ABSTRACT

Developing agents capable of defeating competitive human players in real-time strategy games remains an open research problem. My dissertation work aims to achieve this goal and in undertaking this process, address several interesting AI research challenges. Notably, building systems capable of intelligently interacting with human participants in complex, real-time, partially observable domains in which actions must be concurrently executed at multiple scales. My approach to addressing these challenges is to develop an integrated agent architecture that implements the goal-driven autonomy conceptual model. The proposed architecture will enable the system to intelligently react to opportunities as well as unanticipated situations caused by opponents and exogenous events. The agent will perform multi-scale reasoning by concurrently pursuing multiple goals based on a conceptual partitioning of the domain.

The goal of achieving human-level performance in this domain will be accomplished by integrating several forms of domain knowledge including hand-authored behavior, expert demonstrations, and player created repositories of strategies and tactics. To employ these disparate sources of knowledge, the agent will utilize an ensemble of symbolic and statistical AI techniques. These techniques will be integrated by defining ontologies for communication between components in the goal-driven autonomy conceptual model. The claim of building a human-level AI for real-time strategy games will be evaluated by testing the system against competitive human opponents in StarCraft.
CONTENTS

1 INTRODUCTION 1
  1.1 Research Question and Contributions 2
  1.2 Proposal Overview 3

2 RELATED WORK 4
  2.1 Commercial Game AI 4
  2.2 Case-Based Reasoning 5
    2.2.1 Case-Based Reasoning in RTS Games 5
  2.3 Agent Architectures 7
    2.3.1 Reactive Planning 7
    2.3.2 Cognitive Architectures 7
    2.3.3 Goal-Driven Autonomy 8
  2.4 Opponent Modeling 8
    2.4.1 Game State Estimation 8
    2.4.2 Plan Recognition 9

3 STARCraft 10
  3.1 Gameplay 11
  3.2 Domain Properties 12
  3.3 Decision Complexity 13

4 PRIOR WORK 14
  4.1 ABL Wargus 14
  4.2 Multi-Scale Game AI 15
  4.3 Goal-Driven Autonomy 16
  4.4 Learning from Demonstration 17
  4.5 Discussion 18

5 PROPOSED WORK 19
  5.1 Architecture 19
  5.2 Competencies 21
  5.3 Example Scenarios 21
  5.4 Components 23
    5.4.1 Event Recognizer 23
    5.4.2 Explanation Generator 25
    5.4.3 Opponent Model 27
    5.4.4 Goal Formulator 27
    5.4.5 Goal Manager 29
  5.5 Example Execution 29
  5.6 Evaluation 31

6 SCHEDULE 32

7 CONCLUSION 33

BIBLIOGRAPHY 34
Artificial Intelligence (AI) has a long history of using games as a testbed for advancing the state of the field [69]. Games are an excellent environment for AI research, because they provide problems that can lead to incremental and integrative advances needed to achieve human-level AI [39]. While human-level performance has been achieved for traditional board games, such as chess, it has not yet been achieved for modern video games, such as real-time strategy (RTS) games. While significant effort has been applied to building human-level AI for RTS games [50, 57, 84], the complexity of the domain has resulted in research providing only marginal improvements. RTS games are an interesting domain for AI research, because they provide an environment for evaluating techniques which perform intelligent real-time decision making, while bypassing many of the problems encountered when interacting with the real world [38].

The domain of RTS games provides several research challenges that need to be addressed in order to build real-world AI systems. Specifically, building agents capable of performing decision making in real-time, partially observable domains that contain enormous decision complexities [2]. RTS games are multi-scale and require concurrently reasoning about goals at multiple levels of granularity [80]. An initial call for research in RTS games was proposed by Michael Buro at IJCAI in 2003 [10]. He claimed that research in this area would lead to improvements in adversarial planning under uncertainty, learning and opponent modeling, and spatial and temporal reasoning.

Building human-level AI for RTS games remains an open research problem. There are two main reasons why AI research has not yet overcome this challenge. First, RTS games contain large strategy spaces, which support a variety of gameplay styles. Hand-authoring an agent that can counter every possible strategy becomes an authorial burden. Second, even if it is feasible to hand-author all possible strategies, players will innovate new strategies that were not anticipated during the development of the game. RTS games have an evolving meta-game in which new strategies are constantly being created. Therefore, achieving human-level performance in RTS games requires not only a large library of strategies and knowledge about how to counter known strategies, it requires the ability to learn new strategies and counters.

RTS games provide several sources of knowledge that can be used to build agents that learn. One source an agent can use is its previous gameplay experiences [2]. However, this approach used in isolation is unlikely to result in significant improvements in gameplay, due to the enormous state space of RTS games. Another source of knowledge is expert demonstrations in the form of replays [83]. The main challenge in learning from expert replays is learning from the limited amount of information contained in replays, which are sequences of noisy game actions. A third source of knowledge is user created repositories of strategies and tactics [73]. However, this type of information is mostly unstructured and learning strategies automatically from repositories may require natural language understanding.

While a large amount of domain knowledge can be learned from these sources of knowledge, there is a limit to how much can be learned automatically. In RTS games, there are maneuvers that require such fine control that it is currently intractable to learn them automatically. To overcome this limitation, these types of behaviors should be authored by a
human expert. Achieving human-level performance in RTS games requires both performing precise maneuvers as well as tracking the evolving meta-game. Therefore, my dissertation work aims to build human-level AI that integrates expert knowledge, in the form of human-authored behaviors, with experience, in the form of replays and repositories. Achieving this goal will require utilizing an ensemble of symbolic and statistical AI techniques.

1.1 RESEARCH QUESTION AND CONTRIBUTIONS

My dissertation work aims to advance the state of game AI, focusing on the RTS game StarCraft. Achieving this goal will require building a system that can reason about long-term goals while concurrently reacting to unanticipated second-by-second state changes. Additionally, the system will need to learn new strategies and tactics from several disparate knowledge sources. This work will address the following research question:

What techniques are required to develop adaptive agents capable of intelligently reacting to unforeseen situations and exhibiting human-level performance in complex domains?

The main contribution of this work is techniques for expanding reactive planning with experience learned from expert demonstrations. The main challenge in accomplishing this goal is integrating symbolic planning with learning algorithms. This will be achieved by building an integrated agent architecture that implements the goal-driven autonomy (GDA) conceptual model [51]. My work will expand the integrated agent framework of McCoy and Mateas [47] by fully implementing the proposed managers using reactive planning, machine learning, and case-based reasoning components. This framework will be realized using the ABL reactive planning language, which enables reactive, parallel goal pursuit with long-term planfulness [45]. The framework will also incorporate a mental model that predicts opponent actions, enabling the system to operate in an imperfect information environment.

The system will learn additional domain knowledge from expert demonstrations using both knowledge-weak and knowledge-rich techniques. Knowledge-weak techniques will be used to predict an opponent’s strategy and future state, while knowledge-rich techniques will be used to explain the game state and formulate goals. This work will also assess the limit of how much can be learned automatically from demonstrations.

The agent will implement the GDA conceptual model, which was proposed as a methodology for building agents capable of intelligently responding to planning opportunities and failures in a domain [51]. This model will be used to identify important events that will trigger the agent to assess the current game state as well as its current goals. A contribution of my work is demonstration of a system that applies the GDA conceptual model to concurrent goal pursuit. This will be achieved by using ABL as the goal management component, which provides native support for parallel goal pursuit.

The system will perform multi-scale reasoning by conceptually partitioning the domain of StarCraft into distinct areas of competence. This partitioning will enable the agent to reason about separate aspects of the game while minimizing goal conflicts. To address the issue of goal conflicts, the system will utilize reactive planning idioms [80] for building multi-scale game AI.

The result of my work will be an agent that integrates symbolic and statistical techniques in a game playing agent. The expected outcomes are techniques for reducing the authorial burden of building agents for complex domains and building adaptive agents that learn
from experience. Another contribution of my work is a system that demonstrates human-level performance against competitive players in StarCraft. This work is significant to the AI community in general, because techniques used to achieve human-level intelligence in RTS games may be applicable to other domains that have similar properties. The claim of achieving human-level intelligence will be validated by testing the agent against competitive human players.

1.2 PROPOSAL OVERVIEW

The remainder of this thesis proposal is structured as follows: Section 2 provides a discussion of related work with a focus on the application to RTS games, Section 3 contains an overview of StarCraft and its potential as an AI testbed, Section 4 describes my previous work building towards this proposal, Section 5 specifies my proposed research and agent architecture, Section 6 provides a schedule outlining a timeline for my research plan, and Section 7 provides a summary of this proposal.
My research project draws ideas from a variety of areas including case-based reasoning, planning, and machine learning. This section provides an overview of related work in these areas, focusing on the application to RTS games.

2.1 COMMERCIAL GAME AI

One of the goals of my project is to build a system capable of being deployed in a commercial game environment. The main challenge in achieving this goal is dealing with the severe computational constraints placed on game AI [10, 35]. While game platforms are increasingly becoming more powerful, enabling additional computational resources to be allocated to AI, it is first necessary to demonstrate that my approach not only outperforms current game AI practices, but that it is robust as well.

Finite state machines (FSMs) are one of the most commonly used techniques for building game AI [65]. They are popular due to their efficiency, simplicity and expressivity [25]. FSMs define a set of states and allowed transitions between states. Transitions contain a set of conditions, which specify when it is valid to switch between states. While using FSMs is straightforward for simple domains, the approach does not scale well, because the number of transitions increases polynomially as the number of states increases. One way of overcoming this authorial burden is to group states together into super-states [25], which results in hierarchical FSMs. While hierarchically structuring FSMs helps humans partition complex behavior into domains of competence, in practice it does not reduce the authoring burden [35]. One of the major limitations of FSMs is that logic encoded in this representation cannot be reused across different contexts [59] (e.g. different unit types).

Another approach to building game AI is subsumption architectures, which are a layered approach. In subsumption architectures, lower levels take care of immediate goals and higher levels manage long-term goals [86]. The levels are unified by a common set of inputs and outputs, and each level acts as a function between them, overriding higher levels. While subsumption architectures enable reasoning at multiple levels of granularity, only one layer can be active at a time. Therefore, agents utilizing this technique can have at most a single active goal.

Behavior trees have recently become an adopted technique for building commercial game AI [13, 35]. In a behavior tree, an agent’s behavior is defined by a hierarchically structured, prioritized lists of behaviors. Behaviors in this representation contain a set of activation conditions, which specify when the behavior is valid given the world state. Non-leaf nodes contain a set of activation conditions, but are not associated with a set of actions to execute. Each decision cycle, a behavior is selected by re-evaluating the tree from the root node and expanding the tree until a leaf node is reached. Each node in the tree evaluates which child nodes are valid based on the activation conditions, and expands the node with the highest priority. Behavior trees have been used separately for both individual unit control and squad control [36]. The main drawback of behavior trees is that a substantial amount of domain engineering is required to build an agent capable of anticipating all game events [54].
Goal-oriented action planning (GOAP) is a technique for building game AI using planning [58]. GOAP architectures utilize a STRIPS-style representation [24] and build plans by performing heuristic search through a set of operators. In order to operate in a real-time environment, the planner uses efficient data structures and heuristics for guiding the planning process [29]. GOAP was first implemented in the FPS game F.E.A.R. [59] and has recently been applied to building game AI for additional genres\(^1\). The RTS game Empire: Total War\(^2\) uses GOAP to generate plans for high-level strategies, while employing FSMs for low-level behaviors.

### 2.2 Case-based Reasoning

Case-based reasoning (CBR) is a methodology for building systems that learn from experience [1]. In CBR systems, knowledge learned from previously solved problems is stored in instances known as cases, which are utilized when faced with a new problem situation. CBR is a cyclic process of problem solving, where experience learned from solving new problems results in cases that are added to the knowledge base. When presented with a new problem situation, CBR first encodes the problem as a case. Next, the most similar case is retrieved from a library of cases. Third, the information from the retrieved case is applied to solving the current solution. Next, the case is revised based on the outcome of the application of the case to the problem. Finally, the revised case is retained in the case library for future use. The CBR process enables systems to learn new knowledge online, by constantly revising their knowledge base.

CBR is integral to building my proposed system, because it provides techniques for automatically acquiring domain knowledge and learning online. This section provides an overview of CBR research applied to RTS games. For a broader discussion of CBR research, refer to the CBR wiki\(^3\).

#### 2.2.1 Case-Based Reasoning in RTS Games

Case-based reasoning has been applied to solving several sub-problems in RTS games including micromanagement [72], tactics [52], and strategy selection [2]. There are also CBR systems aimed at building complete game playing agents [57]. While some of the approaches discussed here could be considered strictly instance-based as opposed to case-based (cases are expected to contain a degree of richness in their representation [1]), all of them learn from experience and build libraries of cases.

Aha et al. were the first to apply CBR to building game AI for RTS games\(^2\). Their system used CBR to perform strategy selection in Wargus. It makes use of three sources of domain knowledge: a state lattice, a set of strategies for each node in the lattice, and cases, which map game situations to specific nodes in the lattice and contain a performance attribute. Using this technique requires human authoring of the strategies the agent can pursue and the lattice which classifies the game state based on high-level strategies. The state lattice contains 20 nodes, each of which are associated with one or more of the seven identified strategies [61]. The system learns cases online by exploring states in the lattice. After execution, a case is assigned a performance metric based on changes in the agent’s score, which is a function of the number of units and buildings constructed and destroyed.

---

1 Jeff Orkin maintains a list of games using GOAP: [http://web.media.mit.edu/~jorkin/goap.html](http://web.media.mit.edu/~jorkin/goap.html)
2 [http://worthplaying.com/article/2008/12/1/interviews/57018/](http://worthplaying.com/article/2008/12/1/interviews/57018/)
3 [http://cbrwiki.fdi.ucm.es/wiki](http://cbrwiki.fdi.ucm.es/wiki)
2.2 CASE-BASED REASONING

The system was shown to improve in performance over time when applied to the task of playing against a pool of fixed strategies. Later work performed cross-validation of the technique, in which the strategy tested against was not used during the training phase [50].

CBR has also been applied to the task of micromanagement in RTS games. Szczepański and Aamodt used CBR to improve the micromanagement of units in WarCraft III [72]. Their case representation contains actions and behaviors, where actions are primitive game actions that are adapted to the current game situation and behaviors are hand-authored rules, such as retreating when a unit has low health. Case retrieval occurs once per second and behaviors are used to achieve split-second reactions. One of the major limitations of their representation is a lack of unit history, which often results in a unit dithering between actions, such as retreating too frequently. Despite this limitation, their approach was shown to significantly increase the utility of units in micromanagement tasks.

Darmok is a case-based planner that learns to play complete games of Wargus from expert demonstrations [57]. Case-based planning is a technique that builds plans in plan-space rather than state-space by applying CBR to the plan construction process as opposed to search [20]. Darmok extends traditional case-base planning by operating online, interleaving planning and execution. Darmok’s case representation includes an encoding of the game state, a goal, and a set of actions. The goal attribute describes the goal that the case accomplishes in a human-authored goal hierarchy. The actions in a case specify primitive actions to perform or additional sub-goals for the planner to pursue. The major limitation of Darmok is that generating the case library requires manual annotation of demonstrations, where each action taken by the player is annotated with one or more goals.

Darmok 2 is the successor to the original Darmok system and eliminates the need for players to manually annotate demonstrations [56]. The system uses a technique based on HTN-maker [30] in order to automatically learn cases from expert demonstrations. One of the limitations of this approach is that Darmok 2 relies on the assumption that for each action, the conditions that become true as an effect of the executed action were intended by the expert. The system can learn spurious plans for goals that were achieved only accidentally [77]. Darmok 2 is used to power MakeMEPlayMe4, a website where players build bots by playing games.

Bakkes et al. present a CBR approach for improving a pre-existing game AI for the RTS game Spring [5]. The purpose of modifying an existing AI was to demonstrate that learning could be incorporated in a robust AI. The system adds adaptation to the game AI by integrating a CBR component that selects parameter values used by the AI. Their case representation contains a set of features describing game state and a label, consisting of 27 parameter values. Their results showed that adapting the parameters of the AI improved the performance of the system while maintaining robustness.

Hybrid CBR approaches have also been applied to RTS games. Sharma et al. applied hybrid CBR/reinforcement learning to MadRTS [70]. They demonstrated that their approach enables transfer learning of tactical tasks in RTS games, where knowledge learned from a specific tactical situation can be applied in different tactical situations. Their system uses CBR as a function approximator for the reinforcement learning component. More recent work demonstrated the use of hybrid CBR/reinforcement learning in continuous action spaces [52].

Baumgarten et al. present a hybrid CBR system that integrates simulated annealing, decision tree learning and case-based reasoning [7]. Their system is evaluated in the tactics-based RTS game DEFCON: Everybody Dies. It uses an initial case library generated from a

4 http://makemeplayme.com/
set of randomly played games and refines the library by applying an evaluation function to retrieved cases. Cases are retrieved at the beginning of a game to build a game plan and during the game to predict opponent movements.

2.3 AGENT ARCHITECTURES

One of the contributions of my proposed work is an architecture for building multi-scale agents that operate in real-time environments. This section provides an overview of agent architectures that have been applied to building game AI.

2.3.1 Reactive Planning

Reactive planning has been applied to creating autonomous characters for virtual environments [42, 44]. In a reactive planning system, no sequence of actions is planned in advance, rather a new action is selected at every instant. A system utilizing reactive planning can operate in an environment with uncertain effects and handle exogenous events [64]. Reactive planning is well suited for real-time environments, because the mechanism for behavior selection is strictly bounded and efficient [42].

A Behavior Language (ABL) is a reactive planning language developed to support the creation of the believable characters in the interactive drama Façade [46]. ABL adds significant features to the original Hap [42] semantics, including first-class support for meta-behaviors and joint intentions across teams of multiple agents [45]. ABL is well suited for developing autonomous agents, because it was developed to achieve the following goals: reactivity, responsiveness, deliberation, and explicit goals [44]. While ABL was designed with the intention of supporting the creation of autonomous characters, it is applicable to other domains that require combining reactive, parallel goal pursuit with long-term planfulness [45]. For example, ABL has been applied to the task of playing complete games of Wargus [47]. Recent work has explored the use of reinforcement learning to enable the partial specification of ABL agents [71].

2.3.2 Cognitive Architectures

Cognitive architectures work towards developing mechanisms that underlie human cognition [41]. ICARUS is a cognitive architecture for physical agents that incorporates concepts from cognitive psychology [40]. One of the goals of ICARUS is to build an architecture capable of concurrently reasoning about multiple goals, which can be interrupted and resumed. ICARUS uses means-ends analysis when confronted with a new problem situation. The ICARUS cognitive architecture was applied to the FPS game Urban Combat, a stand-alone modification of Quake 3 Arena [17]. ICARUS differs from ABL in that ABL does not attempt to model cognitive processes and ABL does not utilize means-ends analysis when there is no behavior for achieving a goal.

SOAR is a cognitive architecture that uses production rules to select behaviors to execute [41]. It uses a learning technique called chunking, where new production rules are learned by the system based on the courses of action taken. SOAR has been applied to playing complete games of ORTS [84]. The agent includes algorithms based on human perception to form groups, which are managed by a global coordinator. Micromanagement is handled in SOAR’s middleware, which employs FSMs. The main difference between my proposed
architecture and SOAR is additional coordination between units, which is accomplished through the application of reactive planning idioms [80].

2.3.3 Goal-Driven Autonomy

Goal-driven autonomy (GDA) is a research topic in AI that strives to develop agents capable of intelligently reacting to planning opportunities and failures in a domain [26]. It is motivated by Cox's claim that an agent should reason about itself as well as the world around it in a meaningful way in order to continuously operate with independence [19]. The goal-driven autonomy conceptual model [51, 54] is not an agent architecture, rather it provides a framework for building agents capable of responding to failures in a domain. The model contains several components and establishes interfaces between them, but makes no commitment to specific implementations for any of the components. Current implementations of components in the model include production rules [54] and case-based reasoning [53].

The GDA model has been used to build a team coordinator for FPS bots in a team domination game [54]. The system is responsible for selecting which domination locations to control, while individual unit behavior is managed using HTN planning [29]. The agent performs goal formulation using a set of if-then rules and requires a large amount of hand-coded background knowledge. Currently, the system executes at most a single active goal.

2.4 OPPONENT MODELING

My system will be evaluated against highly-competitive players. Performing well in this environment requires building predictions of the opponent's possible actions. The agent will need to exhibit foresight in order to execute effective counter-strategies. Additionally, the system will operate in an imperfect information environment. Therefore, the agent needs to build estimations of the current game state. These issues will be addressed through the application of opponent modeling techniques.

Van den Herik et al. [76] identify a variety of techniques used for opponent modeling in commercial games: evaluation functions [4], machine-learned function approximators [83], probabilistic models [14], and case-based models [78].

2.4.1 Game State Estimation

Particle filters provide a technique for performing current and future game state estimation. In a traditional particle filter, the location of a unit is modeled via a cloud of moving particles, each of which perform a random walk [74]. Particle filters have been applied to predicting opponent locations in a FPS game [8]. When an opponent leaves the agent’s field of vision, a cloud of particles is spawned at the last known position of the enemy unit. Once a cloud has been established, further observations modify the distribution of particles. The accuracy of particle filters can be improved by using simulacra, which imitate the behavior of the agents they represent, rather than performing a random walk [21]. Darken and Anderegg applied simulacra to opponent tracking in FPS games.

Another approach to game state estimation is evaluation functions. Bakkes et al. applied an evaluation function to predict the outcome of games in Spring [4]. TD-learning is used to
determine parameters for the evaluation function. To handle imperfect information, they assume a uniform distribution of units placed throughout the environment.

2.4.2 Plan Recognition

Plan recognition is the task of inferring an agent’s plans or goals based on observed actions [9]. Techniques for performing plan recognition in games include Bayesian models [3], case-based reasoning [23], and machine learning [68].

Albrecht et al. applied dynamic belief networks to predicting the player’s current goal in a dungeon adventure game [3]. Their approach is based on Bayesian models [14], which apply a probabilistic approach to plan recognition. Explanations for the player’s behavior are assembled into a Bayesian network, which is a probability distribution over the set of possible explanations. Their representation enables the use of incomplete and noisy data during both training and testing, while supporting a stateful model. Their results suggest that dynamic belief networks offer a promising approach to plan recognition in situations where the causal structure of the network can be clearly identified [12].

Case-based plan recognition offers an instance-based approach. It has been applied to performing player modeling in Space Invaders [23]. Case-based plan recognition is an experience-based approach to plan recognition, where case libraries are constructed by observing game play. Each observed sequence of actions is assigned a support count, which is used to identify common strategies. Action sequences with a high support count are marked as plans, added to the case library, and used to predict a player’s future actions. Cheng and Thawonmas discuss the application of case-based plan recognition to RTS games [16].

Machine learning has also been applied to plan recognition. Hsieh and Sun built a player model by analyzing StarCraft replays [31]. Their approach uses a state lattice to represent the possible strategies and their model predicts the next strategic action a player will execute. The model is trained on a single player’s replays and therefore represents a player model for a specific player. Schadd et al. present a two level classifier for predicting a player’s strategy [68]. The top level classifies the player’s style, while the bottom level classifies specific unit types. Their approach requires hand authoring rules for labeling strategies at each of the levels.
My proposed work is targeted at building an agent capable of human-level StarCraft gameplay. *StarCraft* and its expansion, *StarCraft: Brood War* were released by Blizzard Entertainment™ in 1998. In StarCraft, players take the role of a commander in a complex strategy simulation in which three diverse races compete for galactic domination: Protoss, Terran, and Zerg. A screen capture of StarCraft is shown in Figure 1. While several other RTS frameworks are available for evaluating AI research including Stratagus [63], ORTS [11], LagoonCraft [49] and Spring¹, there are several characteristics of StarCraft that make it a strong candidate for AI research.

![Figure 1: A screen capture of StarCraft showing a Terran (Yellow) attacking a Protoss opponent](image)

StarCraft is the successor to Blizzard’s RTS game *WarCraft II*. Ponsen et al. re-implemented Warcraft II us using the open-source Stratagus engine [63], providing an environment for RTS research. A large amount of research on RTS games has used Wargus as an AI testbed [2, 43, 47, 48, 55, 62, 6, 82]. Prior research on Wargus should be applicable to StarCraft, because there are many similarities between the games. However, the decision complexity of StarCraft is much greater due to a larger tech tree, number of units and spells, and number of tactics. Therefore, techniques that do not scale well in Wargus will not scale well

¹ [http://springrts.com/](http://springrts.com/)
in StarCraft. StarCraft offers a more interesting environment for AI research than Wargus, and several universities are now utilizing StarCraft as an AI testbed\(^2\).

There is a large community of players supporting StarCraft, which ranges from amateurs to professionals. StarCraft is played all over the world, which ensures that there is a large, active player base for evaluating research against human players that have a strong understanding of the game. Agents that play StarCraft can be tested on ladder servers against human players that provide metrics, similar to the Elo rating in chess, which can be used to describe the performance of an agent. Additionally, StarCraft is played at a professional level in South Korea\(^3\). Professional gameplay requires performing over 300 actions per minute (APM) during peak gameplay and has been compared to grandmaster play in chess \([47]\).

The professional leagues create an environment in which the meta-game of StarCraft is constantly evolving. New strategies are constantly being developed to counter the currently most popular strategies, which ensures that no strategy becomes dominant. Professional gaming has played an important role in maintaining the popularity of StarCraft for over a decade after its release \([15]\).

Due to the popularity of StarCraft, there is a large amount of data that can be used to automate the process of learning domain knowledge. This knowledge is available in a variety of forms, which range in richness of information. StarCraft provides the ability to save games in the form of replays, which contain the user interface actions performed by each player during the game. While replays are easy to parse, the information that can be extracted is limited to primitive game actions \([78]\). Commentaries are another form of information for describing RTS gameplay. In a StarCraft commentary, a commentator provides insight describing a player’s intentions and goals, similar to commentary provided for sporting events such as football. However, extracting information from commentaries requires natural language understanding. Another source of domain knowledge is user-created repositories of information, such as Liquipedia \([73]\). It is less difficult to extract domain knowledge from repositories than commentaries, because the information is semi-structured. The tradeoff between richness of information and machine readability of these different knowledge sources is shown in Figure 2.

![Figure 2: Sources of domain knowledge for StarCraft](image)

3.1 GAMEPLAY

StarCraft is a science fiction RTS game in which players manage an economy, produce units and buildings, and vie for control of the map with the goal of destroying all opponents. It

\(^2\) Personal correspondence with Ashwin Ram and Alan Fern

\(^3\) Korea e-Sports Association: [http://www.e-sports.or.kr/](http://www.e-sports.or.kr/)
requires multi-scale reasoning, because it involves low-level tactical decisions that must complement high-level strategic reasoning.

At the strategic level, StarCraft requires decision-making about long-term resource and technology management. For example, if the agent is able to control a large portion of the map, it gains access to more resources, which is useful in the long term. However, to gain map control, the agent must have a strong combat force, which requires more immediate spending on military units, and thus less spending on economic units in the short term.

At the economic level, the agent must also consider how much to invest in various technologies. For example, to defeat cloaked units, advanced detection is required. But the resources invested in developing detection are wasted if the opponent does not develop cloaking technology in the first place.

At the tactical level, effective StarCraft gameplay requires both micromanagement of individual units in small-scale combat scenarios and squad-based tactics such as formations. In micromanagement scenarios, units are controlled individually to maximize their utility in combat. For example, a common technique is to harass an opponent’s melee units with fast ranged units that can outrun the opponent. In these scenarios, the main goal of a unit is self-preservation, which requires a quick reaction time.

Effective tactical gameplay also requires well coordinated group attacks and formations. For example, in some situations, cheap units should be positioned surrounding long-ranged and more expensive units to maximize the effectiveness of an army. One of the challenges in implementing formations in an agent is that the same units used in micromanagement tactics may be reused in squad-based attacks. In these different situations, a single unit has different goals: self-preservation in the micromanagement situation and a higher-level strategic goal in the squad situation.

### 3.2 Domain Properties

StarCraft is a complex domain with many real-world properties. An overview of StarCraft in terms of Russell and Norvig’s task environment properties [67] is shown in Table 1. The only difference between StarCraft and a real-world task, such as taxi driving, is that StarCraft is a deterministic simulation. That is, StarCraft provides a complex simulation environment for evaluating AI research [38, 39].

<table>
<thead>
<tr>
<th>Fully vs. Partially Observable</th>
<th>Chess</th>
<th>StarCraft</th>
<th>Taxi Driving</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic vs. Stochastic</td>
<td>Deterministic</td>
<td>Deterministic</td>
<td>Stochastic</td>
</tr>
<tr>
<td>(Strategic)</td>
<td>(Strategic)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Episodic vs. Sequential</td>
<td>Sequential</td>
<td>Sequential</td>
<td>Sequential</td>
</tr>
<tr>
<td>Static vs. Dynamic</td>
<td>Static</td>
<td>Dynamic</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Discrete vs. Continuous</td>
<td>Discrete</td>
<td>Continuous</td>
<td>Continuous</td>
</tr>
<tr>
<td>Single vs. Multiagent</td>
<td>Multiagent</td>
<td>Multiagent</td>
<td>Multiagent</td>
</tr>
<tr>
<td>(Competitive)</td>
<td>(Competitive)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I am defining StarCraft as a continuous domain, even though the state space is finite, due to the ability to perform actions in continuous time.
3.3 Decision Complexity

A formal analysis of the decision complexity of RTS games was first proposed by Aha et al. [2]. They estimated the decision space of Wargus, which is the set of possible actions that can be executed at a particular moment, as follows:

$$O(2^W (A \cdot P) + 2^T (D + S) + B(R + C))$$

W - number of workers  
A - number of the type of worker assignments  
P - average number of workplaces  
T - number of troops  
D - number of movement directions  
S - number of troop stances (Attack, Move, Hold)  
B - number of buildings  
R - average number of research options at buildings  
C - average number of unit types at buildings

This estimation transfers well to StarCraft, but most parameter values will be larger for StarCraft than Wargus. Assuming that the decision to perform for each unit can be selected independently, the decision complexity can be expressed as follows:

$$O((W \cdot A \cdot P) + (T \cdot D \cdot S) + (B \cdot (R + C)))$$

Given this formulation, the decision complexity is still large. For example, a 256x256 tile map results in thousands of possible workplaces for worker units.

There are three simplifications that humans use to reduce this complexity. First, the number of possible workplaces is reduced to a few options based on the intended goal of a building. For example, defensive buildings are usually placed close to choke points. For other buildings, the location may not be important, as long as the chosen location is within the player’s base. Second, humans do not individually control large groups of units. Rather, they are placed into control groups, which group several units together. This reduces the number of orders needed to accomplish a task. Third, humans realize that units continue to perform tasks once issued. For example, a worker unit told to mine minerals will continue to mine minerals, unless destroyed. Therefore, the player can focus attention elsewhere once a worker unit has been issued an order. Given these reductions, the decision complexity of StarCraft, from a human’s perspective, is less than 100. This is similar in scale to the decision complexity of chess, which is approximately 30 [2].

An interesting gameplay space to analyze for RTS games is the strategy space, which defines the set of viable strategies in a game. In Wargus, the strategy space is small, because they are near-optimal strategies, such as the knights rush [62]. In StarCraft, there is no dominant strategy, because every strategy has a counter strategy. One approach to estimating the strategy space of a game is to take a combinatorics approach in which all possible permutations of strategies are enumerated. This approach is unsuitable for StarCraft, because most outcomes would be nonsensical. Rather, a knowledge-rich approach must be taken to identify the strategy space as well as determine if two different strategies are distinct enough to be considered unique. Rather than formulate an estimation of the strategy space of StarCraft, it can be observed based on analysis provided by the StarCraft community. For example, Liquipedia [73] lists 25 opening strategies for Terran versus Zerg games.
PRIOR WORK

My previous work on RTS games includes reactive planning, case-based reasoning and machine learning. My progress to date consists of several projects which focused on human authoring and learning from expert demonstrations as separate problems. While initial work used Wargus as a testbed for evaluating my research, more recent work has utilized StarCraft.

4.1 ABL WARGUS

My agent will build upon concepts introduced by McCoy and Mateas in the ABL Wargus project [47]. The goal of the project was to build a RTS AI that integrates multiple specialist components into a complete game playing agent. The presented agent incorporates several sources of domain knowledge by conceptually partitioning Wargus gameplay into domains of competence seen in expert gameplay. It is implemented in the ABL (A Behavior Language [45]) reactive planning language.

The agent consists of a set of managers, each of which performs a specialized task in Wargus. It contains the following managers:

- **Strategy manager**: selects high-level strategies for the agent to pursue.
- **Income manager**: handles resource management and worker units.
- **Production manager**: responsible for constructing building and training units.
- **Tactics manager**: handles tasks involving multi-unit military conflicts.
- **Recon manager**: performs scouting to acquire information about hidden state.

The strategy manager performs high-level decision making, while the remaining managers perform tasks in support of the selected strategy. An overview of the managers is shown in Figure 3.

I extended the ABL Wargus agent in two ways. The first modification was the addition of a case-based reasoning component for replacing the strategy manager [82]. The goal of adding a case-based reasoning component for selecting strategies was to enable the agent to pursue a larger number of strategies and adapt to strategies during a game. The case retrieval process was inspired by Turner's concept of Transform-Recall-Adapt Methods implemented in MINSTREL [75], but operates on a feature vector representation instead of a story graph. A case library was generated by running several different scripted builds against each other on different maps. Using case-based reasoning to implement the strategy manager reduced the number of hard-coded behaviors in the agent while maintaining a comparable win rate. Additionally, increasing the size of the case library was shown to improve the performance of the agent [81].

The second modification was the addition of behaviors enabling the agent to operate in games in which imperfect information is enforced. Scouting behaviors were added to the reconnaissance manager in order to track the opponent's expansion of the tech tree. The number of features describing opponent game state was reduced in order to handle noisy
4.2 Multi-scale Game AI

The successor to ABL Wargus is the EISBot\(^1\) project. The goal of the project is to build an agent capable of human-level gameplay in StarCraft, by applying McCoy and Mateas’ architecture\([47]\) to StarCraft and implementing additional competencies in the integrated agent. While the conceptual partitioning of gameplay into distinct competencies transfers well between Wargus and StarCraft, the complexity of StarCraft required additional coordination between managers in the agent. In the ABL Wargus agent, a minimal number of behaviors were implemented in the tactics manager for facilitating attacking opponents. Performing well in StarCraft required more specialized assignment of military units to specific tasks, because units are utilized in both individual and group behaviors during a game. The need to manage units at several levels of detail motivated a multi-scale approach to building a StarCraft agent.

A multi-scale AI is a system that simultaneously reasons about and executes actions at several levels of scale\([80]\). StarCraft gameplay is multi-scale, because it requires concurrent and coordinated goal pursuit across multiple competencies. Additionally, there are complex interactions between scales, such as concepts of timing and map control. For example, a \textit{timing attack} is a planned assault in which a player positions units in preparation for an attack such that the units are in position to attack as soon as an upgrade completes, optimizing the utility of the upgraded units. Successfully performing an effective timing attack requires coordination between strategic and tactical competencies. Due to the need for coordination between different competencies, purely layered-based approaches, such as subsumption architectures\([86]\), are unsuitable for building multi-scale game AI.

EISBot is implemented in the ABL reactive planning language, which provides mechanisms for concurrent goal pursuit in a real-time environment. Multi-scale reasoning across

---

\(^{1}\) EISBot is short for the Expressive Intelligence Studio Bot, pronounced “Ice Bot”.
competencies is achieved through the use of reactive planning idioms [80]. The following idioms are used in EISBot:

- **Daemon behaviors:** spawn new threads of execution, enabling the planner to pursue new goals in parallel with currently active goals.

- **Managers:** partition the agent’s behaviors into distinct domains of competence [47].

- **Message passing:** utilize ABL’s working memory as a blackboard [28], enabling communication between managers.

- **Micromanagement behaviors:** spawn new behavior trees for managing single units.

Using these design patterns, EISBot is capable of combining highly reactive unit micromanagement tasks with high-level strategic reasoning. Initial results showed that EISBot achieved a win rate of over 60% against the built-in StarCraft AI and performed over 200 game actions per minute on average.

### 4.3 Goal-Driven Autonomy

Performing well in StarCraft requires constantly adapting to opponents, because “no plan survives contact with the enemy” [32]. While the initial version of EISBot included reactive behaviors for low-level tactical tasks, the agent selected a fixed high-level strategy at the start of each game. In order to intelligently respond to game events, EISBot was modified to implement the Goal-Driven Autonomy (GDA) conceptual model [51, 54].

The goal of GDA is to build agents capable of intelligently reacting to planning opportunities and failures in a domain. Molineaux et al. [51] present a conceptual model for GDA containing the following components:

- **Discrepancy detector:** detects unanticipated situations.

- **Explanation generator:** explains why the situation needs attention.

- **Goal formulator:** selects goals for responding to the situation.

- **Goal manager:** executes selected goals.

One of the distinguishing features of the model is the output from the planning component. The planner in a GDA system generates plans which consist of actions to execute as well as expectations of world state after executing each action. Expectations enable an agent to determine if a failure has occurred during plan execution and provide a mechanism for the agent to react to unanticipated events.

EISBot was modified to utilize the GDA conceptual model by implementing the discrepancy detector, explanation generation, and goal formulation components in ABL [79]. Using GDA enabled a clear separation of the goal selection and goal execution logic in the agent. One of the main challenges in implementing the model in ABL was building expectations for the discrepancy detector. ABL does not require a commitment to a formal domain model and behaviors do not have explicit effects or postconditions. Therefore, a static set of expectations was used to enable the agent to react to a wide variety of events. For example, the agent has an expectation that the opponent will not produce new unit types. In response to detecting a violated expectation, EISBot selects a new high-level strategy using a collection of production rules.
The agent was evaluated against human players on a competitive ladder server\(^2\). Results are shown in Table 2. After 100 games, EISBot achieved a win rate of 37% and outranked 48% of players. These initial results demonstrated that EISBot was capable of adapting to a large number of strategies.

<table>
<thead>
<tr>
<th></th>
<th>Versus</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Protoss</td>
</tr>
<tr>
<td>Win-loss record</td>
<td>10-23</td>
</tr>
<tr>
<td>Win ratio</td>
<td>30%</td>
</tr>
<tr>
<td>ICCup Points:</td>
<td>1182</td>
</tr>
<tr>
<td>ICCup Rank:</td>
<td>33,639 / 65,646</td>
</tr>
</tbody>
</table>

Due to the complexity of real-time strategy games, hand-authoring agents that perform well against competitive players is an overwhelming problem. To overcome this issue, my previous work explored techniques for learning domain knowledge from expert demonstrations. Learning from demonstration enables a system to discover effective strategies without exhaustively exploring the state space. Additionally, systems that learn from demonstration can improve performance by adding new demonstrations to the knowledge base.

Demonstrations are available for StarCraft in the form of game replays, which can be mined from the web. Learning from demonstration enables the development of robust intelligence, where agent behavior is built from collective intelligence as opposed to a small number of designers [60]. In contrast to programming by demonstration, in which a domain expert performs a task by executing actions and then annotates the log of the actions [34], learning from demonstration does not require manual annotation of demonstrations. An additional challenge in learning from demonstrations is that real-world replays are noisy and contain spammed actions.

My initial work on learning from demonstration focused on performing strategy prediction by analyzing large collections of replays and training classification algorithms [83]. Replays were converted to logs of game actions and encoded as feature vectors, where each feature described when a unit or building type was first produced by a player. This encoding provides a compact representation for reasoning about a player’s expansion of the tech tree. Feature vectors were assigned a label based on a ruleset which identified common StarCraft strategies. Off-the-shelf classification algorithms [85] were applied to the task of predicting an opponent's strategy at various times throughout a game. Results showed that an opponent's strategy could be predicted with a large degree of confidence, even when provided with imperfect information. Additionally, in the early stages of the game multiple classification techniques were more accurate than the ruleset used to label the strategies, exhibiting "foresight".

\(^2\) http://www.iccup.com/starcraft/gamingprofile/scgda.html
More recent work has focused on building general techniques for predicting opponent game state, while reducing the amount of domain knowledge required. Case-based goal formulation is a technique for predicting an opponent’s future game state by exploiting the temporal structure of gameplay demonstrations [78], where a player’s future game states are inferred as goals. Case-based goal formulation was shown to outperform classification algorithms in an opponent modeling task. The case-based goal formulation component was integrated into EISBot and applied to the task of strategy selection. Given a game state, the component formulates a goal state for the agent to pursue and a plan to achieve the goal state. Case-based goal formulation has also been used to automate the process of annotating demonstrations for case-based planning [77].

4.5 Discussion

My prior research has investigated the use of reactive planning for hand-authoring agent behavior as well as case-based reasoning and machine learning for automatically acquiring domain knowledge from expert demonstrations. The main open issue is a lack of integration between these components: my work on opponent modeling is not yet utilized by the reactive planner, and components that learn from demonstration have been integrated only at the strategic level of the agent.

The main open challenges are applying machine learning and case-based reasoning to learn additional competencies from demonstrations, enabling the agent to intelligently operate in an imperfect information environment, and implementing additional reactive planning behaviors for highly-responsive tasks.
The goal of my proposed work is to develop techniques for building agents capable of operating in real-time, multi-scale, partially observable domains in which exploration of the state space is prohibitively expensive. To accomplish this goal, it will be necessary to integrate my previous work into a unified agent as well as add several new competencies to the system. EISBot will perform multi-scale reasoning by conceptually partitioning StarCraft gameplay into distinct competencies. The agent will utilize a layered architecture in which each competency is managed by a GDA implementation. This will enable EISBot to intelligently respond, in real time, to opportunities and failures at multiple levels of reasoning. The diverse set of techniques used by different GDA layers will be unified into an agent which extends McCoy and Mateas’ integrated agent architecture [47].

5.1 Architecture

The majority of EISBot’s functionality will be implemented in the ABL reactive planning language [45]. ABL is similar in flavor to Belief-Desire-Intention (BDI) architectures [66], such as PRS [27]. Beliefs, desires, and intentions are not explicitly modeled in ABL; they are incorporated into the behavior selection and execution logic. An ABL agent has a working memory which contains Working Memory Elements (WMEs) that describe the agent’s perception of world state, analogous to beliefs. Each behavior contains one of more steps, which can be subgoals, mental actions, or physical actions. The subgoaling mechanism in ABL is similar to HTN planning [22] and is analogous to creating new desires for the agent to achieve. Physical actions enable an agent to perform tasks in an environment and serve as the intentions of an agent. Therefore, agents implemented in the ABL reactive planning language conform to the BDI software model.

EISBot will integrate ABL with parallel GDA implementations. ABL behaviors will contain the majority of the execution logic for pursuing goals, while the GDA components will contain logic for selecting which goals need to be achieved. The agent will contain a GDA layer for each of its competencies. The components in a GDA layer are shown in Figure 4. Each layer contains instantiations of GDA components, which interface with the game environment and a working memory that serves as a blackboard. Additional discussion of the components in the GDA layers are provided in Section 5.4.

My implementation differs from the proposed GDA conceptual model of Molineaux et al. [51] in two ways. First, my system does not generate expectations in response to the agent’s intentions. This results from ABL behaviors not having explicit effects or postconditions. My system is unable to detect discrepancies, due to a lack of expectations. The discrepancy detector will be replaced with an event recognition component that identifies game events which may require the agent to pursue new goals. The result is additional complexity in the explanation generation component, which determines if game events correspond to an opportunity or failure given the agent’s current goals. The second difference is the lack of a classical planner. Instead of generating plans in response to selected goals, the system will immediately execute goals using ABL’s subgoaling mechanism. Despite these
differences, I claim that my system still implements goal-driven autonomy, because it triggering reasoning about goals in response to unanticipated events.

EISBot’s architecture will integrate the BDI software model with the GDA conceptual model. The outputs from the GDA components include explanations, goals, and actions. Explanations contain descriptions of current and future game state, augmenting the agent’s current set of beliefs. Goals specify new behaviors for the agent to pursue, and correspond to new desires. Actions specify physical actions for the agent to execute, resulting in new agent intentions. The interaction between BDI and GDA components is shown in Figure 5. EISBot’s architecture can be summarized as a BDI agent which augments beliefs, desires, and intentions through the use of the GDA conceptual model.

Implementation-level details of EISBot are shown in Figure 6. EISBot will run as a separate process from StarCraft and interact using socket-based communication with BWAPI\(^1\). The ProxyBot component provides a bridge between the reactive planner and the BWAPI interface, which enables the agent to query StarCraft state and issue orders to units. The ProxyBot component polls for game state and updates the working memory of the agent each game cycle, and buffers actions selected by the reactive planner. The ABL behaviors contain the logic needed for the agent to achieve goals selected for execution. The system also contains case-based reasoning (CBR) and machine learning (ML) components. The CBR and ML components utilize a collection of replays in order to formulate goals for the agent to pursue. Each GDA layer in the agent utilizes both ABL behaviors as well as CBR and ML components.

\(^1\) Brood War API: [http://code.google.com/p/bwapi/](http://code.google.com/p/bwapi/)
5.2 COMPETENCIES

StarCraft gameplay will be decomposed into distinct aspects of gameplay, similar to the original partitioning proposed by the ABL Wargus project [47]. A GDA layer will be implemented for each aspect of gameplay, enabling concurrent goal pursuit across multiple competencies. EISBot will include the following competencies:

- **Strategy**: manages high-level strategies and determines which buildings to construct, units to train, and upgrades to research
- **Economy**: responsible for worker units, managing the income ratio of different resources, and determining when to expand
- **Tactics**: manages map control and combat scenarios including small-scale attacks and large-scale assaults
- **Scouting**: responsible for scouting the opponent and predicting troop movements

Several of the competencies are composed of sub-competencies. For example, the strategy competency manages the build order, army composition, and base defense sub-competencies.

The system will facilitate coordination between different competencies using ABL’s working memory for message passing. One of the cross-cutting concerns in the agent is the concept of map control, which is how vulnerable a region of a map is to opponent attacks. For tasks such as building an expansion, the agent will first secure control of a region before building the expansion base. Ensuring this ordering of tasks requires coordination between the agent’s economy and tactics competencies. Building an expansion consists of three steps. First, the economy competency formulates the goal of building an expansion base and selects a suitable location. Next, the tactics competency responds to the economic goal of expanding by securing the selected expansion location. Finally, the agent begins constructing the expansion once map control has been established at the target location.

5.3 EXAMPLE SCENARIOS

Two example scenarios will be used to illustrate the ability of the agent to intelligently react to events. In the *Lurker Egg* scenario, the agent encounters a unit capable of morphing into a cloakable unit. The most common reaction to this game event is to build detection
5.3 Example Scenarios

**Event:** Opponent lurker egg in agent vision

**Explanation:** Opponent is researching cloaking technology

**Goal:** Build detector units

**Behavior:** Construct prerequisite buildings for detector units

**Behavior:** Train detector units

---

Figure 7: Expected agent decision making in the Lurker Egg scenario

---

**Event:** Agent has vision of opponent base

**Explanation:** Opponent is susceptible to a timing attack

**Goal:** Execute an early-upgrade timing attack

**Behavior:** Perform probe stop

**Behavior:** Build mass Zealots

**Behavior:** Research upgrade

**Behavior:** Move Zealots into position

**Behavior:** Attack opponent

---

Figure 8: Expected agent decision making in the Timing Attack scenario

---

units which provide vision of cloaked units. Responding to this event requires reacting at only the strategic competency. The expected agent behavior in the Lurker Egg scenario is shown in Figure 7. In response to scouting an egg, the agent formulates a goal of producing detector units, which results in executing behaviors for constructing prerequisite buildings and training detector units.

In the Timing Attack scenario, the agent scouts an opponent's strategy, which is susceptible to a timing attack. Two common reactions to this event are to perform a timing attack in order to overwhelm the opponent or focus on expanding in order to gain an economic advantage over the opponent. If the agent decides to perform a timing attack, coordination is required between the strategy, economy, and tactics competencies. The expected agent behavior for the Timing Attack scenario is shown in Figure 8. The agent reacts to scouting the opponent's base by formulating a goal to perform a timing attack. Several behaviors are required to achieve this goal: the probe stop behavior halts the production of worker units in order to maximize the number of combat units that can be produced by the build mass Zealots behavior, the research upgrade behavior begins a Zealot upgrade, the move behavior positions combat units in preparation for the timing attack, and the attack behavior initiates the assault on the opponent base.
5.4 COMPONENTS

Each competency in EISBot will be managed by a GDA layer, shown in Figure 4, which is composed of the following components:

- **Event detector**: detects game events that may require attention
- **Explanation generator**: generates explanations for important game events
- **Goal formulator**: selects goals for responding to generated explanations
- **Goal manager**: executes selected goals by subgoaling ABL behaviors

Each layer interacts with the game environment by executing physical actions, which result in updated game state. The different components in a layer communicate using a blackboard, which facilitates communication between different competencies. The following objects types are posted to the blackboard:

- **Event**: a game state change that may trigger goal formulation
- **Explanation**: a prediction of game state or opponent intentions
- **Opponent model**: maintains beliefs about the opponent’s state
- **Goal**: a task to accomplish

Each of the GDA components and blackboard object types are discussed in more detail below.

5.4.1 Event Recognizer

The event recognizer is responsible for detecting game state changes that may require a response. It takes the game state as input and generates events based on changes between the current state and previous state. The event recognizer does not reason about the implications of generated events; this task is performed by the other components in the system. Events do not have a one-to-one correspondence to competencies, because an event may trigger responses from multiple competencies. For example, in the Lurker Egg scenario, the agent may respond at the strategic competency by building detection units, while also responding at the tactical level by initiating an attack.

The types of events that can be generated by the event recognizer are defined by an event ontology. The event ontology will be a static, hand-authored knowledge representation for describing the events in a domain. While the ontology will be structured based on the game mechanics of StarCraft, it should be applicable to other RTS games with minimal modification. However, applying the event recognizer to a different game genre would require substantial domain engineering. Two types of events would be generated in the example scenarios: a newUnitType(lurkerEgg) event in the Lurker Egg scenario and a newUnitScouted(expansion) event in the Timing Attack scenario. The newUnitType(int unitType) event specifies that the agent has scouted a new unit type that is controlled by the opponent, while the newUnitScouted(int unitType) event specifies that a new instance of a unit type has been scouted by the agent.

The specification for the event recognizer component is shown in Figure 9. EISBot will include two implementations of the component: hard-coded Java rules for recognizing
5.4 Components

<table>
<thead>
<tr>
<th>Event Recognizer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs:</td>
</tr>
<tr>
<td>• Game State</td>
</tr>
<tr>
<td>Outputs:</td>
</tr>
<tr>
<td>• Events</td>
</tr>
<tr>
<td>Implementations:</td>
</tr>
<tr>
<td>• Hard-Coded Java Rules</td>
</tr>
<tr>
<td>• ABL Behaviors</td>
</tr>
</tbody>
</table>

Figure 9: Event recognizer specification

<table>
<thead>
<tr>
<th></th>
<th>Java Rules</th>
<th>ABL Behaviors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strategy</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Economy</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Tactics</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Scouting</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 3: Event recognition implementations by competency

Events, and ABL behaviors for generating events. Both implementations will utilize a set of hand-authored rules. Since the types of events that can be generated are static, as defined by the game mechanics, learning algorithms will not be used for the event recognizer component. The event recognizer implementations for each competency are shown in Table 3.

An example ABL behavior for generating newUnitType(int unitType) events is shown in Figure 10. The behavior has a set of preconditions that check for an enemy unit, bind its type to a variable, and check whether an event for the unit type is currently in working memory. If there is not currently a unit type event for the bound type, a mental act is used to place a new event into working memory.

```java
sequential behavior eventRecognizer() {
    precondition {
        (EnemyUnitWME type::type)
        !(NewUnitTypeWME type==type)
    }

    mental_act {
        workingMemory.add(new NewUnitTypeWME(type));
    }
}
```

Figure 10: An ABL behavior for recognizing new unit type events
5.4.2 **Explanation Generator**

The explanation generator monitors events and builds explanations of the current game state. Explanations can be created by EISBot in response to unanticipated events at several levels of detail. At the level of issuing commands to individual units, the agent will formulate new plans when identifying that a previously issued command failed to execute. At the strategic level, EISBot will adapt its high-level strategy in response to actions executed by the opponent. The goal of the explanation generator is to provide EISBot with the capability to reason able why a particular situation requires attention. The main purpose of the explanation component in EISBot will be to build estimations of the current and future game state and to form predictions of an opponent's goals.

The specification for the explanation generator is shown in Figure 11. The component takes as input game state and events, and outputs explanations. The system will perform game state estimation and maintain a list of game state predictions in an opponent model. The opponent model is described in more detail in Section 5.4.3.

Several algorithms will be used to implement the explanation generator component for the different competencies. ABL behaviors will be used to implement production rules for each of the competencies. For the strategy and economy competencies, case-based goal formulation [78] and strategy prediction [83] will be used to build explanations of opponent
behavior and track the opponent’s goals. For the tactical and scouting competencies, the agent will use simulation techniques to track hidden information.

In RTS games, there are two forms of hidden information. The first form is information that is occluded from the player’s vision due to the “fog of war”. Players overcome this limitation by performing active scouting to gain additional vision of the map. In addition to scouting, players have expectations of which actions an opponent will execute. At the tactical level of gameplay, players build expectations of where an opponent will concentrate forces on a specific map based on previous gameplay experiences. EISBot will emulate this model of reasoning about opponent troop placement through a combination of simulation and replay-mining techniques. EISBot will apply particle filters \[8\] to build trajectories of expected force movements. The system will identify patterns of force movement on specific maps, as shown in Figure 12, in order to build knowledge-rich particles which imitate movement patterns of players.

The second form of hidden information is future game state, which results from the actuation of a player’s intentions in the current game situation. At the strategic level, players form expectations of an opponent’s strategy in order to predict the future game state. My previous work has mined game traces to model these expectations and exhibited predictive capabilities \[83\]. However, at the tactical level, the time scale for reacting to future game state is much smaller. Players tend to follow rules of thumb at this level of detail, because planning ahead for each encounter would be cognitively overwhelming. This is an aspect of gameplay in which a computer potentially has an advantage over human players. Game tree search can be used to build predictions of a players future game state, but several modifications need to be made for adversarial search to be tractable in this domain \[10\]. Monte Carlo simulation has shown potential for improving the performance of units at the tactical level \[6, 18\], and could be leveraged within EISBot by utilizing it in small-scale combat scenarios.

To enable operation between different implementations of the explanation generation component, a fixed explanation ontology will be defined. While a large portion of the ontology will be specific to RTS games, and StarCraft in particular, there may be potential for reuse in similar domains, such as explanations generated by a particle filter algorithm for representing predictions in a spatial environments.

Two example explanations used by EISBot will be `enemyCloaking()` and `enemyFastExpanding()`, which notify the agent that an opponent is researching cloaking technology, or executing an economy-focused strategy, respectively. In the Lurker Egg scenario, the `enemyCloaking()` explanation will be generated in response to scouting units or structures that indicate cloak-
5.4 Components

### Goal Formulator

<table>
<thead>
<tr>
<th>Inputs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Game State</td>
</tr>
<tr>
<td>• Explanations</td>
</tr>
<tr>
<td>• Opponent Model</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Goals</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• ABL Behaviors</td>
</tr>
<tr>
<td>• Case-Based Goal Formulation</td>
</tr>
<tr>
<td>• Precondition Learning</td>
</tr>
</tbody>
</table>

Figure 13: Goal formulator specification

...ting technology. This game situation is relevant to the agent, because additional branches of the tech tree may need to be expanded for the agent to counter cloaked units. In the Timing Attack scenario, the enemyFastExpanding() explanation will be created in response to scouting an opponent expansion early in the game. It is necessary for the agent to react to this event, because getting “behind on economy” could cost the game.

5.4.3 Opponent Model

EISBot will operate in an imperfect information environment. To deal with this additional complexity, the agent will include an opponent model to manage estimations of the game state. The model will be implemented using ABL's working memory and will track game state across several competencies:

- **Tech tree expansion**: monitors the opponent’s expansion of the tech tree
- **Army composition**: estimates the size and composition of opponent forces
- **Economy strength**: monitors the opponent’s income rate
- **Army placement**: tracks the expected locations of opponent troops

The opponent model will be used primarily by the expectation generator and goal formulation components. The explanation component will post beliefs about opponent actions that are utilized by the goal formulator when parameterizing goals.

5.4.4 Goal Formulator

The goal formulator is responsible for selecting goals for the agent to pursue. Its specification is shown in Figure 13. The component is given the opponent model and a set of explanations as input, and selects goals to execute. Goals in the system will be specified as ABL behaviors that can be selected for execution One of the design goals of the goal formulator component is to externalize precondition checks that are directly encoded in ABL behaviors and work towards the goal of partial specification for agent programs [71].
Table 5: Goal formulator implementations by competency

<table>
<thead>
<tr>
<th>ABL Behaviors</th>
<th>Case-Based Goal Formulation</th>
<th>Precondition Learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strategy</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Economy</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Tactics</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Scouting</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

EISBot will perform goal formulation using two forms of lazy learning. In the first approach, case-based goal formulation [78] will be used to formulate new goal states for the agent using instance-based retrieval. While previous work applied this technique to strategy selection, my proposed work will explore other competencies that could benefit from this approach. I will also evaluate lazy learning for precondition learning [33]. In this approach, preconditions for an ABL behavior could be specified as a set of training instances and a distance metric. One of the major challenges introduced by this approach is the knowledge engineering burden of specifying a dataset for training. To reduce this burden, my proposed work will explore techniques for automatically building datasets from game replays [77]. To make this task feasible, I will be focusing on automated case extraction for narrow sub-domains within StarCraft.

One potential way of integrating lazy learning into EISBot is to include the retrieval logic into preconditions, as shown in Figure 14. The triggerAttack behavior performs retrieval by iterating through a set of cases and checking if the case state and current gate state are similar enough to trigger behavior activation. It first binds the case's state variable to the caseState variable then computes the distance using the DistanceToCurrentState method. Upon retrieving a case within the specified threshold, the attack behavior is subgoaled. In this example, the behavior requires a case library which demonstrates situations in which attacking the opponent is a preferred action. The case library will be built by extracting examples from game traces in which a player performed an attack. This task will be automated by writing rule sets for identifying the characteristics of an attack, as in previous work [77], and through the use of explanation-based learning to pinpoint the actions in a trace that explain the pursuit of a particular goal [37].

Behaviors that can be selected by the goal formulation component will be hand-authored ABL code. In the Lurker Egg example, the buildDetectorUnits() behavior is selected in response to the enemyCloaking() explanation. Behavior selection for the Timing Attack scenario is more complicated, because achieving the goal of a timing attack consists of...
5.5 Example Execution

This section provides an overview of the decision making process of EISBot while performing a scouting task. The goal is to demonstrate how the components in the proposed agent will work together to achieve the goal of locating an opponent. Accomplishing this task requires several iterations of the scouting competency’s GDA layer, because the agent may need to scout multiple locations before determining the location of the opponent. In this scenario, the agent uses a worker unit, unit1, to search between four known starting locations, L0, L1, L2, and L3. The iterations performed by the scouting GDA layer are shown in Table 6.

In the first iteration of the scouting competency, the agent initializes its opponent model based on knowledge currently available. At the start of the game an event is generated called gameStart(). The explanation generator responds to this event by predicting the likelihood that the opponent is at each starting location. Given that EISBot is at L0 and several behaviors that need to be executed, as shown in Figure 8. There is also a partial-ordering between these behaviors, in which the last behavior should not begin execution until the other behaviors have completed. Once the selected goals have been partially ordered, they are passed to the goal manager for execution, which results in the agent performing physical acts in the game environment.

5.4.5 Goal Manager

The goal manager provides the logic for executing goals selected by the formulation algorithms. It will be implemented in ABL, which serves as the glue for the integrated agent [45, 47]. The specification of the goal manager is shown in Figure 15. The primary task of the goal manager, with respect to the GDA model is to implement a goal management policy. This includes rules for goal replacement, suspension, and prioritization.

My proposed system will build upon my previous work on applying reactive planning to StarCraft. Additional functionality will be implemented for each of the competencies. At the strategic level, behaviors will be added for fully expanding the tech tree, which were not present in past work. At the tactics level, logic for building and managing squads will be used to improve the utility of units. At the individual unit level, additional micromanagement behaviors will be added to the agent.

### Table 6: Iterations of the Scouting GDA Layer

<table>
<thead>
<tr>
<th>Location</th>
<th>Iteration 1</th>
<th>Iteration 2</th>
<th>Iteration 3</th>
<th>Iteration 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 15: Goal manager specification

In the first iteration of the scouting competency, the agent initializes its opponent model based on knowledge currently available. At the start of the game an event is generated called gameStart(). The explanation generator responds to this event by predicting the likelihood that the opponent is at each starting location. Given that EISBot is at L0 and several behaviors that need to be executed, as shown in Figure 8. There is also a partial-ordering between these behaviors, in which the last behavior should not begin execution until the other behaviors have completed. Once the selected goals have been partially ordered, they are passed to the goal manager for execution, which results in the agent performing physical acts in the game environment.
Table 6: Visualization of EISBot’s decision making process while performing a scouting task. The annotations on the down arrows correspond to the component that generates the output objects: Explanation Generator (EG), Goal Formulation (GF), and Goal Manager (GM)

<table>
<thead>
<tr>
<th>Iteration 1</th>
<th>Iteration 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>event: gameStart()</td>
<td>event: agentSupply(10)</td>
</tr>
<tr>
<td>$\downarrow_{EG}$</td>
<td>$\downarrow_{EG}$</td>
</tr>
<tr>
<td>Update opponent model</td>
<td>explanation: unknownOpponentLocation()</td>
</tr>
<tr>
<td></td>
<td>$\downarrow_{GF}$</td>
</tr>
<tr>
<td></td>
<td>goal: scout(L1)</td>
</tr>
<tr>
<td></td>
<td>$\downarrow_{GM}$</td>
</tr>
<tr>
<td></td>
<td>behavior: move(unit1,L1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration 3</th>
<th>Iteration 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>event: atLocation(unit1,L1)</td>
<td>event: atLocation(unit1,L3)</td>
</tr>
<tr>
<td>$\downarrow_{EG}$</td>
<td>event: opponentInVision(unit1)</td>
</tr>
<tr>
<td>explanation: unknownOpponentLocation()</td>
<td>$\downarrow_{EG}$</td>
</tr>
<tr>
<td>Update opponent model</td>
<td>explanation: opponentAt(L3)</td>
</tr>
<tr>
<td>$\downarrow_{GF}$</td>
<td>Update opponent model</td>
</tr>
<tr>
<td>goal: scout(L3)</td>
<td>$\downarrow_{GM}$</td>
</tr>
<tr>
<td>$\downarrow_{GM}$</td>
<td>behavior: move(unit1, L3 )</td>
</tr>
</tbody>
</table>
starting locations are chosen randomly, the opponent model for starting locations is set to the following state: \(\{L_0 \rightarrow 0, L_1 \rightarrow 0.33, L_2 \rightarrow 0.33, L_3 \rightarrow 0.33\}\). During the initial iteration, no explanations are generated.

Once the agent reaches a supply count of 10, an `agentSupply(10)` event is generated by the system. During the second iteration of the GDA layer, EISBot responds to the event by creating an explanation that the location of the opponent is unknown. The goal formulator reacts to the explanation by selecting a target location to scout based on the opponent model and creates a goal to scout the selected location. The goal is then executed by the reactive planner, which assigns the worker unit an order to move to location \(L_1\).

When the scouting unit researches its destination, \(L_1\), the system generates a new event. In the third iteration of the agent, the explanation generator responds to the event based on whether or not an opponent unit entered the scouting unit’s vision while traversing to the selected destination. In this example scenario, it is assumed that the unit did not encounter any opponent units. The agent creates another explanation to scout the opponent, given that the opponent’s location is still unknown. The agent also updates the opponent model given the knowledge that the opponent is not at location \(L_1\), resulting in the following state: \(\{L_0 \rightarrow 0, L_1 \rightarrow 0, L_2 \rightarrow 0.5, L_3 \rightarrow 0.5\}\). Next, the unit is ordered to move to location \(L_3\).

In the final iteration of the GDA layer, the unit encounters an enemy unit. In response to this event, the agent forms an explanation that the opponent’s base is at location \(L_3\). EISBot updates the opponent model as well stating that the starting location is now known. The agent now knows the location of the opponent base and this knowledge can be used by the other competencies, such as the tactics competency for selecting an attack target.

5.6 Evaluation

The agent presented in this proposal will be evaluated based on my hypothesis:

Integrating learning in a reactive planner enables the development of agents capable of adapting to unforeseen situations in complex domains.

The techniques developed for augmenting reactive planning with experience will be evaluated at the algorithmic, component, and system levels. At the algorithmic level, offline experiments will be run to compare the performance of different algorithms for implementing components in the system. The goal of the algorithmic-level evaluation is to select the best algorithm for each component. Component-level evaluation will be performed to demonstrate the utility of each component in the system. Ablation studies will be used to validate that each component improves the overall performance of the agent. The complete system will be evaluated based on win rates against several different types of opponents: the built-in AI, other StarCraft bots\(^2\), and competitive human players. The goal of system-level testing is to demonstrate the ability of EISBot to perform at the level of competitive human players in a complex, real-time domain.

\(^2\) http://eis.ucsc.edu/StarCraftAICompetition
The target date for completing my dissertation is summer quarter of 2012. A timeline of tasks to be completed to achieve this goal is shown in Table 7. Target publications building towards my dissertation are shown in Table 8.

### Table 7: Schedule

<table>
<thead>
<tr>
<th>Year</th>
<th>Quarter</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
<td>Fall</td>
<td>System infrastructure</td>
</tr>
<tr>
<td>2011</td>
<td>Winter</td>
<td>Data mine additional competencies</td>
</tr>
<tr>
<td></td>
<td>Spring</td>
<td>Integrate data mining results into EISBot</td>
</tr>
<tr>
<td></td>
<td>Summer</td>
<td>Additional replay mining</td>
</tr>
<tr>
<td></td>
<td>Fall</td>
<td>Opponent modeling and state estimation</td>
</tr>
<tr>
<td>2012</td>
<td>Winter</td>
<td>User and ablation studies</td>
</tr>
<tr>
<td></td>
<td>Spring</td>
<td>Dissertation writing</td>
</tr>
<tr>
<td></td>
<td>Summer</td>
<td>Finish writing and defend</td>
</tr>
</tbody>
</table>

### Table 8: Target publications

<table>
<thead>
<tr>
<th>Publication</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI Magazine</td>
<td>Report on the StarCraft AI Competition</td>
</tr>
<tr>
<td>AIIDE 2011</td>
<td>Work on initial integration of CBR and ML in EISBot</td>
</tr>
<tr>
<td>AAMAS 2012</td>
<td>Initial ablation studies demonstrating learning in EISBot</td>
</tr>
<tr>
<td>IEEE TCIAIG</td>
<td>Article on integration of ABL, CBR, and ML in EISBot</td>
</tr>
<tr>
<td>AAAI 2012</td>
<td>Description of the EISBot architecture and knowledge representations</td>
</tr>
<tr>
<td>ICCBR 2012</td>
<td>Results on mining experience from the web</td>
</tr>
<tr>
<td>AIIDE 2012</td>
<td>Applying lessons learned from EISBot to commercial game AI</td>
</tr>
</tbody>
</table>
CONCLUSION

My dissertation work will build towards the goal of overcoming the challenges necessary to create robust intelligence for complex domains. I propose integrating several existing AI methodologies into a unified agent that learns from expert demonstrations. While the system will apply techniques from several areas of AI, the main impact of my work will be in the areas of reactive planning, case-based reasoning, and agent architectures. The expected outcomes of integrating these approaches are techniques for building adaptive agents and reducing the amount of domain engineering required to build agents for complex, real-time domains.
BIBLIOGRAPHY


