Goal-Driven Autonomy with Semantically-annotated Hierarchical Cases

Dustin Dannenhauer and Héctor Muñoz-Avila

Department of Computer Science and Engineering, Lehigh University, Bethlehem PA 18015, USA

Abstract. We present *LUiGi-H* a goal-driven autonomy (GDA) agent. Like other GDA agents it introspectively reasons about its own expectations to formulate new goals. Unlike other GDA agents, *LUiGi-H* uses cases consisting of hierarchical plans and semantic annotations of the expectations of those plans. Expectations indicate conditions that must be true when parts of the plan are executed. Using an ontology, semantic annotations are defined via inferred facts enabling *LUiGi-H* to reason with GDA elements at different levels of abstraction. We compared *LUiGi-H* against an ablated version, *LUiGi*, that uses non-hierarchal cases. Both agents have access to the same base-level (i.e. non-hierarchical plans), while only *LUiGi-H* makes use of hierarchical plans. In our experiments, *LUiGi-H* outperforms *LUiGi*.

1 Introduction

Goal-driven autonomy (GDA) is a goal reasoning method in which agents introspectively examine the outcomes of their decisions and formulate new goals as-needed. GDA agents reason about their own expectations of actions by comparing the state obtained after executing actions against an expected state. When a discrepancy occurs, GDA agents formulate an explanation for the discrepancy and based on this explanation, new goals are generated for the agent to pursue.

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. [1][2][3]). A common trait of these works is a plain (non-hierarchical) representation for these elements. In this work we propose the use of episodic GDA knowledge in the form of hierarchical plans that reason on stratified expectations and explanations modeled with ontologies. We conjecture that 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.

To test our ideas we implemented a new system, which we refer to as *LUiGi-H* and compared it against a baseline that uses plain GDA representations: *LUiGi*. 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 couldn't. For planning from scratch, HTN planning has been shown to be capable of expressing strategies that cannot be expressed in STRIPS planning [4]. 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 so 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. This will become more clear in Section 5 where we discuss the architecture of both agents. In our results *LUiGi-H* outperforms *LUiGi* demonstrating that it can take advantage of the case-based hierarchical knowledge without incurring periods of inactivity from running time overhead.

2 Example

We present an example in the real-time strategy game Starcraft. In Starcraft, players control armies of units to combat and defeat an opponent. In our example and experiments we concentrate on macro-level decisions; low-level management is performed by the underlying default game controller.

Figure 1 shows a hierarchical plan or h-plan used by *LUiGi-H*. This plan, and every plan in the case base, is composed of the primitive actions found in Table 1 at the lowest level of the h-plan (we refer to the lowest level as the 0-level plan). This h-plan achieves the *Attack Ground Surround task*. For visualization purposes we divide the h-plan into two bubbles A and B. Bubble A achieves the two subtasks *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 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.

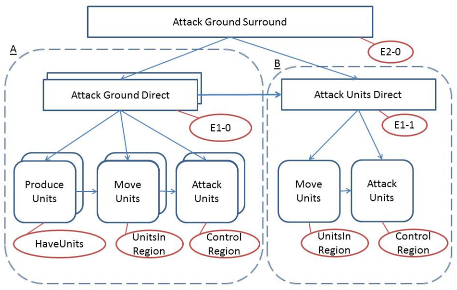

Fig. 1: High Level Plan: AttackGroundSurround

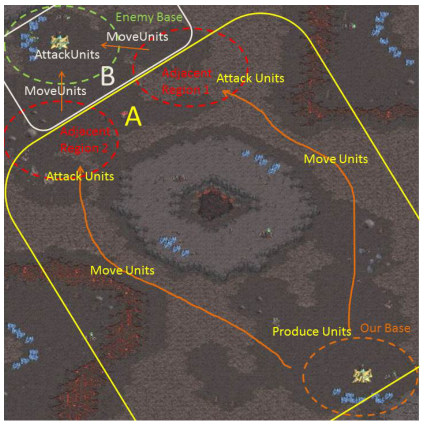

Fig. 2: AttackGroundSurround on map Volcanis

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 of more complex expectations. Specifically, the expectation labeled *E1-0* in Figure 1 represents the condition that all regions that were attacked are under control (Table 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 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 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 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 2: High Level Expectations used in Attack Ground Surround

3 Goal Driven Autonomy

Goal-driven autonomy's goal reasoning mechanism consists of a four step cycle. First a goal is selected by the Goal Manager and sent to planner. While the plan is being executed, the Discrepancy Detector step is checking to see if the plan's expectations are met before and after actions are being executed. If a discrepancy is found, the discrepancy is sent to the Explanation Generator and the system comes up with an explanation, which is then sent to the Goal Formulator to create new goal(s) and finally those goals are sent to the Goal Manager and the cycle is repeated again. These four steps are shown in *LUiGi-H* in Figure 3.

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. In the domain of Starcraft, the state is very large (on the order of thousands of atoms). 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. *LUiGi* uses the ontology to represent the current state of the world as the agent percieves it (i.e. taking into account fog of war: partial observability of the state). This is done to restrict the size of the ontology while still maintaining the ability to infer much needed concept which serve as higher level expectations. The ontology is discussed in more detail in Section 5.3.

The Discrepancy Detector reasons over the ontology comparing inferred facts to the expectations of the current plans' actions to determine if there is a discrepancy between the current state and the expected state. The Explanation Generator provides an explanation for the discrepancy. The Goal Formulator generates a new goal based on the explanation. The Goal Manager manages which goals will be achieved next.

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 hieararchy. The next sections describe details of the inner workings of *LUiGi-H.*

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, [5] uses the actions' semantics of the level-0 plans to check if the plans' expectations are met but does not check the upper layers. Our system *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 an h-plan. We don't assume the general knowledge needed 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 bridge or road 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 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 hiearchy consists exclusevely 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 that 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 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 coumpound 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 in contrast to systems like HTNbots-GDA 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 later. It will start the second *Produce Units* action 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.

5 A Hierarchical Case-based GDA System

Our *LUiGi-H* system combines CBR episodic and hierarchical task network (HTN) representation to attain a GDA system that reasons with expectations, discrepancies and explanations at varied level of abstraction.

Figure 3 shows an overview of *LUiGi-H*. It consists of two main components: the Controller and the Bot. The Controller is the main component of the system and it is responsible for performing the GDA cycle (shown under the box "GDA Cycle"), planning, and reasoning with the ontology.

The Bot is in charge of executing and monitoring the agent's actions. In our experiments *LUiGi-H* plays Starcraft games. Communication between the Controller and the Bot are made with TCP/IP sockets and file share systems.

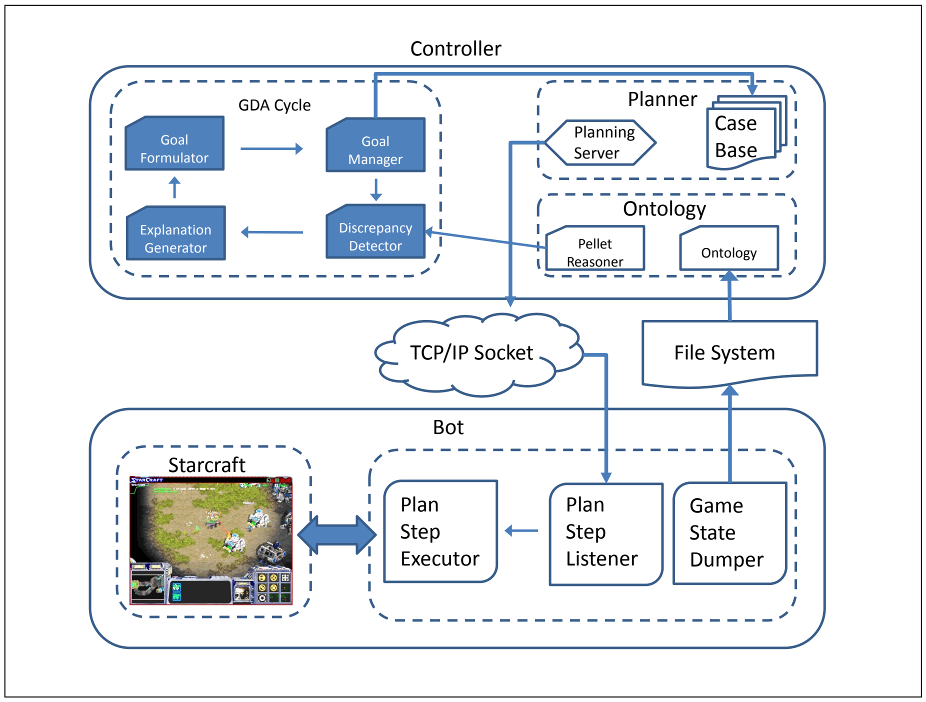

Fig. 3: *LUiGi-H* Overview

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

Here we give a brief overview of each component before going into more detail for the Planner in Section 5.2 and the Ontology in Section 5.3.

The Planner. As explained in Section 4, 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.

5.2 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, 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 to harvest the necessary resources, build required buildings, and issue commands 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 units to any region.

5.3 Ontology

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 uses facts as its representation of atoms. Facts are ⟨subject, predicate, object⟩ triples. 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⟩). We use an ontology to represent high-level concepts such as controlling a region. By

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 3: Plans

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 the fog of war. That is, a player can vie only the portion of the map where it has units deployed. As a result, 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.
- **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.

5.4 Discussion

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 controller then uses this data to populate a semantic web ontology, in which to reason about the game to infer more abstract conclusions (these are used in discrepancy detection). 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).

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 a 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.

6 Empirical Evaluation

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 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. 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[1] is assigned 150 points whereas a factory[2] 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.
- **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*.

[1] 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.

[2] 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.

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 4 below.[3]

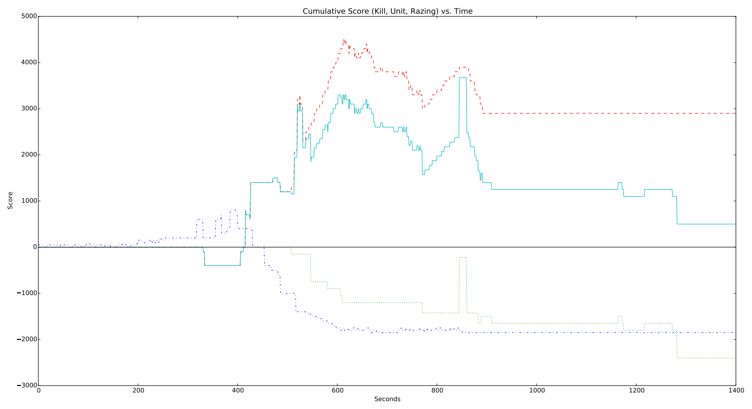

Fig. 4: *LUiGi-H* vs. *LUiGi* on Heartbreak Ridge

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

From Figure 4 we see that *LUiGi-H* ended with a higher total score than *LUiGi*, starting around the 400 second mark. In Figure 4, 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 5, on the map Challenger.

Figure 5 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

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

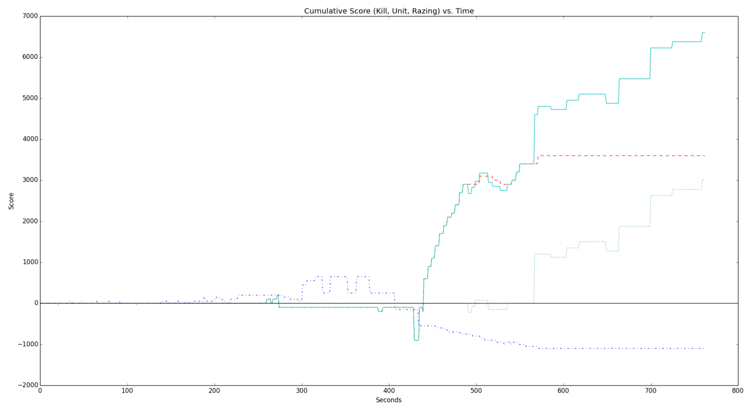

Fig. 5: *LUiGi-H* vs. *LUiGi* on Challenger

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.

7 Related Work

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 *LUiGi* [1], GRL [2] and EISBot [3]. As with all GDA systems, their main motivation is for the agents to autonomously react to unexpected situations in the environment. From these, the most closely related is *LUiGi* as it uses ontologies to infer the expectations. However, none of these GDA systems, including *LUiGi*, uses h-plan representations.

The most closely related works are the one from [5] and [6–8], which describe the HTNbots and the ARTUE system respectively. Both HTNBots and ARTUE uses the HTN planner SHOP [9]. 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 [10] Caplan/CbC system [11], Process manufacturing case-based HTN planners [12] and the SiN system [13]. 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.

8 Conclusion

In this paper, we presented *LUiGi-H*, a GDA agent that combine CBR episodic knowledge, h-plan knowledge and ontological information enabling it to reason about the plans, expectations, discrepancies, explanations and new goals at different levels of abstraction.

We compared *LUiGi-H* against an ablated version, *LUiGi*. 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.

For future work, we will explore using case-based learning techniques to acquire the h-plans automatically from previous problem-solving experiences. Specifically, we envision a situation in which *LUiGi-H* starts with no h-plans and learns these plans from multiple starcraft matches against different opponents. This will in turn allows us to test *LUiGi-H* versus the highly optimized (and hard-coded) entries in the Starcraft competition.

References

1. D. Dannenhauer and H. Muñoz-Avila. LUIGi: A Goal-Driven Autonomy Agent Reasoning with Ontologies. In *Advances in Cognitive Systems (ACS-13)*, 2013.
2. Ulit Jaidee and Héctor Muñoz-Avila. Modeling Unit Classes as Agents in Real-Time Strategy Games. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.
3. Ben Weber. *Integrating Learning in a Multi-Scale Agent.* PhD thesis, University of California, Santa Cruz, June 2012.
4. Kutluhan Erol, James Hendler, and Dana S Nau. Htn planning: Complexity and expressivity. In *AAAI*, volume 94, pages 1123–1128, 1994.
5. Héctor Muñoz-Avila, David W Aha, Ulit Jaidee, Matthew Klenk, and Matthew Molineaux. Applying Goal Driven Autonomy to a Team Shooter Game. In *FLAIRS Conference*, 2010.
6. Matthew Molineaux, Matthew Klenk, and David W Aha. Goal-Driven Autonomy in a Navy Strategy Simulation. In *AAAI*, 2010.
7. M. Molineaux and D.W. Aha. Learning Models for Predicting Surprising Events. In *Advances in Cognitive Systems Workshop on Goal Reasoning*, 2013.
8. Vikas Shivashankar, UMD EDU, Ron Alford, Ugur Kuter, and Dana Nau. Hierarchical goal networks and goal-driven autonomy: Going where ai planning meets goal reasoning. In *Goal Reasoning: Papers from the ACS Workshop*, page 95, 2013.
9. Dana Nau, Yue Cao, Amnon Lotem, and Héctor Muñoz-Avila. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, pages 968–973. Morgan Kaufmann Publishers Inc., 1999.
10. Subbarao Kambhampati and James A Hendler. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence*, 55(2):193–258, 1992.
11. Héctor Muñoz, Jürgen Paulokat, and Stefan Wess. *Controlling a nonlinear hierarchical planner using case replay.* Springer, 1995.
12. H-C Chang, Lijun Dong, FX Liu, and Wen F Lu. Indexing and retrieval in machining process planning using case-based reasoning. *Artificial Intelligence in Engineering*, 14(1):1–13, 2000.
13. Héctor Muñoz-Avila, David W Aha, Dana S Nau, Rosina Weber, Len Breslow, and Fusun Yaman. Sin: Integrating case-based reasoning with task decomposition. Technical report, DTIC Document, 2001.