$\top \sqsubseteq \forall$ hasMapHeight Datatypehttp://www.w3.org/2001/XMLSchema#int

hasMapName

$\exists$ hasMapName Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Match

$\top \sqsubseteq \forall$ hasMapName Datatypehttp://www.w3.org/2001/XMLSchema#string

hasMapWidth

$\exists$ hasMapWidth Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Match

$\top \sqsubseteq \forall$ hasMapWidth Datatypehttp://www.w3.org/2001/XMLSchema#int

hasMaxHitPoints

$\exists$ hasMaxHitPoints Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Unit

$\top \sqsubseteq \forall$ hasMaxHitPoints Datatypehttp://www.w3.org/2001/XMLSchema#int

hasMaxNumOfPlayers

$\top \sqsubseteq \forall$ hasMaxNumOfPlayers Datatypehttp://www.w3.org/2001/XMLSchema#int

hasMineralCost

$\exists$ hasMineralCost Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Unit

$\top \sqsubseteq \forall$ hasMineralCost Datatypehttp://www.w3.org/2001/XMLSchema#int

hasName

$\top \sqsubseteq \forall$ hasName Datatypehttp://www.w3.org/2001/XMLSchema#string

hasNumberOfPlayers

$\top \sqsubseteq \forall$ hasNumberOfPlayers Datatypehttp://www.w3.org/2001/XMLSchema#int

hasNumberOfRegions

$\top \sqsubseteq \forall$ hasNumberOfRegions Datatypehttp://www.w3.org/2001/XMLSchema#int

hasPlayerId

$\top \sqsubseteq \leq 1$ hasPlayerId

$\exists$ hasPlayerId Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Player

$\top \sqsubseteq \forall$ hasPlayerId Datatypehttp://www.w3.org/2001/XMLSchema#int

hasRateOfFire

$\exists$ hasRateOfFire Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Fightin-gUnit

$\top \sqsubseteq \forall$ hasRateOfFire Datatypehttp://www.w3.org/2001/XMLSchema#int

hasRazingScore

$\exists$ hasRazingScore Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Player

$\top \sqsubseteq \forall$ hasRazingScore Datatypehttp://www.w3.org/2001/XMLSchema#int

hasRegionId

$\exists$ hasRegionId Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Region

$\top \sqsubseteq \forall$ hasRegionId Datatypehttp://www.w3.org/2001/XMLSchema#int

hasSize

$\exists$ hasSize Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Unit

$\top \sqsubseteq \forall$ hasSize Datatypehttp://www.w3.org/2001/XMLSchema#int

hasSpeed

$\exists$ hasSpeed Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ FightingUnit

$\top \sqsubseteq \forall$ hasSpeed Datatypehttp://www.w3.org/2001/XMLSchema#int

hasSpentGas

$\exists$ hasSpentGas Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Player

$\top \sqsubseteq \forall$ hasSpentGas Datatypehttp://www.w3.org/2001/XMLSchema#int

hasSpentMinerals

$\exists$ hasSpentMinerals Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Player

$\top \sqsubseteq \forall$ hasSpentMinerals Datatypehttp://www.w3.org/2001/XMLSchema#int

hasStartDate

$\top \sqsubseteq \forall$ hasStartDate Datatypehttp://www.w3.org/2001/XMLSchema#dateTime

hasSupplyTotal

$\exists$ hasSupplyTotal Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Player

$\top \sqsubseteq \forall$ hasSupplyTotal Datatypehttp://www.w3.org/2001/XMLSchema#int

hasTileSet

$\top \sqsubseteq \forall$ hasTileSet Datatypehttp://www.w3.org/2001/XMLSchema#string

hasTravelType

$\exists$ hasTravelType Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ FightingUnit

$\top \sqsubseteq \forall$ hasTravelType Datatypehttp://www.w3.org/2001/XMLSchema#string

hasUnitId

$\top \sqsubseteq \leq 1$ hasUnitId

$\exists$ hasUnitId Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Unit

$\top \sqsubseteq \forall$ hasUnitId Datatypehttp://www.w3.org/2001/XMLSchema#int

hasUnitScore

$\exists$ hasUnitScore Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Player

$\top \sqsubseteq \forall$ hasUnitScore Datatypehttp://www.w3.org/2001/XMLSchema#int

hasWebsite

$\top \sqsubseteq \forall$ hasWebsite Datatypehttp://www.w3.org/2001/XMLSchema#anyURI

hasWidth

$\exists$ hasWidth Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Chokepoint

$\top \sqsubseteq \forall$ hasWidth Datatypehttp://www.w3.org/2001/XMLSchema#double

hasXCoord

$\exists$ hasXCoord Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Unit

$\top \sqsubseteq \forall$ hasXCoord Datatypehttp://www.w3.org/2001/XMLSchema#int

hasYCoord

$\exists$ hasYCoord Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Unit

$\top \sqsubseteq \forall$ hasYCoord Datatypehttp://www.w3.org/2001/XMLSchema#int

isBeingAttacked

$\exists$ isBeingAttacked Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Unit

$\top \sqsubseteq \forall$ isBeingAttacked Datatypehttp://www.w3.org/2001/XMLSchema#boolean

sameAs

$\exists$ sameAs Datatypehttp://www.w3.org/2000/01/rdf-schema#Literal $\sqsubseteq$ Thing

$\top \sqsubseteq \forall$ sameAs Datatypehttp://www.w3.org/2001/XMLSchema#anyURI

SWRLs

Region(?r) $\wedge$ Unit(?u) $\wedge$ isInRegion(?u, ?r) $\rightarrow$ KnownRegion(?r)

Player(?p) $\wedge$ Region(?r) $\wedge$ Unit(?u) $\wedge$ isInRegion(?u, ?r) $\wedge$ isOwnedBy(?u, ?p) $\rightarrow$ hasPresenceIn(?p, ?r)

Player(?p1) $\wedge$ Player(?p2) $\wedge$ Region(?r) $\wedge$ hasPresenceIn(?p1, ?r) $\wedge$ hasPresenceIn(?p2, ?r) $\wedge$ differentFrom(?p1, ?p2) $\rightarrow$ ContestedRegion(?r)

Individuals

academyAvailable

armoryAvailable

barracksAvailable

battleCruiserAvailable

bunkerAvailable

commandCenterAvailable

comsatStationAvailable

controlTowerAvailable

covertOpsAvailable

dropshipAvailable

engineeringbayAvaliable

factoryAvailable

firebatAvailable

ghostAvailable

goliathAvailable

machineShopAvailable

marineAvailable

medicAvailable

missileTurretAvailable

nuclearMissileAvailable

nuclearSiloAvailable

physicsLabAvailable

player0

player0 : Player

$\{\text{player0}\} \not\equiv \{\text{player1}\}$

hasPlayerId (player0 ”0” http://www.w3.org/2001/XMLSchema#int)

player1

player1 : Player

$\{\text{player0}\} \not\equiv \{\text{player1}\}$

hasPlayerId (player1 ”1” http://www.w3.org/2001/XMLSchema#int)

refineryAvailable

scienceFacilityAvailable

scienceVesselAvailable

scvAvailable

siegeTankAvailable

spellScannerSweepAvailable

starportAvailable

supplyDepotAvailable

tankAvailable

valkyrieAvailable

vultureAvailable

wraithAvailable

Datatypes

PlainLiteral

anyURI

boolean

dateTime

double

int

string

Bibliography

Aha, D. W., Anderson, T. S., Bengfort, B., Burstein, M., Cerys, D., Coman, A., Cox, M. T., Dannenhauer, D., Floyd, M. W., Gillespie, K., et al. (2015). Goal reasoning: Papers from the acs workshop. Technical report, Georgia Institute of Technology.

Anderson, M., Gomaa, W., Grant, J., and Perlis, D. (2008). Active logic semantics for a single agent in a static world. *Artificial Intelligence*, 172(8-9):1045–1063.

Ayan, N., Kuter, U., Yaman, F., and Goldman, R. (2007). HOTRiDE: Hierarchical ordered task replanning in dynamic environments.

Bäckström, C. and Nebel, B. (1995). Complexity results for sas+ planning. *Computational Intelligence*, 11(4):625–655.

Benson, S. and Nilsson, N. (1993). Reacting, Planning, and Learning in an Autonomous Agent. *Machine intelligence 14*.

Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, 129(1):5–33.

Bouguerra, A., Karlsson, L., and Saffiotti, A. (2007). Active execution monitoring using planning and semantic knowledge. *ICAPS Workshop on Planning and Plan Execution for Real-World Systems*.

Čertickỳ, M. (2014). Real-time action model learning with online algorithm 3 sg. *Applied Artificial Intelligence*, 28(7):690–711.

Chang, H.-C., Dong, L., Liu, F., and Lu, W. F. (2000). Indexing and retrieval in machining process planning using case-based reasoning. *Artificial Intelligence in Engineering*, 14(1):1–13.

Churchill, D. (2013). 2013 AIIDE StarCraft AI Competition Report. `http://webdocs.cs.ualberta.ca/~cdavid/starcraftaicomp/report2013.shtml`.

Coddington, A., Fox, M., Gough, J., Long, D., and Serina, I. (2005). Madbot: A motivated and goal directed robot. In *Proceedings of the National Conference on Artificial Intelligence*, volume 20.

Cox, M. T. (1996). *Introspective multistrategy learning: Constructing a learning strategy under reasoning failure*. PhD thesis, Georgia Institute of Technology, College of Computing, Atlanta. Tech. Rep. No. GIT-CC-96-06.

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

Cox, M. T., Alavi, Z., Dannenhauer, D., Eyorokon, V., Munoz-Avila, H., and Perlis, D. (2016). Midca: A metacognitive, integrated dual-cycle architecture for self-regulated autonomy. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence.* Palo Alto, CA: AAAI Press.

Cox, M. T., Oates, T., Paisner, M., and Perlis, D. (2012). Noting anomalies in streams of symbolic predicates using a-distance. *Advances in Cognitive Systems*, 2:167–184.

Cox, M. T. and Raja, A. (2011). Metareasoning: An introduction. In *Metareasoning: Thinking about thinking*, pages 3–14. MIT Press.

Cox, M. T. and Veloso, M. M. (1998). Goal transformations in continuous planning. In *Proceedings of the 1998 AAAI fall symposium on distributed continual planning*, pages 23–30.

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

Dannenhauer, D. and Muñoz-Avila, H. (2015a). Goal-driven autonomy with semantically-annotated hierarchical cases. In *Case-Based Reasoning Research and Development*, pages 88–103. Springer.

Dannenhauer, D. and Muñoz-Avila, H. (2015b). Raising Expectations in GDA Agents Acting in Dynamic Environments. In *International Joint Conference on Artificial Intelligence (IJCAI-15).*

Dannenhauer, D., Munoz-Avila, H., and Cox, M. T. (2016). Informed Expectations to Guide GDA Agents in Partially Observable Environments. In *International Joint Conference on Artificial Intelligence (IJCAI-16).*

Dannenhauer, D., Munoz-Avila, H., and Cox, M. T. (2017). Metacognitive Expectations. Under Review.

Dunlosky, J., Rawson, K. A., Marsh, E. J., and Nathan, Mitchell J.and Willingham, D. (2013). Improving students learning with effective learning techniques: Promising directions from cognitive and educational psychology. *Psychological Science in the Public Interest*, 14(1):4–58.

Erol, K., Hendler, J., and Nau, D. S. (1994). Htn planning: Complexity and expressivity. In *AAAI*, volume 94, pages 1123–1128.

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.

Flavell, J. H. (1979). Metacognition and cognitive monitoring. a new area of cognitive-development inquiry. *American Psychologist*, 34(10):906–911.

Flavell, J. H. and Wellman, H. M. (1977). Metamemory. In *Perspectives on the Development of Memory and Cognition*, pages 3–33. Lawrence Erlbaum Associates.

Fox, M., Gerevini, A., Long, D., and Serina, I. (2006). Plan Stability: Replanning versus Plan Repair. *ICAPS*.

Fritz, C. and McIlraith, S. A. (2007). Monitoring Plan Optimality During Execution. In *ICAPS*, pages 144–151.

Gerevini, A., Saetti, A., and Serina, I. (2003). Planning through stochastic local search and temporal action graphs in lpg. *Journal of Artificial Intelligence Research*, pages 239–290.

Ghallab, M., Nau, D., and Traverso, P. (2014). The actors view of automated planning and acting: A position paper. *Artificial Intelligence*, 208:1–17.

Gil, Y. and Blythe, J. (2000). How Can a Structured Representation of Capabilities Help in Planning. In *Proceedings of the AAAI–Workshop on Representational Issues for Realworld Planning Systems*.

Goldman, R., Boddy, M., and Pryor, L. (1996). Planning with observations and knowledge. In *AAAI-97 workshop on theories of action, planning and control*.

Hawes, N. (2011). A survey of motivation frameworks for intelligent systems. *Artificial Intelligence*, 175(5-6):1020–1036.

Hawes, N., Hanheide, M., Hargreaves, J., Page, B., Zender, H., and Jensfelt, P. (2011). Home Alone: Autonomous Extension and Correction of Spatial Representations. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3907–3914. IEEE.

Howard, R. A. (1960). Dynamic programming and markov processes..

Ives, Z. G., Florescu, D., Friedman, M., Levy, A., and Weld, D. S. (1999). An adaptive query execution system for data integration. In *ACM SIGMOD Record*, volume 28, pages 299–310. ACM.

Jaidee, U. and Muñoz-Avila, H. (2013). Modeling Unit Classes as Agents in Real-Time Strategy Games. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Jaidee, U., Muñoz-Avila, H., and Aha, D. W. (2011). 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.

Jaidee, U., Muñoz-Avila, H., and Aha, D. W. (2012). Learning and Reusing Goal-Specific Policies for Goal-Driven Autonomy. In *Case-Based Reasoning Research and Development*, pages 182–195. Springer.

Kaelbling, L., Littman, M., and Cassandra, A. (1995). Partially observable markov decision processes for artificial intelligence. *KI-95: Advances in Artificial Intelligence*, pages 1–17.

Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1):99–134.

Kambhampati, S. and Hendler, J. A. (1992). A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence*, 55(2):193–258.

Klenk, M., Molineaux, M., and Aha, D. W. (2013). Goal-driven autonomy for responding to unexpected events in strategy simulations. *Computational Intelligence*, 29(2):187–206.

Knoblock, C. A. (1995). Planning, executing, sensing, and replanning for information gathering. In *In Proceedings Of The Fourteenth International Joint Conference On Artificial Intelligence*.

Knoblock, C. A. (1996). Building a planner for information gathering: A report from the trenches. In *In AIPS-96*. Citeseer.

Kurup, U., Lebiere, C., Stentz, A., and Hebert, M. (2012). Using expectations to drive cognitive behavior. *Twenty-Sixth AAAI Conference on Artificial Intelligence*.

Lee, P. Y. and Cox, M. T. (2002). Dimensional indexing for targeted case-base retrieval: The smirks system. In *FLAIRS Conference*, pages 62–66.

Lotem, A. and Nau, D. S. (2000). New advances in graphhtn: Identifying independent subproblems in large htn domains. In *AIPS*, pages 206–215.

Mei, Y., Lu, Y.-H., Hu, Y. C., and Lee, C. G. (2005). A case study of mobile robot's energy consumption and conservation techniques. In *Advanced Robotics, 2005. ICAR'05. Proceedings., 12th International Conference on*, pages 492–497. IEEE.

Michael, C. and Ashwin, R. (1999). Introspective multistrategy learning: On the construction of learning strategies. *Artificial Intelligence*, 112:1–55.

Mitchell, T. M., Keller, R. M., and Kedar-Cabelli, S. T. (1986). Explanation-based generalization: A unifying view. *Machine learning*, 1(1):47–80.

Molineaux, M. and Aha, D. (2013). Learning Models for Predicting Surprising Events. In *Advances in Cognitive Systems Workshop on Goal Reasoning*.

Molineaux, M. and Aha, D. W. (2014). Learning unknown event models. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*.

Molineaux, M., Klenk, M., and Aha, D. W. (2010). Goal-driven autonomy in a navy strategy simulation. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 1548–1554. AAAI Press.

Molineaux, M., Kuter, U., and Klenk, M. (2012). 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.

Muñoz, H., Paulokat, J., and Wess, S. (1995). *Controlling a nonlinear hierarchical planner using case replay*. Springer.

Muñoz-Avila, H., Aha, D. W., Jaidee, U., Klenk, M., and Molineaux, M. (2010a). Applying goal-driven autonomy to a team shooter game. In *FLAIRS Conference*.

Muñoz-Avila, H., Aha, D. W., Nau, D. S., Weber, R., Breslow, L., and Yamal, F. (2001). Sin: integrating case-based reasoning with task decomposition. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence-Volume 2*, pages 999–1004. Morgan Kaufmann Publishers Inc.

Muñoz-Avila, H., Jaidee, U., Aha, D. W., and Carter, E. (2010b). Goal-Driven Autonomy with Case-Based Reasoning. In *Case-Based Reasoning. Research and Development*, pages 228–241. Springer.

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

Nau, D., Cao, Y., Lotem, A., and Muñoz-Avila, H. (1999). 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.

Nau, D. S. (2007). Current Trends in Automated Planning. *AI magazine*, 28(4):43.

Nau, D. S., Au, T.-C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., and Yaman, F. (2003). Shop2: An htn planning system. *J. Artif. Intell. Res.(JAIR)*, 20:379–404.

Paisner, M., Maynord, M., Cox, M. T., and Perlis, D. (2013). Goal-driven autonomy in dynamic environments. In *Goal Reasoning: Papers from the ACS Workshop*, page 79.

Paolucci, M., Shehory, O., Sycara, K., Kalp, D., and Pannu, A. (1999). A planning component for retsina agents. In *Intelligent Agents VI. Agent Theories, Architectures, and Languages*, pages 147–161. Springer.

Pettersson, O. (2005). Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53:73–88.

Rao, A. S. and Georgeff, M. P. (1995). BDI Agents: From Theory to Practice.

Reder, L. M. and Ritter, F. (1992). What determines initial feeling of knowing? familiarity with question terms, not with the answer. *Journal of Experimental Psychology: Learning, memory, and cognition*, 18(3):435–451.

Roberts, M., Vattam, S., Alford, R., Auslander, B., Karneeb, J., Molineaux, M., Apker, T., Wilson, M., McMahon, J., and Aha, D. W. (2014). Iterative goal refinement for robotics. In *ICAPS Workshop on Planning and Robotics*.

Schank, R. and Owens, C. (1987). Understanding by explaining expectation failures. In Reilly, R. G., editor, *Communication failure in dialogue and discourse*, pages 201–202. Elsevier Science.

Schank, R. C. (1982). *Dynamic memory: A theory of reminding and learning in computers and people*. Cambridge University Press.

Shivashankar, V., EDU, U., Alford, R., Kuter, U., and Nau, D. (2013). Hierarchical goal networks and goal-driven autonomy: Going where ai planning meets goal reasoning. In *Goal Reasoning: Papers from the ACS Workshop*, page 95.

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., and Katz, Y. (2007). Pellet: A practical OWL-DL reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53.

Smith, D. E. (2004). Choosing objectives in over-subscription planning. In *ICAPS*, volume 4, page 393.

Sugandh, N., Ontañón, S., and Ram, A. (2008). On-Line Case-Based Plan Adaptation for Real-Time Strategy Games. *AAAI*.

Talamadupula, K., Benton, J., Kambhampati, S., Schermerhorn, P., and Scheutz, M. (2010). Planning for human-robot teaming in open worlds. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 1(2):14.

Van Der Krogt, R. and De Weerdt, M. (2005). Plan repair as an extension of planning. In *ICAPS*, volume 5, pages 161–170.

Vattam, S., Klenk, M., Molineaux, M., and Aha, D. (2013). Breadth of approaches to goal reasoning: A research survey.

Veloso, M., Pollack, M., and Cox, M. (1998). Rationale-based monitoring for planning in dynamic environments. *AIPS*.

Wang, X. (1995). Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of the 12th International Conference on Machine Learning*.

Wang, X. (1996). Planning while learning operators. In *Proceedings AIPS*.

Warfield, I., Hogg, C., Lee-Urban, S., and Munoz-Avila, H. (2007). Adaptation of hierarchical task network plans. In *FLAIRS Conference*, pages 429–434.

Weber, B. (2012). *Integrating Learning in a Multi-Scale Agent*. PhD thesis, University of California, Santa Cruz.

Weber, B. G., Mateas, M., and Jhala, A. (2010). Applying Goal-Driven Autonomy to StarCraft. In *AIIDE*.

Weber, B. G., Mateas, M., and Jhala, A. (2012). Learning from demonstration for goal-driven autonomy. In *AAAI*.

Wilson, T. D. and Schooler, J. W. (1991). Thinking too much: Introspection can reduce the quality of preferences and decisions. *Journal of Personality and Social Psychology*, 60(2):181–192.

Yang, Q., Kangheng, W., and Yunfei, J. (2007). Learning action models from plan examples using weighted max-sat. artificial intelligence. *Artificial Intelligence*, 171:107–143.

Vita

Dustin Dannenhauer was born in 1989 in Chicago, IL to Judith and Roger Dannenhauer. He received his Bachelor's of Science in Computer Science from Indiana University Bloomington in May of 2012. He received his Ph.D. in Computer Science from Lehigh University in May of 2017.

Dustin has published the following works:

[AAAI-17] Cox, M. T., Dannenhauer, D., and Kondrakunta, S. "Goal Operations for Cognitive Systems." To appear in: Proceedings of the 31st AAAI Conference on Artificial Intelligence. Palo Alto, CA: AAAI Press. In press.

[IJCAI-16] Dannenhauer, D., Munoz-Avila, H., and Cox, M. T. "Informed Expectations to Guide GDA Agents in Partially Observable Environments." Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI). Palo Alto, CA: AAAI Press. 2016. [PDF]

[IJCAI-16 Goal Reasoning Workshop] Cox, M. T., and Dannenhauer, D. Goal transformation and goal reasoning. M.Roberts (Ed.), Working Notes of the 4th Workshop on Goal Reasoning. New York, IJCAI. 2016. http://makro.ink/ijcai2016grw/

[IJCAI-16 Doctoral Consortium] Dannenhauer, D. "Self Monitoring Goal Driven

Autonomy Agents." Doctoral Consortium, International Joint Conference on Artificial Intelligence (IJCAI). AAAI Press, 2016.

[AAAI-16] Cox, M. T., Alavi, Z., Dannenhauer, D., Eyorokon, V., and Munoz-Avila, H. "MIDCA: A Metacognitive, Integrated Dual-Cycle Architecture for Self-regulated Autonomy." Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI). Palo Alto, CA: AAAI Press. 2016. [PDF]

[IJCAI-15] Dannenhauer, D., and Munoz-Avila, H. "Raising expectations in GDA agents acting in dynamic environments." Proceedings of the 24th International Conference on Artificial Intelligence (IJCAI). AAAI Press, 2015. [PDF]

[ICCBR-15] Dannenhauer, D., and Munoz-Avila, H. "Goal-Driven Autonomy with Semantically-annotated Hierarchical Cases." International Conference on Case-Based Reasoning (ICCBR). Springer International Publishing, 2015. 88-103. [PDF]

[ACS-15 Goal Reasoning Workshop] Munoz-Avila, H., Dannenhauer, D., and Cox, M. T. "Towards cognition-level goal reasoning for playing real-time strategy games." Goal Reasoning: Papers from the ACS Workshop. 2015. [PDF]

[BICA-14] Dannenhauer, D., Cox, M. T., Gupta, S., Paisner, M. and Perlis, D. "Toward meta-level control of autonomous agents." In Proceedings of the 2014 Annual International Conference on Biologically Inspired Cognitive Architectures: Fifth annual meeting of the BICA Society (BICA). Elsevier/Procedia Computer Science, 41, 226-232. 2014. [PDF]

[AIIDE-13 Doctoral Consortium] Dannenhauer, D. "Ontological Knowledge for Goal-

Driven Autonomy Agents in Starcraft." Doctoral Consortium, 9th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE). 2013.

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

[ICCBR-13] Dannenhauer, D., Munoz-Avila, H. "Case-based Goal Selection Inspired by IBM's Watson." International Conference on Case-Based Reasoning (ICCBR). Springer. 29-43. 2013. [PDF]

Dustin has served as a teaching assistant during the Fall 2014 semester at Lehigh University. During his time at Indiana University, he acted as an Undergraduate Instructor for each semester from the Fall of 2009 until the Spring of 2012.