UNIVERSITY OF CALIFORNIA
SANTA CRUZ

INTEGRATING LEARNING IN A MULTI-SCALE AGENT

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Ben G. Weber

June 2012

The Dissertation of Ben G. Weber
is approved:

Professor Michael Mateas, Chair

Professor Arnav Jhala

Dr. David W. Aha

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Integrating Learning in a Multi-Scale Agent

by

Ben G. Weber

Video games are complex simulation environments with many real-world properties that need to be addressed in order to build robust intelligence. StarCraft is a real-time strategy (RTS) game that exhibits both cognitive complexity and task environment complexity. Expert StarCraft gameplay involves many cognitive processes including estimation, anticipation, and adaptation. Achieving the objective of destroying all enemy forces requires managing a number of concurrent subtasks while working towards higher-level objectives. Working towards the goal of building expert-level performance for RTS games presents a multi-scale AI problem, which motivates the need for integrative AI systems.

This thesis investigates the capabilities necessary to realize expert StarCraft gameplay in an agent. My central claim is that in order to perform at the level of an expert player, a StarCraft agent must utilize heterogeneous reasoning capabilities. This requirement is motivated by the structure of RTS gameplay, which involves both deliberative and reactive decision making, and analysis of professional gameplay, which demonstrates the need for estimation, adaptation, and anticipation reasoning capabilities. Additionally, StarCraft gameplay involves decision making across multiple scales, or levels of coordination. My approach for supporting these capabilities in an agent is to

identify the competencies necessary for RTS gameplay, and develop techniques for implementing and integrating these competencies. The resulting agent, EISBot, integrates reactive planning for plan execution and monitoring, machine learning for opponent modeling, and case-based reasoning for goal formulation and strategy learning. EISBot plays StarCraft at the same action and sensing granularity as human players, and is evaluated against AI and human opponents.

The contributions of this thesis are idioms for authoring agents for multi-scale AI problems, techniques for learning domain knowledge from gameplay demonstrations, and methods for integrating a variety of learning algorithms in a real-time, multi-scale agent.

Acknowledgments

I would like to thank my advisors, Michael Mateas and Arnav Jhala, for their supervision and guidance throughout my dissertation. I would also like to thank David Aha for serving on my dissertation committee and serving as a mentor during the advancement process.

I learned a great deal from my collaborations throughout the dissertation process. I would like to thank Peter Mawhorter for his help in shaping the development of EISBot and Santiago Ontañón for help developing methods that learn from demonstrations.

I would also like to thank all of the members of the Expressive Intelligence Studio for creating an engaging social environment and providing invaluable support. Josh McCoy provided an excellent foundation for RTS research that helped motivate my research. James Skorupski and Gillian Smith participated in a number of interesting discussions, and helped shape my project. Adam Smith provided engaging brainstorming sessions about any topic at hand. Outside the lab, Teale Fristoe and Chris Lewis provided encouragement and support.

My dissertation work was supported by the National Science Foundation under Grant Number IIS-1018954 and the JL Moore Fellowship offered by Cal Poly's computer science department. I would also like to thank my parents for providing support during my studies as well as offering encouragement and moral support.

Chapter 1

Introduction

One of the central goals of artificial intelligence (AI) is to emulate the cognitive abilities demonstrated by humans in computational systems. Humans exhibit a broad range of reasoning capabilities and are able to accomplish goals in complex environments. In the domain of strategy games, expert players perform a number of cognitive processes for building plans of action, anticipating the actions of other players, and reasoning under uncertainty. Games provide an environment in which these capabilities can be observed, emulated, and evaluated.

AI has a long history of using games as a testbed for advancing the state of the field [91]. Video games provide complex simulation environments that require intelligent decision making to perform at expert human level. One of the benefits of using games for evaluating decision making systems is that it bypasses many of the problems encountered when building systems that interact with the real-world [48]. Using a simulated environment enables direct analysis of the complexity of a task, independent

of perception and action concerns. Gameplay involves a number of cognitive processes, and games provide an environment in which to study intelligent decision making.

Real-time strategy (RTS) games provide several research challenges that need to be addressed in order to build real-world AI systems. RTS gameplay involves decision making in a real-time, partially observable environment with an enormous decision complexity [3]. RTS games are multi-scale and require concurrently reasoning about goals at multiple levels of coordination [109]. An initial call for research in RTS games was proposed by Michael Buro at IJCAI in 2003 [13]. 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. More generally, John Laird and Michael Van Lent claim that games provide the types of problems that can lead to incremental and integrative advances in AI [49]. While significant effort has been applied to developing AI for RTS games (e.g. [64, 74, 111]), building expert-level AI for RTS games remains an open research problem.

This thesis explores the development of an agent for the RTS game *StarCraft: Brood War*. I selected this game because it is well studied by humans, presents a deep strategy space, provides several data sources for learning, and has an active player base. The game is played at a professional level and agents developed for StarCraft can be evaluated against human opponents with a strong understanding of the game. This thesis works towards the AI vision of emulating intelligent decision making in a computational system by identifying a number of competencies needed for expert StarCraft gameplay, learning additional behavior from expert demonstrations, and integrating

Figure 1.1: In real-time strategy games, players take the role of a commander in a military scenario and pursue objectives including securing areas of the map and destroying opponent forces. Popular RTS games include *StarCraft II* by Blizzard Entertainment (left) and *Command & Conquer: Generals* by Electronic Arts (right).

these capabilities in a game-playing agent.

1.1 Real-Time Strategy Games

Real-time strategy is a genre of video games in which a player takes the role of a commander, usually in a military scenario. The player performs actions by giving orders to units or squads within a theater of operation. The objectives assigned to the player can include securing areas of the map, gaining control of specific assets, and destroying opponent forces or bases. Achieving these objectives requires performing a number of different subtasks such as economy management, production, technology expansion, reconnaissance, and tactics. Two popular commercial RTS games are shown in Figure 1.1.

RTS games present complex tasks and learning to play is difficult for both humans and AI systems. It is difficult to evaluate the reward of individual commands,

because the game duration is often large and may involve thousands of commands. Additionally, making a mistake early in the game can have drastic consequences. A key difference from other forms of strategy games is that decisions are made in real-time, which also has the side effect that inaction is extremely detrimental. One way expert players approach this task is by studying a corpus of games, building models for anticipating opponent actions, and practicing within the game environment in order to develop strong gameplay mechanics.

1.2 Motivation

There are a number of ways in which building agents for StarCraft, and more generally RTS games, motivates AI research. Playing StarCraft requires managing a number of subtasks, which work towards the high-level objective of defeating opponents. It involves concurrent and coordinated problem solving across a broad range of competencies, including reactive and deliberative decision making. I classify the task of playing StarCraft as a *multi-scale* AI problem, which is presented in more detail in Section 4.1. Multi-scale problems motivate heterogeneous agent architectures, because they involve problem solving across multiple abstractions and require a variety of reasoning capabilities.

Another challenge in developing AI for RTS games is dealing with the massive state space. RTS games feature hundreds of units with many numeric properties and fast-paced action, resulting in an astronomical number of states. Due to the size of the

state space, it is difficult to apply exploratory approaches which can learn effective policies online. One way of dealing with this problem is to bootstrap the learning processes using examples from expert demonstrations. Because StarCraft presents a massive state space, but has many expert demonstrations available for analysis, it motivates work in learning from demonstration. Over the past decade, professional players have generated a huge collection of StarCraft replays that can be harnessed by agents that learn from demonstration.

Strategy games with no dominant strategy have a constantly evolving meta game. In StarCraft, new strategies are constantly being developed to counter the current most effective strategies, which results in new types of strategies being employed by players. This evolution is also the result of different maps being added to the game, which support additional types of gameplay and strategies. In order to perform at an expert level in strategy games it is necessary to track the evolving meta game to learn which strategies are most effective in specific match ups. This aspect of strategy gameplay motivates work in strategy learning and adaptation.

StarCraft enforces imperfect information through a *fog-of-war*, which limits visibility to portions of the map where units are controlled by the player. In order to determine which actions are being performed by opponents, it is necessary to actively scout the map to find the location of enemy forces, uncover the technology options unlocked by opponents, and evaluate the economic growth of opponents. One way players manage uncertainty is by building expectations of opponent actions and using these expectations to develop counter strategies. Reproducing this behavior motivates

work in building state estimation and anticipation capabilities for agents.

One of the benefits of developing AI for a popular game is that the system can be evaluated against human experts. Rather than limiting evaluation to a set of benchmark problems or static opponents, using human participants in the evaluation process motivates the development of agents capable of adapting, learning, and responding to unforeseen game situations.

1.3 Objectives

My main objective for this work is to identify the capabilities necessary for expert StarCraft gameplay and to realize these capabilities in a game-playing agent. To achieve this goal, I investigate the following research questions:

1. What competencies are required for expert RTS gameplay?

2. Which competencies can be learned from demonstrations?

3. How can distinct competencies be integrated in a real-time agent?

Investigating these questions works towards the goals of building human-level AI for RTS games and reproducing a subset of the cognitive capabilities demonstrated by humans.

In order to demonstrate that an agent emulates many of the capabilities necessary for RTS gameplay, the system should be evaluated in an environment that resembles the interface provided to human players as closely as possible. Specifically, the agent

should not be allowed to utilize any game state information not available to a human player, such as the locations of non-visible units, and the set of actions provided to the agent should be identical to those provided through the game's user interface. Enforcing this constraint ensures that an agent which performs well in this domain can demonstrate estimation and anticipation reasoning capabilities.

RTS gameplay involves decision making across multiple levels of coordination. Units can be controlled individually, at a squad level, or globally. I refer to problems that involve decision making and acting across multiple levels of coordination as multi-scale AI problems. StarCraft is an instance of a multi-scale AI problem, and one of the objectives of this work is to investigate methods for authoring agents that perform multi-scale reasoning.

Another objective of this work is to leverage domain knowledge extracted from gameplay demonstrations. While several competencies can be implemented using hand-authored rules from a domain expert, this approach is unable to handle the evolving meta-game of StarCraft. The motivation for learning from demonstrations is to enable capabilities for learning new behaviors and managing the evolving meta-game of StarCraft. Another benefit of developing techniques for learning from demonstrations is that it provides mechanisms for reducing the amount of hand-specified behavior in an agent.

The objective of building a complete game-playing agent that interacts with human opponents is motivated by the research areas of cognitive systems and expressive AI. My approach to building an agent for complete gameplay, as opposed to solving

specific problems within RTS games, is inspired by system-level research in cognitive systems [50]. The goal of building an AI system that interacts with players is inspired by the interaction between AI systems and audiences common in expressive AI [58]. One of the outcomes of this approach is that the resulting system can be evaluated with respect to human-level intelligence.

1.4 Contributions

I have focused on the goal of building human-level AI for RTS games. The main contributions of this work are:

- *Multi-Scale Idioms.* I classify StarCraft gameplay as a multi-scale AI problem and present design patterns for authoring agents which perform multi-scale tasks. These idioms build upon the ABL reactive planning language [59], and enable authoring agents that perform concurrent and coordinated goal pursuit. These idioms are used to extend the integrated agent framework of McCoy and Mateas [61], which decomposes RTS gameplay into domains of competence. The resulting system, *EISBot*, uses message passing, unit subtask, and manager idioms to incorporate specialized hand-authored behaviors within a reactive planning agent.

- *Learning from Demonstration.* I develop methods that learn from demonstrations and enable estimation, anticipation, and adaptation capabilities in an agent. My approach uses game replays from professional StarCraft players as a source of gameplay demonstrations. I explore three applications of gameplay demonstra-

tions: model training for classification and regression algorithms that identify the strategy an opponent is performing and estimate when specific technologies will be produced by a player, case-based goal formulation for selecting strategies for the agent to pursue and anticipating the goals of opponents, and parameter selection for a particle model that tracks opponent forces. I present a number of experiments in which these different models are evaluated offline.

- *Integrated Agent.* I extend the integrated agent framework of McCoy and Mateas [59] to support external, heterogeneous components. Approaches for integrating these components include augmenting working memory, external goal formulation, external plan generation, and behavior activation. These integration approaches build on the message passing idioms used within the reactive planner and enable a mix of deliberative and reactive reasoning processes in the agent. An additional way in which components are integrated in EISBot is through the goal-driven autonomy conceptual model [65], which identifies a framework for building self-introspective agents.

The result of this work is a system that integrates a variety of learning algorithms in a multi-scale agent and performs at the level of a competitive amateur player. An additional outcome of this work is the first evaluation of a StarCraft agent with competitive human participants, providing a high-water mark for machine play.

1.5 Organization

Chapter 2 presents an overview of related work in agent architectures, game AI, opponent modeling and learning in games. It focuses on related work in RTS games.

Chapter 3 provides an overview of StarCraft and the tasks involved in gameplay. It also presents an analysis of the domain characteristics and complexity.

Chapter 4 defines multi-scale AI and presents three examples of multi-scale AI problems. The chapter introduces design patterns that are built upon the ABL reactive planning language that support the authoring of agents that perform multi-scale reasoning. These design patterns are applied in EISBot to play complete games of StarCraft.

Chapter 5 presents methods for learning gameplay models from expert demonstrations. The models enable an agent to anticipate opponent actions, adapt to strategies as gameplay evolves, and estimate opponent locations. I discuss three ways to use demonstrations: classification and regression model training, parameter selection for model optimization, and case-based goal formulation.

Chapter 6 introduces approaches for integrating gameplay models learned from demonstrations. These integration approaches enable the ABL reactive planner to interface with external components that augment working memory, generate plans, formulate goals, and activate behaviors. The chapter then discusses how these design patterns are used in EISBot and provides an overview of the agent implementation.

Chapter 7 presents an evaluation of EISBot. Ablations of the system are

evaluated versus computer and human opponents.

Chapter 8 provides a recap of the claims presented in this thesis and discusses directions for future work.

Chapter 2

Related Work

The work presented in this thesis draws upon a variety of research areas including agent architectures, case-based reasoning, and machine learning. Building human-level AI for RTS games involves integrating heterogeneous reasoning components, dealing with the complexity of game environments, supporting capabilities for estimation and anticipation, and learning from demonstration and experience. This chapter provides an overview of related work in these areas, focusing on the application to RTS games.

2.1 Agent Architectures

A number of agent architectures have been used to build agents for games. The most suitable architecture for a particular task depends on the goals of the system, which can include support for designer-specified behavior and operating within computational constraints. For real-time strategy games, an agent architecture should

enable reactive and deliberative reasoning capabilities, support mechanisms for strategy learning, and track anticipated opponent actions. My system builds on ideas from cognitive architectures, goal-driven autonomy, and reactive planning in order to realize these capabilities.

2.1.1 Cognitive Architectures

Cognitive architectures work towards the goal of developing mechanisms that underlie human cognition and provide many of the mechanisms required for integrating heterogeneous competencies [53]. Research in cognitive architectures strives to develop integrated systems that exhibit broad capabilities and focuses on performing evaluation at the system level [52]. Langley et al. identify several open issues for cognitive architectures including representational frameworks that move beyond production systems and plans, and robust learning mechanisms that extend to complex and unfamiliar domains [52]. While cognitive architectures make strong claims about modeling human cognitive processes, my system makes no such claims.

ICARUS is a cognitive architecture for physical agents that incorporates concepts from cognitive psychology [51]. One of the goals of ICARUS is to build an architecture capable of concurrently reasoning about multiple goals, which can be interrupted and resumed. The system has components for perception, planning, and acting which communicate through the use of an active memory. ICARUS uses means-ends analysis when confronted with a new problem situation. The ICARUS cognitive architecture has been applied to real-time domains including an urban driving simulation [20] and the

first-person shooter game *Urban Combat* [21]. While ICARUS provides many of the capabilities necessary for expert RTS gameplay, it lacks fine-grained controlled structures for authoring highly-specialized behaviors, such as micromanagement actions.

SOAR is a cognitive architecture that performs state abstraction, planning, and multitasking [53]. It uses a learning technique called *chunking* that operates as a caching mechanism and enables the system to intermix learning and problem solving. SOAR has been applied to the task of playing real-time strategy games [111]. The agent includes a middleware layer that serves as the perception system and gaming interface, global coordinator for forming and managing groups of units, and finite-state machines for implementing micromanagement actions. RTS games provide a challenge for the architecture, because gameplay requires managing many cognitively distant tasks and there is a cost for switching between these tasks. The main difference from my approach is that my system has a larger focus on coordination across tasks.

2.1.2 Goal-Driven Autonomy

The goal-driven autonomy (GDA) conceptual model provides a framework for creating agents capable of responding to unanticipated failures during plan execution in complex, dynamic environments [65]. It is motivated by Michael Cox's claim that agents should reason about their goals in order to continuously operate with independence [23]. The conceptual model specifies subtasks that enable an agent to detect, reason about, and respond to unanticipated events, and provides a framework for building agents capable of operating in complex domains. 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 [105] and case-based reasoning [67].

The GDA model has been used to build autonomous agents for naval strategy simulations [65], first-person shooters [68, 66], and real-time strategy games [45]. These systems require a domain expert to specify expectations for every action, explanations for every discrepancy, and goals for every explanation. To reduce the amount of domain engineering required to author GDA agents, Muñoz-Avila et al. use case-based reasoning to predict expected states and formulate goals in response to discrepancies [67]. Jaidee et al. build upon this work and apply reinforcement learning to learn expected values for goals [46]. My approach generates expectations, explanations and goals using case-based reasoning as well, but learns a case library by directly capturing examples from professional demonstrations rather than online learning. Another difference in my approach is that intent recognition is used to generate explanations of future world state rather than generating explanations of the current world state.

2.1.3 Reactive Planning

Reactive planning has been applied to creating autonomous characters for virtual environments [54, 58]. 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 [85]. Reactive planning is well suited for real-time environments,

because the mechanism for behavior selection is strictly bounded and efficient [54]. One of the strengths of reactive planning is support for acting on partial plans while pursuing goal-directed tasks and responding to changes in the environment [47].

A Behavior Language (ABL) is a reactive planning language developed to support the creation of the believable characters in the interactive drama Façade [60]. ABL adds significant features to the original Hap [54] semantics, including first-class support for meta-behaviors and joint intentions across teams of multiple agents [59]. ABL is well suited for developing autonomous agents, because it was developed to achieve the following goals: reactivity, responsiveness, deliberation, and explicit goals [58]. 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 [59]. McCoy and Mateas applied ABL to the task of playing complete games of *Wargus* [61], an open-source clone of *WarCraft II*.

An ABL agent can be extended in a number of ways. For example, the agent's working memory can be used as a blackboard [36] to enable communication with an external goal formulation component. Additional methods for extending ABL agents are presented in Section 6.2. Another extension of ABL is the use of reinforcement learning to enable partial specification of ABL agents [94].

2.2 Game AI

AI has been used for several processes in games, with a large focus on authoring non-player characters (NPCs). Building AI for games presents several challenges, because games are complex simulation environments with many real-world properties. Game AI is an instance of expressive AI [58], in which the primary goal is to provide players with an engaging experience. One of the primary roles of game AI is to provide tools which enable designers to author engaging gameplay experiences within virtual environments.

A challenge in building commercial game AI is managing the severe computational constraints placed on the AI system [13]. While game platforms are increasingly becoming more powerful, enabling additional computational resources to be allocated to AI, several additional challenges are present. Ideally, techniques for game AI should enable flexibility during development [41], mechanisms for explaining why actions have been selected, and tools for managing authoring complexity.

Finite state machines (FSMs) are one of the most commonly used techniques for building game AI [87]. They are popular due to their efficiency, simplicity and expressivity [31]. 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 straight-forward 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 [31], 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 [41]. One of the major limitations of FSMs is that logic encoded in this representation cannot be reused across different contexts [77].

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 [114]. 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 become an adopted technique for building commercial game AI [16, 41]. 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 [43]. 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 [105]. This drawback applies to reactive planning as well, and authoring ABL agents involves substantial domain engineering.

Goal-oriented action planning (GOAP) is a technique for building game AI using planning [76]. In a GOAP architecture, a character has a set of goals, each of which is mapped to a set of trigger conditions. When the trigger conditions for a goal become true, the system begins planning for the activated goal. GOAP architectures utilize a STRIPS-style representation [30] 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 [38]. GOAP was first implemented in the FPS game F.E.A.R [77] and has been applied to building game AI for additional genres.

2.3 Opponent Modeling

RTS games are imperfect information environments in which only a portion of the map is visible to the player. In order to operate with this constraint, players form estimations of the current game state and build anticipations of opponent actions. Building expectations of game state based on adversarial actions is a form of opponent modeling. There are two ways opponent modeling can be used in a game-playing agent: state estimation is the process of building predictions of the current game state based on prior observations, and plan recognition is the process of building predictions of future

game state based on anticipated opponent actions. Van den Herik et al. [101] identify a variety of techniques used for opponent modeling in games: evaluation functions [6], machine-learned function approximators [103], probabilistic models [17], and case-based models [106].

2.3.1 State Estimation

One of the environment properties often under the designer's control is how much information to make available to agents. Designers often limit the amount of information available to agents, because it enables more immersive and human-like behavior [14]. However, accomplishing this goal requires developing techniques for agents to operate in partially observable environments.

A problem encountered in imperfect information environments is tracking objects that were previously observed. The problem of target tracking occurs in RTS games as well as other genres such as first person shooters. There are two main approaches for estimating the position of a target which is not visible to an agent. In a space-based model, the map is represented as a graph and each vertex is assigned a probability that it contains the target. In a particle-based model, a cloud of particles represent a sampling of potential coordinates of the target [25]. Both approaches can apply a movement model for updating estimations of target locations.

Occupancy maps are a space-based model for tracking targets in a partially observable environment [42]. The map is broken up into a grid, where each node in the grid is connected to adjacent nodes. During each update there is a diffusion step

where each node transfers a portion of the probability it contains the target uniformly to adjacent nodes. The update cycle contains a visibility check where nodes visible to the agent not containing the target are assigned a weight of zero, and a normalization process that scales the weights of nodes. One of the challenges in applying occupancy maps is selecting a suitable grid resolution, because visibility computations can become prohibitively expensive on large grids.

Tozour presents a variation of occupancy maps which incorporate observations made by an agent [99]. The agent maintains a list of visited nodes, and searches for targets by exploring nodes that have not been investigated. The model can incorporate a decay, which will cause the agent to gradually investigate previously explored nodes. Hladky and Bulitko demonstrate how the accuracy of occupancy maps can be improved by applying movement models based on player behavior [37]. Rather than uniformly transfer probability to adjacent nodes, their approach uses hidden semi-Markov models (HSMM) to learn transitions between grid nodes based on previous observations.

Particle filters are an alternative method for tracking targets. This approach has a rich history in robotics, and has been applied to several problems including localization and entity tracking [97]. The application of particle filters to state estimation in games was first proposed by Bererton as a technique for creating agents that exhibit an illusion of intelligence when tracking targets [10]. Particle filters can be applied to target tracking by placing a cloud of particles at the target's last known position, where each particle performs a random walk and represents a potential target location. Each update cycle, the position of each particle is updated based on a movement model,

particles visible by the agent are removed from the list of candidate target locations, and the weights of the particles are normalized. One of the problems encountered in applying particle filters to RTS games is that there may be several indistinguishable units on the map, making the culling process non-trivial. My system uses a particle model inspired by particle filters, in which single particles are used to track targets and confidence in predictions decays over time.

2.3.2 Plan Recognition

Plan recognition is the task of inferring an agent's plans or goals based on observed actions [11]. Techniques for performing plan recognition in games include Bayesian models [5], case-based reasoning [28], and machine learning [89].

Albrecht et al. applied dynamic belief networks to predicting the player's current goal in a dungeon adventure game [5]. Their approach is based on Bayesian models [17], 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 [15].

Case-based plan recognition offers an instance-based approach. It has been applied to performing player modeling in Space Invaders [28]. Case-based plan recog-

nition 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 [19].

Machine learning has also been applied to plan recognition. Hsieh and Sun built a player model by analyzing StarCraft replays [40]. 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 [89]. 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.

2.4 Learning in Games

Performing well in RTS games involves learning a variety of different tasks. Two approaches used for building game AI with learning capabilities are online learning and learning from demonstration. Christian Thurau identifies three layers of behavior that can be learned in games: reactive behaviors are direct stimuli responses, tactical behaviors are intermediate-level actions, and strategic behaviors are actions performed

in pursuit of goals [98]. RTS games involve all three behavior levels, but my approach focuses on learning the reactive and strategic layers.

Due to the vast game space of RTS games, learning approaches have focused on individual aspects of gameplay. Reinforcement learning [63] and Monte Carlo [8] methods have shown promising results for tactical aspects in RTS games, while genetic algorithms [84] and case-based reasoning [3] have been applied to strategy selection. Techniques for strategy selection have relied on domain-specific knowledge representations, such as a state-lattice [83], to limit the types of strategies that are explored.

2.4.1 Learning from Demonstration

One way of bootstrapping the learning process in games is to use gameplay demonstrations. Demonstrations can be generated by individual players or large groups of players. One of the benefits of learning from groups of players is that it enables the development of robust intelligence, where agent behavior is built from *collective intelligence* rather than a small number of designers [79]. Learning from demonstration also presents several challenges, because gameplay demonstrations contain noisy, non-intentional actions and natural language, and lack annotation.

In a typical *learning by observation* configuration, an agent learns to perform a task solely from data collected through observation [29]. This is similar to *programming by demonstration*, in which an expert purposely demonstrations how to perform a task. Learning from demonstration is a generalization of learning by observation in which the players being observed do not need to purposely demonstrate how to accomplish a

specific task [75].

Learning from demonstration has been used to build systems which imitate player behavior and accomplish goals using a variety of different behaviors. The Drivatar system in *Forza Motorsport* learns how to drive using a collection of racing demonstrations from a player[1]. The Drivatar system uses gameplay demonstrations to train reactive-level actions for controlling a vehicle, as well as tactical-level actions which manage racing tactics such as overtaking and blocking. *The Restaurant Game* is an experiment that works towards the goal of building agents that exhibit convincing social behavior by collecting thousands of demonstrations from human players [78]. These demonstrations provide examples of how to perform specific tasks, and can be used to assist in the authoring of game agents. One of the main challenges in accomplishing this goal is inferring hierarchical task structure, which can be addressed by leveraging human-annotated demonstrations [80].

Darmok is a case-based planner that learns to play complete games of Wargus from expert demonstrations [74]. 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 [24]. 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

[1] `http://research.microsoft.com/en-us/projects/drivatar`

pursue. Behaviors selected for execution by Darmok can be customized by specifying a behavior tree for performing actions [81]. 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 [71]. The system uses a technique based on HTN-maker [39] 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 [110].

2.4.2 Case-Based Reasoning in RTS Games

Case-based reasoning (CBR) is a methodology for building systems that learn from experience [1]. CBR has been applied to solving several sub-problems in RTS games including micromanagement [96], tactics [63], and strategy selection [3]. There are also CBR systems aimed at building complete game playing agents [74].

Aha et al. were the first to apply CBR to building game AI for RTS games[3]. 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 [83]. 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. 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 [64].

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 War-Craft III [96]. 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 noticeably increase the utility of units in micromanagement tasks.

Bakkes et al. present a CBR approach for improving a pre-existing game AI for the RTS game *Spring* [7]. 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* [92]. 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 [63].

Baumgarten et al. present a hybrid CBR system that integrates simulated annealing, decision tree learning and case-based reasoning [9]. Their system is evaluated in the tactics-based RTS game *DEFCON: Everybody Dies*. It uses an initial case library generated from a 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.

2.5 Summary

Previous work has explored many of the capabilities necessary for expert RTS gameplay. Cognitive architectures work towards developing mechanisms that reproduce

human cognitive processes, while game AI techniques support the authoring of agents that operate in complex environments. A variety of learning approaches have been applied to games enabling agents to learn from experience and demonstrations, model the actions of opponents, and build estimations of game state. Generally, previous work has focused on subsets of RTS gameplay or simplifications of the state space. The main outstanding issues are integrating these diverse capabilities in a complete game-playing system, and evaluating against human opponents.

Chapter 3

StarCraft

StarCraft and its expansion, *StarCraft: Brood War* were released by Blizzard Entertainment™ in 1998. It is the successor to Blizzard Entertainment's RTS game *WarCraft II*, which has been studied by a number of researchers using the open source clone *Wargus* [3, 8, 56, 61, 62, 72, 84, 104]. StarCraft is a science fiction real-time strategy game in which players take the role of a military commander in a complex strategy simulation. A screenshot of a combat scenario in StarCraft is shown in Figure 3.1. It is one of the few video games played at a professional level, in part because gameplay is deep, fast-paced and unforgiving. StarCraft is an excellent domain for AI research, because gameplay requires many of the capabilities necessary for human-level AI and it is an environment in which human behavior can be observed, emulated, and evaluated.

In StarCraft, the player takes the role of a military commander and is assigned the objective of destroying all opponents. Achieving this objective requires performing a number of tasks including economy management, production, and tactics. This

Figure 3.1: StarCraft is a real-time strategy game in which the player takes the role of a military commander and pursues the objective of destroying all opponents.

thesis focuses on *1 vs. 1* battles, in which two players fight for control of a territory with equivalent starting conditions. This is the most common format for professional matches. StarCraft has three distinct races that players can command: *Protoss*, *Terran*, and *Zerg*. Each race has a unique technology graph (*tech tree*) which can be expanded to unlock new unit types, structures, and upgrades. Additionally, each race supports different styles of gameplay. Mastering a single race requires a great deal of practice, and experts in this domain focus on playing a single race.

There is a large community of players supporting StarCraft, which ranges from novices to professionals. StarCraft is played all over the world, which ensures that there is a large, active player base for evaluating agents versus human players that have a strong understanding of the game. StarCraft is played at a professional level in South Korea[1], and players spend substantial amounts of time training and developing gameplay skills. Professional gameplay involves performing over 300 actions per minute (APM) during peak gameplay and has been compared to grandmaster play in chess [61]. 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 [18].

This chapter provides an overview of StarCraft gameplay and the competencies involved, analysis of the task environment complexity, and discussion of the sources of

[1]Korea e-Sports Association: `http://www.e-sports.or.kr/`

domain knowledge available for building AI systems.

3.1 Gameplay

StarCraft gameplay involves performing tasks across multiple scales. These tasks include managing an economy in order to more rapidly bring in resources, expanding a technology tree to unlock new units and technologies, producing units to develop a combat force, and attacking opponents to destroy opponent forces and bases. An overview of the different tasks is shown in Figure 3.2. Gameplay involves coordinated decision making, because actions performed at each of these levels complement each other and work towards the high-level goal of defeating the opponent.

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

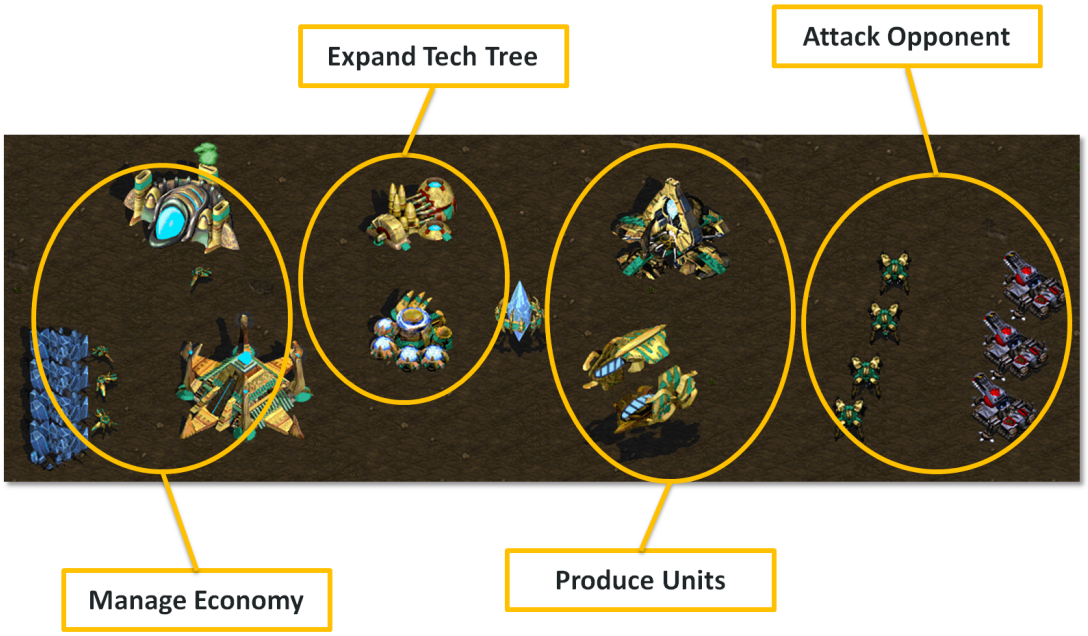

Figure 3.2: StarCraft gameplay involves managing several tasks in order to defeat opponents. These tasks include managing an economy, expanding a technology tree, producing units, and attacking opponents.

agement 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.

Expert gameplay requires specializing in several distinct competencies. Micromanagement is the task of controlling individual units in combat in order to maximize the utility of a unit. Terrain analysis is another area of competence which determines where to place units and structures to gain tactical advantages. Strategy selection is the task of determining which unit types to produce and upgrades to research, in order to counter opponent strategies. Attack timing is a competency for determining the ideal moment to launch an attack against the opponent.

3.1.1 Micromanagement

Expert players issue movement and attack commands to individual units to increase their effectiveness in combat. The motivation for performing these actions is to override the default low-level behavior of a unit. When a unit is given a command to attack a location, it will begin attacking the first enemy unit that comes into range. The default behavior can lead to ineffective target selection, because there is no coordination between different units.

Target selection and damage avoidance are two forms of micromanagement applied to StarCraft. To increase the effectiveness of units, a player will manually select the targets for units to acquire. This technique enables units to focus fire on specific targets, which reduces the enemy unit's damage output. Another use of target selection

is to select specific targets which are low in health or high-profile targets. Dancing is the process of commanding individual units to flee from battle while the squad remains engaged. Running away from enemy fire causes the opponent units to acquire new targets. Once the enemy units have acquired new targets, the fleeing unit is brought back into combat. Dancing increases the effectiveness of a squad, because damage is spread across multiple units, increasing the average lifespan of each unit.

Micromanagement is a highly reactive process, because it requires a large number of actions to be performed. During peak gameplay, expert players often perform over three hundred actions per minute [61]. Effective micromanagement requires a large amount of attention and focusing too much on micromanagement can be detrimental to other aspects of gameplay, since the overall advantage gained by micromanaging units is bounded. Due to the attention demands of micromanagement, it is more common earlier in the game when there are fewer tasks to manage and units to control.

3.1.2 Terrain Analysis

Expert players utilize terrain in a variety of ways to gain tactical advantages. These tasks include determining where to build structures to create bottlenecks for opponents, selecting locations for engaging enemy forces, and choosing routes for force movements. High ground plays an important role in StarCraft, because it provides two advantages: units on low ground do not have vision of units on high ground unless engaged, and units on low ground have a 70% chance of hitting targets on high ground. Players utilize high ground to increase the effectiveness of units when engaging and

retreating from enemy units.

Terrain analysis in RTS games can be modeled as a qualitative spatial reasoning task [32]. While several of the terrain analysis tasks such as base layout are deliberative, a portion of these tasks can be performed offline using static map analysis. This competency also requires the ability to manage dynamic changes in terrain caused by unit creation and destruction.

3.1.3 Strategy Selection

Strategy selection is a competency for determining the order in which to expand the tech tree, produce units, and research upgrades. Expert players approach this task by developing build orders, which are sequences of actions to execute during the opening stage of a game. As a game unfolds, it is necessary to adapt the initial strategy to counter opponent strategies. In order to determine which strategy an opponent has selected, it is necessary to actively scout to determine which unit types and upgrades are being produced. Players also incorporate predictions of an opponent's strategy during the strategy selection process. Predictions can be built by studying the player's gameplay history and analyzing strategies favored on the selected map.

Strategy selection is a deliberative process, because it involves selecting sequences of actions to achieve a particular goal. Strategy planning can be performed both offline and online. Build order selection is an offline process performed before the start of a game, informed by the specific map and anticipated opponent strategies, while developing a counter strategy based on the opponent strategy during a game is

an online task.

3.1.4 Attack Timing

Attack timing is a critical aspect of StarCraft gameplay, because there is a large commitment involved. There are several conditions that are used for triggering an attack. A timing attack is a planned attack based on a selected build order. Timing attacks are used to take advantage of the selected strategy, such as attacking as soon as an upgrade completes. An attack can also be triggered by observing an opportunity to damage the opponent such as scouting units out of place or an undefended base. There are several other conditions which are used to trigger attacks, such as the need to place pressure on the opponent.

The attack timing competency is deliberative and reactive. Timing attacks can be planned offline and incorporated into the strategy selection process. The online task of determining when to attack based on encountered game situations is complex, because a player has only partial observability.

3.2 Domain Properties

StarCraft is a complex domain with many real-world properties. It presents players with a large decision complexity, partially observable environment, hundreds of units to manage that can have a variety of different actions, and adversarial players. This section focuses on the environment complexity of StarCraft, while the task complexity of StarCraft gameplay is discussed in Chapter 4.

Table 3.1: A classification of three domains in terms of Russell and Norvig's task environment properties [88] shows that many of the properties of StarCraft are also present in real-world tasks, such as taxi driving.

	Chess	**StarCraft**	**Taxi Driving**
Fully vs. Partially Observable	Fully	Partially	Partially
Deterministic vs. Stochastic	Deterministic (Strategic)	Deterministic (Strategic)	Stochastic
Episodic vs. Sequential	Sequential	Sequential	Sequential
Static vs. Dynamic	Static	Dynamic	Dynamic
Discrete vs. Continuous	Discrete	Continuous	Continuous
Single vs. Multiagent	Multiagent (Competitive)	Multiagent (Competitive)	Multiagent

An overview of StarCraft in terms of Russell and Norvig's task environment properties [88] is shown in Table 3.1. StarCraft is a partially observable, deterministic, sequential, dynamic, continuous, and multiagent environment. While StarCraft's state space is finite, the task is real-time and therefore a continuous-time problem. StarCraft also has random elements, such as projectiles having a 70% hit rate from low to high ground, but these outcomes are managed by a seeded random number generator which produces a deterministic output. However, this process is hidden from players and it can be argued that the domain is stochastic, because during gameplay it is unknown whether a projectile will damage a target. I am classifying the domain as deterministic, because this is a minor aspect of gameplay and assigning the same actions will always result in the same outcome. The main difference between StarCraft and a real-world task, such as taxi driving, is that StarCraft is a deterministic simulation.

One of the key differences between RTS games and traditional board games such as chess is the state-space complexity. In StarCraft there can be hundreds of units in play across a large map, with several attributes including unit type, health, energy,

heading, and acceleration. Considering only unit types and positions, the state-space complexity of StarCraft can be estimated as follows:

$$O((TXY)^U)$$

U – number of units in play

T – number of unit types in StarCraft

X – number of horizontal map tiles

Y – number of vertical map tiles

StarCraft supports up to 1700 units in play, has over 100 units types, and allows maps with maximum dimensions of 256 by 256 tiles. This formulation results in a complexity of $(100*256*256)^{1700}$, or roughly $10^{11,500}$. While this estimation allows for illegal states, such as overlapping units, the actual state-space of StarCraft is much larger, because units have many more attributes. This complexity is vastly larger than chess, which Shannon estimated as 10^{43} [91].

3.2.1 Decision Complexity

A formal analysis of the decision complexity of RTS games was first proposed by Aha et al. [3]. They estimated the decision space of Wargus, which is the set of possible actions that can be executed at a particular moment, as follows:

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

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:

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

Given this formulation, the decision complexity is still large. For example, a game on a 256x256 tile map with 50 worker units results in more than $1,000,000$ possible actions. This is vastly different in scale than the decision complexity of chess, which is approximately 30 [3].

There are several 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 much smaller.

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. In StarCraft, there is no dominant strategy and there is a much larger strategy space. 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.

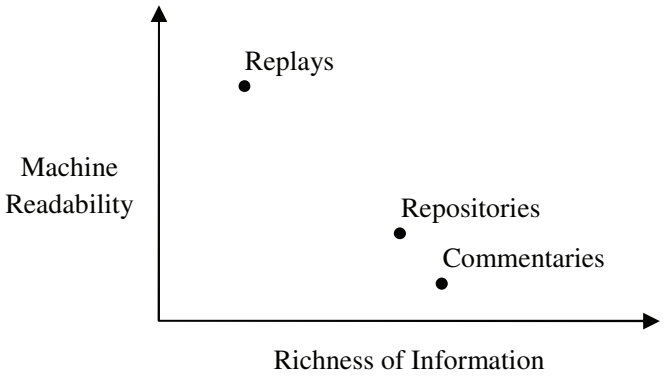

Figure 3.3: There are several sources of domain knowledge for StarCraft including replay files, user-created repositories of tactics and strategies, and commentaries of professional matches.

3.3 Knowledge Sources

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 data 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 a game. While replays are easy to parse, the information that can be extracted is limited to primitive game actions. Additional discussion of learning from StarCraft replays is presented in Chapter 5.

Commentaries are another form of information for describing RTS gameplay. In a StarCraft commentary, a commentator provides insight inferring and 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

Table 3.2: A user-provided summary for the 10/15 gates build order listed on Liquipedia. The left column describes when to perform build-order actions, while the right column specifies the build-order action to perform.

Supply	**Action**
8/9	Pylon
10/17	Gate
11/17	Assimilator
13/17	Cybernetics Core
15/17	Gateway (Cut Probes)
15/17	Dragoon Range
15/17	Dragoon
17/17	Pylon
17/25	2 Dragoons
21/25	Pylon

of information, such as Liquipedia[2]. It is less difficult to extract domain knowledge from repositories than commentaries, because the information is semi-structured. The trade-off between richness of information and machine readability of these different knowledge sources is shown in Figure 3.3.

One of the ways players learn to play StarCraft is by studying replays and extracting build orders. A build order is an abstraction of an opening strategy that players can use for communicating different strategies. An example build-order summary for the *10/15 gates* strategy listed on Liquipedia is shown in Table 3.2. The summary is a partially-specified plan that lists when to perform specific build-order actions in the opening phase of the game. The summary also includes a list of build orders this opening is strong against, such as fast expanding Terran opponents, as well as build orders the opening is countered by, such as an early two factory opening.

[2] `http://wiki.teamliquid.net/starcraft`

There are several actions that need to be performed in order to execute build orders that are not specified in summaries. For these actions, it is assumed that the player will follow rules of thumb for decision making tasks such as worker production. In this example, the player should continue producing worker units until the second gateway is produced, which is when the plan specifies a "cut probe" action. Additional actions often omitted from build orders include information about when to scout and how to transition to later strategies. In order to develop a system capable of utilizing these knowledge sources, it is necessary to encode these rules of thumb in the AI system [61].

3.4 Summary

StarCraft is a real-time strategy game in which the player takes the role of a commander in a military scenario. Gameplay involves managing actions across several different scales as well as mastering several competencies including micromanagement, strategy selection, and attack timing. In addition to presenting players with complex gameplay mechanics, the domain presents several real-world properties to manage including an enormous decision complexity and partial observability. As a task environment, StarCraft presents many of the challenges that need to be addressed in order to build AI systems for real-world tasks. One of the ways in which these problems can be addressed is by leveraging domain knowledge from sources including replays, repositories, and commentaries. Utilizing this information is difficult, because strategy

summaries such as build orders are under specified and rely on players following rules of thumb. An AI system which plays StarCraft at an expert level needs to encode these rules of thumb in order to harness these knowledge sources and learn from gameplay demonstrations.

Chapter 4

Multi-Scale AI

One of the capabilities necessary for expert RTS gameplay is the ability to simultaneously manage several interconnected tasks in pursuit of higher-level goals. In StarCraft, focusing too much attention on a specific task is detrimental to overall gameplay, because opponents may gain advantages in other aspects of gameplay. One way players approach this problem is by satisficing [93], following rules of thumb which provide adequate policies for a subset of gameplay while focusing attention on tasks that require immediate attention.

A key characteristic of RTS games is that they require concurrent and coordinated goal pursuit across multiple scales of reasoning. While it is possible to decompose gameplay into distinct competencies, there is not a clear hierarchical structure, due to cross-cutting concerns. RTS games present a multi-task problem in which there is not a strict separation across tasks, and performance in each task impacts other tasks. Additionally, RTS gameplay involves managing units that work towards individual,

squad-level, and global goals. I define AI problems that exhibit these characteristics as multi-scale AI problems.

I use the term *scale* to refer to an agent that reasons about the actions of actors in the environment at different levels of detail. This situation occurs in RTS games, because a single agent controls a large number of units, and it is necessary to reason about both individual units as well as groups of units. The use of the term scale in this context refers to decision making across different granularities, or structural scales, and is distinct from temporal scales or spatial scales.

This chapter introduces the class of multi-scale AI problems and presents three examples. In order to build agents for this class of problems, I advocate reactive planning, which provides many of the capabilities necessary for multi-scale reasoning. I then discuss EISBot, which is built using the ABL reactive planning language, and present idioms used to assist in the agent authoring process.

4.1 Definition

A multi-task domain is an environment in which an agent performs two or more separate tasks [90]. A *multi-scale* domain is an instance of a multi-task domain in which the following conditions are met:

1. **Multiple Structural Scales:** Actions are performed across multiple levels of coordination.

2. **Interrelated Tasks:** There is not a strict separation across tasks and the performance in each task impacts other tasks.

3. **Real-time:** Actions are performed in real-time.

Multi-scale domains present several challenges for building AI systems, because they have many real-world properties.

The first characteristic of multi-scale domains is that they require agents to perform decision making across multiple scales, where a scale is a level of coordination. In environments in which an AI system controls multiple characters or units, decision making can be performed for individual units or groups of units. Performing well in a multi-scale domain involves pursuing goals at multiple levels of coordination. RTS games involve decision making across individual and squad scales, and actions at the squad scale are performed at a higher level of detail than actions at the individual scale. Effective RTS gameplay involves pursuing goals at different levels of coordination based on the current situation.

A second characteristic of multi-scale domains is that the tasks are interrelated. This situation arises when there are dependencies or shared resources across tasks, or there are not clear boundaries between tasks. One of the challenges in building an agent for a multi-scale domain is that it is not possible to optimize for each problem separately, due to interactions between different tasks. In StarCraft, worker units can be utilized for a variety of purposes including resource collection, scouting, and defense, which means that building an optimal policy for resource gathering must also factor in

Figure 4.1: The autonomous characters Grace and Trip in the interactive drama Façade [60] manage behaviors across several different scales to create a dramatic experience.

other aspects such as base defense. Interrelated tasks are present in domains in which resources are shared across multiple tasks and there are cross-cutting concerns between aspects of gameplay.

The third characteristic of multi-scale domains is that they are real-time environments. Due to this property, agents that operate in multi-scale domains must perform decision making and act in real-time. This environmental property also limits the amount of time an agent can spend deliberating on an individual task, because several tasks need to be managed concurrently.

Many computer games are multi-task, but involve decision making at a single scale. For example, first-person shooter (FPS) games may involve pursuing objectives and managing weapon inventory. An AI system that controls a single bot in an FPS game performs decision making at a single scale. If the system also coordinates with

other teammates or bots, then the system is performing multi-scale reasoning with individual and group scales. Instances of games with multi-scale problems include StarCraft and the interactive drama Façade. Multi-scale can also be used to classify real-world domains, such as RoboCup Soccer.

4.1.1 Façade

Façade is an interactive drama developed by Michael Mateas and Andrew Stern [60]. It is an experiment in Expressive AI in which the player is presented with a dramatically engaging experience [58]. The player interacts with the autonomous characters Grace and Trip, shown in Figure 4.1. The system includes a drama manager for handling narrative progression, a natural language understanding component for processing player responses, and reactive planners for implementing the behaviors of Grace and Trip.

Each of the autonomous characters in Façade presents a multi-scale AI problem. In addition to managing pathfinding and animation goals, the characters interact in group behaviors and perform dramatic actions that work towards beat-level goals, such as posing a dramatic question to the player or waiting for the player to respond to a situation [60]. The autonomous characters Grace and Trip perform actions across the following scales:

1. **Individual:** The characters pursue pathfinding goals and perform a wide range of gestures and facial expressions.

2. **Joint Interactions:** Grace and Trip engage in group behaviors with each other and the player.

3. **Beat Progression:** Grace and Trip perform actions in pursuit of beat-level goals, which are selected by a beat manager.

In order for the characters to exhibit believable behavior, actions performed at each scale need to be coordinated with actions at other scales.

The scales in the autonomous character Grace are interrelated, because there is not a clear separation between tasks and performing actions at one scale may impact other scales as well. In Façade, the distance Grace maintains from the player is driven by the movement scale, but also plays a role in joint interactions with Trip. While it is possible to consider each scale separately, in order to create convincing performances, it is necessary for scales to coordinate which dramatic goals are being pursued. The autonomous characters in Façade present multi-scale domains in which animations, dialog, and timing need to be managed concurrently across multiple scales in order to create engaging performances.

4.1.2 RoboCup Soccer

The aim of the Robot Soccer World Cup is to create autonomous humanoid robots capable of winning against the FIFA World Cup champions[1]. Building robots capable of playing soccer with humans presents several challenges, because robots must sense and act in the real world. Another challenge in creating human-level robots for

[1] `http://www.robocup.org`

soccer is that gameplay presents a multi-scale problem. A soccer-playing agent performs actions across the following scales:

1. **Individual:** Soccer involves second-to-second actions performed in response to the current state. These are actions that can be performed by a single agent, such as defending against a specific player.

2. **Group:** Tactical actions, such as passing to a teammate, are joint interactions involving communication between players.

3. **Team:** Strategic actions, such as utilizing more aggressive formations, are global goals pursued by a team and often managed by external agents.

Each of these scales work towards the high-level goal of winning the game. The tasks in soccer gameplay are interrelated, because individuals need to coordinate in order to accomplish group or global goals. If an agent focuses too much attention on individual goals, it can be detrimental to the team. In soccer, players are a shared resource across different scales and effective gameplay requires concurrent and coordinated pursuit of individual and group goals.

A RoboCup soccer team can be authored as a single multi-scale agent, or as a multi-agent system. In a multi-agent system, each player performs decentralized decision making, and each of the players is at least partially autonomous [113]. In this configuration, each player is managed by a separate agent, and the system may also include a coach agent that coordinates team goals. In a multi-scale agent, all of the players are managed by a single monolithic agent. In this configuration, the multi-

scale agent performs all decision making and players have no autonomy. While both approaches may be applicable to a particular problem, a multi-scale approach is useful for environments in which the actors being managed require minimal autonomy.

4.1.3 StarCraft

Building an agent for StarCraft is a multi-scale AI problem, because gameplay involves low-level tactical decisions that must complement high-level strategic reasoning. At the micromanagement level, individual units are meticulously controlled in combat scenarios to maximize their effectiveness, while at the macromanagement level, players work towards long-term goals, such as building a strong economy and developing strategies to counter opponents. StarCraft gameplay requires performing actions across the following scales:

1. **Individual:** Units pursue actions that accomplish individual orders. This scale also involves micromanagement of individual units in combat.

2. **Squad:** Units can be assigned into control groups and given squad-level orders. Managing units as squads reduces the number of commands that need to be issued.

3. **Global:** High-level goals, such as committing to an aggressive push against an opponent, involve the majority of a player's units.

Gameplay at each of these scales is necessary in order to achieve the objective of destroying all opponent forces.

StarCraft gameplay involves managing a number of interrelated tasks, because there are cross-cutting concerns and shared-resources across different scales. Units can be used for several purposes in StarCraft. For example, worker units can be used for resource gathering, scouting, and defense. While performing these tasks, the unit may be pursuing individual goals such as self preservation, group goals such as absorbing damage, or global goals such as distracting an opponent army. StarCraft presents a multi-scale domain, because it requires managing several highly-coupled resources and tasks.

4.2 Reactive Planning

Reactive planning is well suited for building agents that operate in multi-scale domains, because it provides mechanisms for concurrent and coordinated goal pursuit across different granularities. It provides an excellent framework for authoring agents for complex, real-time environments, because the process for behavior selection is efficient, and it supports acting on partial plans while pursuing goal-directed tasks. My system utilizes reactive planning in order to emulate a subset of the capabilities demonstrated by human players. Additionally, my choice in using reactive planning to implement agent behavior is motivated by previous work that has used the ABL reactive planning language to author agents for the multi-scale domains Wargus [61] and Facade [60].

One of the challenges in authoring multi-scale agents is managing behaviors across different scales. A common approach to this problem is to abstract the different

scales into separate layers and build interfaces between the layers. Layered approaches raise difficulties for multi-scale problems, because layers may compete for access to shared resources, resulting in complicated inter-layer behaviors that break abstraction boundaries. One of the benefits of reactive planners is that they provide a unified architecture for authoring agents and provide mechanisms for reasoning across different scales, including capabilities for message passing and spawning new goals to pursue. These capabilities can be used to build agents that simultaneously manage behaviors across different scales.

A disadvantage of reactive planning is that authoring agents requires substantial domain engineering. This process presents an enormous authorial burden for strategic domains, because it is necessary to specify behaviors to anticipate all events [68]. Unlike cognitive architectures such as ICARUS [51] which use means-end analysis to handle failures that arise, reactive planners fail if there are no behaviors which can accomplish an instantiated goal. In order to overcome these limitations of reactive planning, an agent can integrate external components that support deliberative reasoning capabilities and learning. Examples of integrating external components with a reactive planning agent are presented in Chapter 6.

My system is implemented in the ABL reactive planning language. The following sections provide an overview of the language and semantics. Idioms for authoring ABL agents which operate in multi-scale domains are presented in Section 4.4.

4.2.1 A Behavior Language

A Behavior Language (ABL) is a reactive planning language based on the believable agent language HAP [54] and adds significant features to the original Hap semantics [58]. These include first-class support for meta-behaviors, which manipulate the runtime state of other behaviors, and joint intentions across teams of multiple agents [59]. ABL supports building multi-scale game AI, because it enables agents to pursue multiple goals concurrently and provides mechanisms for facilitating communication between behaviors.

ABL is a reactive planning language in which an agent has an active set of goals to achieve. Agents achieve goals by selecting and executing behaviors from a hand-authored behavior library. A behavior contains a set of zero or more preconditions which specify whether it can be executed given the current world state. There is also a specificity associated with behaviors that assigns a priority, and behaviors with higher specificities are selected for execution before considering lower specificity behaviors. The language supports a number of step modifiers which can be used to persistently pursue goals, continue operating when failures occur, and execute until a goal succeeds.

During the execution of an ABL agent, all of the goals it is pursuing are stored in the active behavior tree (ABT). Each execution cycle, the planner selects from the open leaf nodes and begins executing the selected node. A leaf node is a behavior that pursues a goal and consists of component steps. Component steps can be scripted actions, small computations, or additional goals. When a node is selected, its component

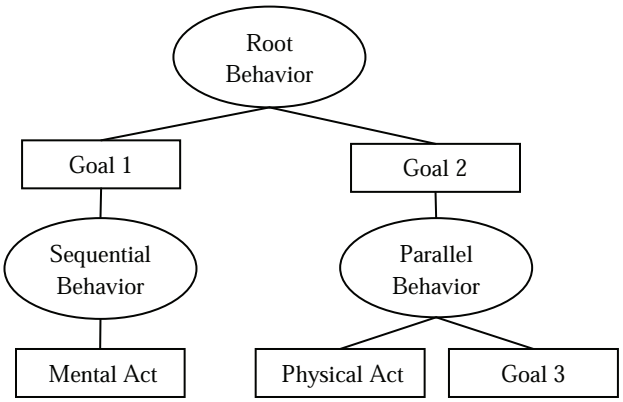

Figure 4.2: An example active behavior tree (ABT) in an ABL agent. The current expansion of the tree shows that `Goal 1` is being accomplished by a sequential behavior performing a mental act, while `Goal 2` is being pursued by simultaneously performing a physical act in the environment and subgoaling `Goal 3`.

steps are placed in the ABT as the children of the behavior. An example ABT is shown in Figure 4.2.

A core component of ABL agents is working memory. ABL's working memory serves as a blackboard for maintaining an agent's view of the world state as well as the current expansion of the active behavior tree. The agent's working memory is maintained through the use of sensors, which add, update and remove working memory elements (WMEs) from ABL's working memory. An agent's working memory can also be modified by the agent at runtime or by an external component.

One of the benefits of using ABL to author game AI is that scheduling of actions and plan monitoring is handled by the architecture. Component steps that contain physical acts begin execution as soon as they are selected from the ABT. Physical acts in ABL can take several game frames to perform. While executing a physical act, the step associated with the act is marked as executing, blocking steps after the physical

act in an enclosing sequential behavior until the physical act completes, while steps that are part of parallel behaviors in the ABT continue. Therefore, unlike behavior tree implementations or deliberative planners, a separate scheduling component is not necessary for scheduling the actions selected by an ABL agent.

ABL provides a mechanism for handling failures in the execution environment. If the behavior chosen for a goal fails for any reason, the behavior and component steps are removed from the ABT and the goal is again available for selection. This results in the agent selecting from different behaviors to achieve the goal. ABL differs from traditional planning in that behaviors selected for expansion may change the state of the execution environment and then be aborted. ABL does not require an explicit proposition to be true for a behavior to succeed. This feature was chosen, because it is difficult to create a complete and accurate domain model of the world. One of the results of this feature is that ABL agents do not require a commitment to a formal domain model [54].

4.2.2 ABL Semantics

ABL agents are written by authoring a collection of behaviors. Behaviors can perform mental acts, execute physical acts in the game world, bind parameters, and add new subgoals to the active behavior tree. Tasks are represented by behaviors in ABL: each behavior contains actions or additional subgoals that work to accomplish some goal. However, there may be multiple behavior rules with the same name that represent multiple means of achieving a particular goal. Thus the name of a behavior is the goal

```
initial_tree {
  subgoal sayHello();
}

sequential behavior sayHello() {
  act consoleOut("Hello World");
}
```

Figure 4.3: A "Hello World" example in ABL where the agent has a single goal of saying hello.

```
sequential behavior attackEnemy() {
  precondition {
    (PlayerUnitWME type==Marine ID::unitID)
    (EnemyUnitWME ID::enemyID)
  }

  act attackUnit(unitID, enemyID);
}
```

Figure 4.4: Behaviors in ABL can include zero of more preconditions, which can also be used to bind behavior-scoped variables.

which it accomplishes, while the contents represent the actual means of achieving that goal.

An example agent with the goal of `sayHello` is shown in Figure 4.3. The agent begins executing the root behavior, defined as initial_tree, which adds the subgoal `sayHello` to the active behavior tree. The agent then selects from the behaviors named `sayHello` to pursue the goal. In this example, the agent will select the behavior `sayHello`, resulting in the execution of a physical act that prints to the console.

ABL behaviors can be sequential or parallel. When a behavior is selected for expansion, its component steps are added to the ABT. For sequential behaviors the steps are executed serially: steps are available for expansion once the previous step has completed. For parallel behaviors, the steps can be expanded concurrently.

Behaviors can include a set of preconditions which specify whether the behavior can be selected. Preconditions evaluate boolean queries about the agent's working memory. If all of the precondition checks evaluate to true, the behavior can be selected for expansion. An example behavior with a precondition check is shown in Figure 4.4. The behavior checks that there is an agent-controlled unit with the type Marine and that there is an enemy unit. The first precondition test is performed by retrieving unit working memory elements from working memory and testing the condition `type==Marine`. The example also demonstrates variable binding in a precondition test. The unit's ID attribute is bound to the `unitID` variable and used in the physical act. The second precondition test retrieves the first enemy unit from working memory and binds the ID to `enemyID`.

An ABL agent can have several behaviors that achieve a specific goal. An optional specificity can be assigned to behaviors to prioritize selection. Behaviors with higher specificities are evaluated before lower specificity actions, but otherwise identically-named behaviors (different ways of accomplishing a specific goal) are selected randomly. This enables authoring of agents that have a prioritized set of behaviors to pursue a goal.

Behaviors may also be parameterized. When a parameterized behavior is expanded, it must be given a parameter as an argument. The contents of the behavior can then reference this argument. This allows for the same behavior to be instantiated multiple times. For example, an attack behavior could be instantiated individually for many different units, and could then order each unit to attack based on that unit's

```
sequential behavior initializeAgent() {
  spawngoal incomeManager();
  mental_act {
    System.out.println("Started manager");
  }
}

sequential behavior incomeManager() {
  with (persistent) subgoal mineMinerals();
}
```

Figure 4.5: The spawngoal keyword is used to spawn a new thread of execution and the persistent keyword enables an agent to continuously pursue a goal.

health. Behavior parameterization is a powerful tool for re-using behaviors across multiple contexts.

Behaviors can perform mental and physical acts. Mental acts are small chunks of computational work and are written in Java. Mental acts can be used to add and remove WMEs from working memory. An example mental act is shown in Figure 4.5. Physical acts are actual actions performed by the agent in the game. Physical acts can be instant or have duration. Physical acts are performed in a separate thread from the decision cycle and do not block the execution of the ABT. They are removed from the ABT once completed.

ABL provides several features for managing the expansion of the active behavior tree. The spawngoal keyword enables an agent to add new goals to the active behavior tree at runtime. The spawned goal is then pursued concurrently with the current goal. The persistent keyword can be used to have an agent continuously pursue a goal. The use of these keywords is demonstrated in the example in Figure 4.5. Upon execution, the `initializeAgent` behavior adds the goal `incomeManager` to the active

behavior tree and then executes the mental act. The persistent modifier is used to force the agent to continuously pursue the `mineMinerals` goal. Note that if subgoal was used instead of spawngoal in the example, the mental act would never get executed.

Behaviors can optionally include success tests and context conditions. A success test is an explicit method for recognizing when a goal has been achieved [54], whereas a context condition provides an explicit declaration of conditions under which a goal is relevant. If a success test evaluates to true, then the associated behavior is aborted and immediately succeeds. Conversely, if a context condition evaluates to false, the associated behavior fails and is removed from the ABT. An example showing success tests and context conditions is shown in Figure 4.6. The behavior binds the current time to the `startTime` variable. The context condition checks that no more than 10 seconds have passed since starting the execution of the behavior. The success test checks if the agent possesses a Marine. When combined with the `wait` subgoal, success tests suspend the execution of a behavior until the test conditions evaluate to true. In the example, the behavior will either return success as soon as the agent has a Marine, or return failure after 10 seconds have passed.

4.3 Agent Design

EISBot[2] is an integrated agent for playing complete games for StarCraft. The core of the agent is implemented in ABL, which interfaces with the game environment,

[2]EISBot is short for the Expressive Intelligence Studio Bot, and pronounced "Ice Bot". The source code for EISBot is available online: `http://code.google.com/p/eisbot/`

```
sequential behavior waitForMarine() {
  precondition { (TimeWME time::startTime) }
  context_condition {
    (TimeWME time < startTime + 10)
  }

  with success_test {
    ((PlayerUnitWME type==Marine)
  } wait;
}
```

Figure 4.6: Behaviors can include success tests that specify conditions for success and context conditions that must remain true throughout the duration of a behavior.

manages the agent's active goals, and handles action execution. While reactive planning does not directly support the estimation, adaptation, or anticipation capabilities necessary for expert-level RTS gameplay, it does provide an excellent framework for authoring agent behavior and serves as the glue for integrating external components with learning and adaptation capabilities. This section focuses on the development of a collection of base behaviors for EISBot, which enable the agent to operate in a multi-scale domain, but limits EISBot to a fixed set of strategies. In Chapter 6, I describe modifications to the agent that enable new strategies to be learned from demonstrations.

EISBot is based on McCoy and Mateas's integrated agent framework [61], and extends it in a number of ways. While the initial agent was applied to the task of playing Wargus, I have transferred many of the concepts to StarCraft. Additionally, I implemented several of the competencies that were previously identified, but were stubbed out in the original agent implementation, such as reconnaissance. I have also interfaced the architecture with external components as detailed in Chapter 6.

To handle the multiple scales necessary for StarCraft, the agent is based on

a conceptual partitioning of gameplay into distinct scales. While the problem of RTS gameplay is non-hierarchical due to cross-cutting concerns and shared resources, it is possible to decompose gameplay into subproblems, as long as there are mechanisms for supporting coordination across multiple scales. EISBot is composed of managers, each of which is responsible for a specific aspect of gameplay. The managers pursue goals in parallel and are authored using multi-scale idioms which enable the agent to handle cross-cutting concerns such as resource contention.

A manager in EISBot is implemented as a collection of ABL behaviors. Each manager is responsible for handling a subtask of gameplay, and facilitating communication with other managers. One of the benefits of partitioning decision making in the agent into separate managers is that specialized behaviors can be authored for subproblems in the agent, such as micromanagement, which complement other scales of reasoning. This design provides an authoring environment which supports specification of rule-of-thumb behaviors as well as highly-specialized behaviors based on domain knowledge [61].

EISBot is composed of several managers, as shown in Figure 4.7, which are based on a decomposition of gameplay into distinct competencies:

- **Strategy Manager:** Selects a build order and handles high-level decisions. Tasks include determining when to attack opponents, research unit upgrades, and establish additional bases. Most decisions are enacted by requesting other managers to perform actions.

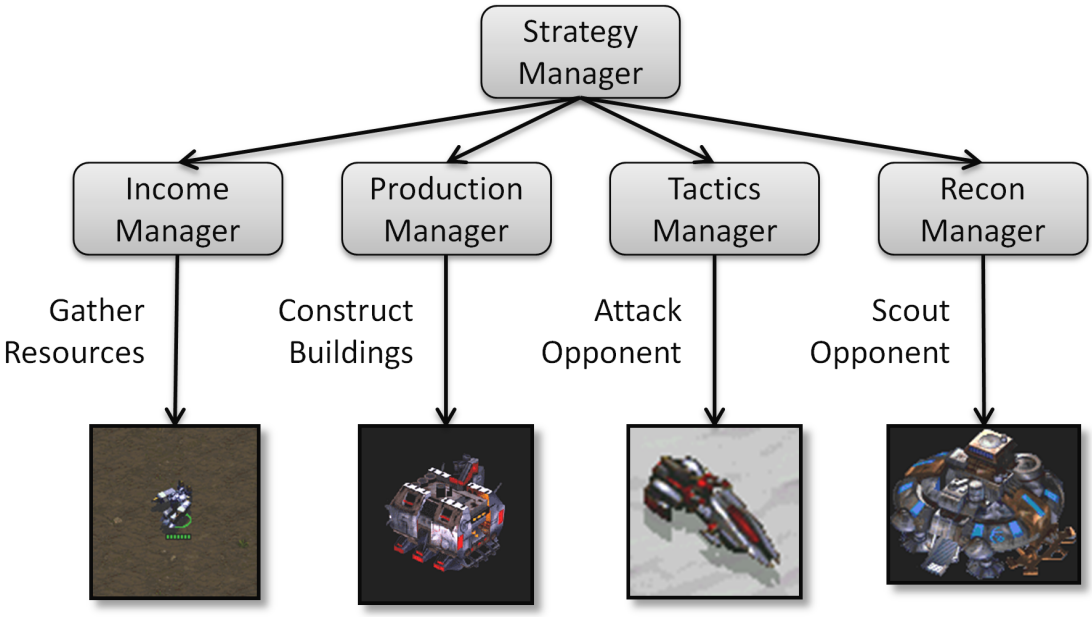

Figure 4.7: EISBot is composed of managers, each of which is responsible for decision making and action execution for a gameplay competency.

- **Income Manager:** Responsible for worker units and resource gathering. Tasks include producing additional worker units, processing requests for expanding, assigning idle workers to gather, producing facilities for collecting vespene gas, and selecting how many workers to gather minerals and gas.

- **Production Manager:** Performs actions for processing construction, training, and research requests. Tasks include constructing new structures and monitoring construction progress, training new units and building additional production facilities, researching upgrades at tech structures, producing additional supply sources, and spending excess resources.

- **Tactics Manager:** Responsible for handling combat units and engaging enemy

units. Tasks include rallying units and defending bases, assigning units to squads, processing requests to initiate attacks, regrouping units in a squad, assigning squads attack and retreat commands, moving support units, and casting abilities and spells.

- **Reconnaissance Manager:** Performs scouting behaviors using worker and combat units. Tasks include requesting worker units for scouting, assigning move commands to scouting workers, recording the positions of enemy units and structures, and assigning wander commands to combat units if there are no estimates of opponent locations.

While some of the tasks performed by the managers can be performed in isolation, most tasks in EISBot rely on other managers in order to operate.

Each of the managers implements an interface for communicating with other managers in the system. To facilitate message passing between different managers, the agent uses ABL's working memory as a blackboard [36]. The interfaces define how working memory elements are posted to and consumed from working memory. In EISBot, the managers communicate by requesting different managers to perform actions, and each manager is expected to process specific request types as well as generate request types. For example, the production manager is responsible for processing `ConstructionRequestWME` objects, which are generated by the strategy manager and specify a building type to construct. In order to satisfy this request, the construction manager generates a `WorkerRequestWME` that is passed to the income manager in order

to free up a worker unit for the construction task.

This approach for building agents enables a modular design with two main benefits. First, new implementations for managers can be swapped into the agent, as long as the new implementation satisfies the manager's interface contract [61]. This is one of the ways in which EISBot integrates external reasoning components. Second, this approach to structuring agents supports the decomposition of tasks. In EISBot, the decision process for selecting which structures to construct is decoupled from the construction process, which involves selecting a construction site and monitoring execution of the task. This decoupling enables the strategy manager to select which structures to build independent of the process of actually constructing the structure.

A side effect of this agent design is that the components in the agent are tightly coupled. Each of the managers depends on other managers and removing any single manager from the system causes the agent to fail. McCoy and Mateas refer to this problem as manager interdependence and show that each manager is necessary for the agent to operate [61]. This side effect is present in my system, because performing multi-scale reasoning for StarCraft requires coordination across many tasks.

4.3.1 Agent Interface

One of the goals of the agent interface is to resemble the interface provided to human players as closely as possible. Specifically, the agent should not be allowed to utilize any game state information not available to a human player, such as the locations of non-visible units, and the set of actions provided to the agent should be identical to

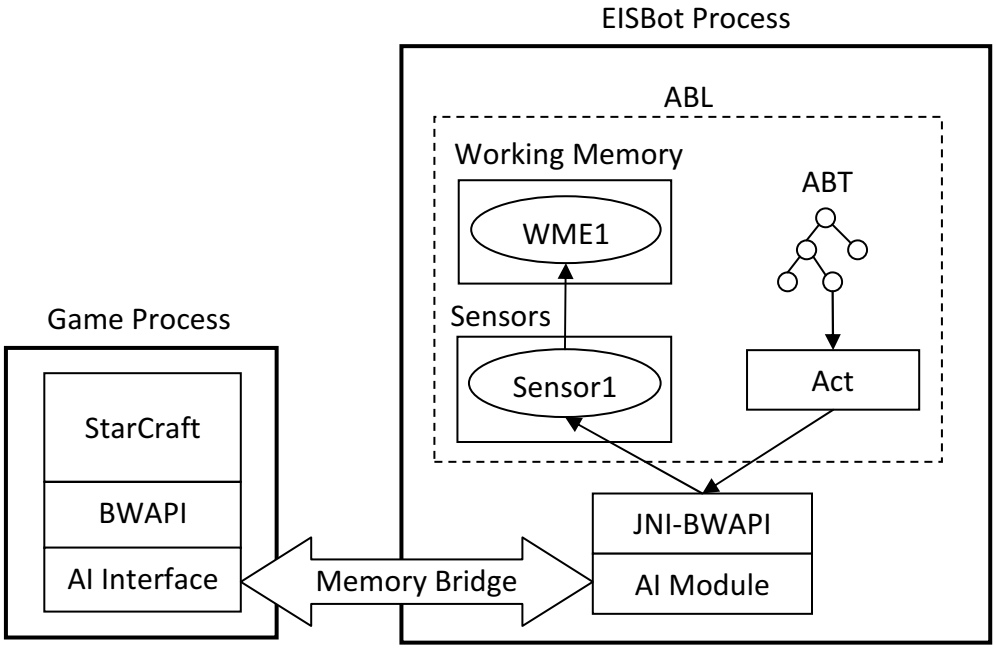

Figure 4.8: The game process exposes an interface for authoring agents by injecting the Brood War API into the StarCraft runtime. EISBot is an ABL agent supported by a Java runtime that communicates with the game process through a shared memory bridge.

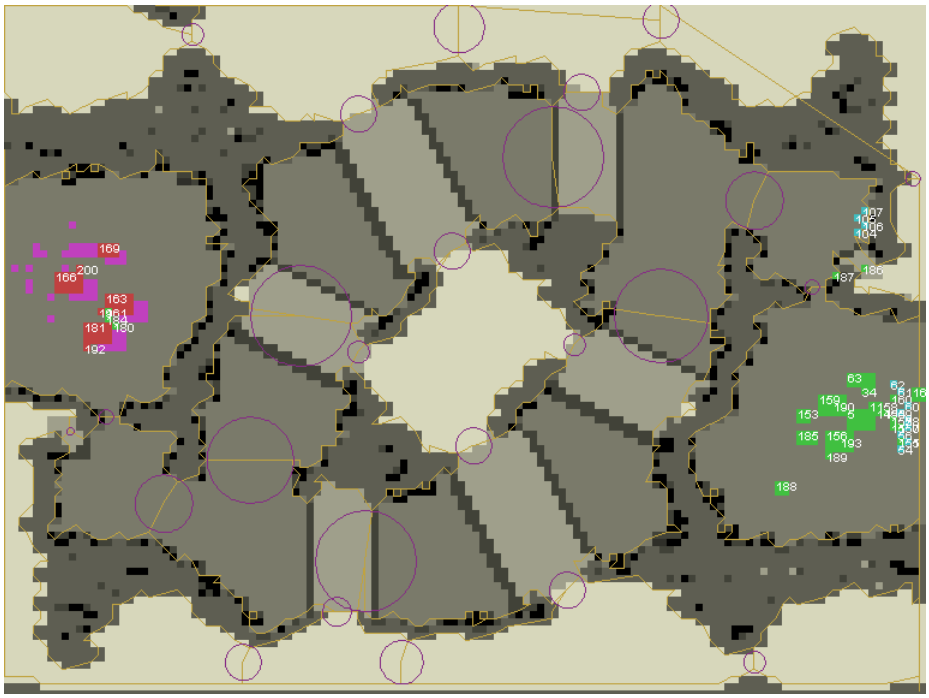

Figure 4.9: EISBot's perception of the world state includes map and unit data, and is managed by a collection of sensors that update world memory during each game update.

those provided through the game's user interface. Enforcing this constraint ensures that an agent which performs well in this domain is capable of the anticipation and estimation capabilities necessary for expert gameplay.

EISBot interfaces with StarCraft through the use of sensors that perceive game state and actuators that enable the agent to send commands to units. An overview of the components in the EISBot-StarCraft interface are shown in Figure 4.8. The game process launches StarCraft using a third party tool, which enables additional dynamically linked libraries (DLLs) to be injected into the process space. The launcher injects the Brood War API[3] (BWAPI), which is a C++ library that exposes an AI

[3] `http://code.google.com/p/bwapi/`

interface for authoring StarCraft agents.

ABL agents are compiled to Java and executed by the ABL runtime, which is also written in Java. To communicate with the game process, EISBot uses the JNI-BWAPI[4] library that exposes the StarCraft AI interface to Java programs through the Java Native Interface. JNI-BWAPI uses a shared memory bridge that enables remote processes to access the functions exposed by BWAPI. The resulting interface provides support for authoring StarCraft agents using the ABL planning language.

The ABL decision cycle runs asynchronously from the StarCraft update thread. The BWAPI library provides callbacks for game start, update, and end events. During the game update event, a number of sensors sense the game state and update the agent's working memory. EISBot includes sensors for the map, units, and type data. The agent's perception of the game world is shown in Figure 4.9. The map data describes the dimensions of the map, resource locations, the height of map tiles, whether tiles are buildable or walkable, as well as regions and chokepoints identified by the Brood War Terrain Analyzer [82]. The unit data describes the locations, types, and owners of units on the map and several other attributes. To perform actions in the game world, the agent sends commands to individual units, such as move, attack, build, train, or research. The JNI-BWAPI library handles action synchronization by queuing up events until the next game update and then dispatching batches of commands.

The ABL behaviors are supported by a middleware layer implemented in Java. The middleware includes Java methods for executing mental acts, which are used mostly

[4] `http://code.google.com/p/jnibwapi/`

for modifying working memory and performing complex game state queries. This layer is also used for tasks including event generation, such as creating an event when a new enemy unit type is scouted, and batch commands, such as ordering all units in a squad to attack a target location.

While the goal of the agent interface is to resemble the interface presented to human players as closely as possible, it provides agents with two advantages. First, agents can directly query game state across the entire map, without the need to pan the screen. Therefore, the current agent does not support capabilities for focusing attention on a specific location of the map, and does not extract game state using screen captures. Second, there is no restriction on the number of actions that the agent can perform. Therefore, it may be possible for an agent to abuse the interface in order to exploit the game, such as micromanaging large numbers of units in combat. The potential benefit of exploiting unlimited action commands is limited, because it requires a large amount of attention and impacts only a subset of gameplay.

4.4 Multi-Scale Idioms

ABL provides several mechanisms for supporting multi-scale agents, including concurrent goal pursuit, plan monitoring, and working memory. This section presents design patterns for authoring ABL agents that operate in multi-scale domains. These idioms are useful for developing agents that perform reasoning at multiple scales, manage resources across scales, and focus on specific tasks as necessary. One of the goals of

```
initial_tree {
  spawngoal restockUnits();
  ...
}

parallel behavior restockUnits() {
  with (persistent) subgoal trainInterceptors();
  with (persistent) subgoal trainScarabs();
}
```

Figure 4.10: An example of a daemon behavior in EISBot is the `restockUnits` behavior. The goal it satisfies is spawned as a separate thread of execution and it continuously pursues subgoals for training additional support units.

explaining these idioms is to demonstrate their usefulness in building multi-scale AI, and to give illustrative, concrete examples so that others can employ these idioms.

The main problems that arise when authoring multi-scale agents are dealing with messy abstraction boundaries and handling concurrency across different scales of reasoning. My approach for addressing these concerns is to utilize managers to handle gameplay at different scales, and use message passing idioms to support communication across these scales. Several design patterns are used in EISBot to manage concurrency including resource holds, behavior locks, and priority queues. Applying these patterns enables the agent to use shared resources across different scales.

I have applied the idioms presented here to author daemon behaviors that continuously pursue goals independently of other tasks, managers that assume responsibility for a scale of gameplay, message passing behaviors that produce and consume messages from working memory, resource-locked behaviors that suspend execution to handle concurrency, and unit subtasks behaviors that temporarily claim units for specific tasks.

4.4.1 Daemon Behaviors

A multi-scale system must be able to reason about several goals simultaneously. In ABL, this can be achieved through the use of daemon behaviors. A daemon behavior is a behavior that spawns a new goal that is then continuously pursued by the agent. This new goal can then reason about a separate problem from the current thread of execution. Daemon behaviors in ABL are analogous to daemon threads. In ABL, a daemon behavior can be created using the spawngoal and persistent keywords. Spawngoal is used to create a new goal for expansion and the persistent modifier is used within the spawned behavior to continuously pursue a subgoal.

In EISBot, daemon behaviors are used primarily for handling isolated tasks that minimally impact other scales. An example daemon behavior in EISBot is shown in Figure 4.10. The `restockUnits` behavior satisfies the `restockUnits` goal, which is added to the agent's ABT using a spawngoal step, and pursued in parallel with the agent's other goals. The behavior contains two steps that continuously pursue goals for training additional interceptors and scarabs, which are support units for carriers and reavers. This behavior is included in EISBot, because carriers and reavers cannot do damage unless support units are produced. One of the ways in which this daemon behavior impacts other scales is that producing support units requires spending minerals, which are utilized primarily by the production manager. Resource holds and behavior locks are used to prevent this daemon behavior using minerals allocated for other tasks.

```
parallel behavior incomeManager() {

  // production
  with (persistent) subgoal pumpProbes();
  with (persistent) subgoal buildAssimilators();
  with (persistent) subgoal processExpansionRequests();

  // harvesting
  with (persistent) subgoal mineMinerals();
  with (persistent) subgoal putWorkersOnGas();
  with (persistent) subgoal pullWorkersOffGas();

  // assign new tasks
  with (persistent) subgoal freeWorkers();
  with (persistent) subgoal detectIdleWorkers();
  with (persistent) subgoal checkMinedOut();
}
```

Figure 4.11: The income manager simultaneously pursues a variety of goals including producing additional worker units and resource facilities, harvesting minerals and vespene gas, and assigning workers to perform new tasks.

4.4.2 Managers

Managers are a design pattern for conceptually partitioning an agent into distinct areas of competence. A manager is a collection of behaviors that is responsible for handling a distinct subset of an agent's behavior, such as a scale of reasoning. Managers in an agent pursue goals simultaneously and one of the ways managers can be implemented is using daemon behaviors. The main distinction between managers and daemon behaviors is that managers are tightly coupled with other managers in the agent. Each manager in EISBot implicitly defines an interface that specifies which types of WMEs it receives as inputs and which types of WMEs it generates as outputs.

Authoring agents as a collection of managers provides many benefits. Managers enable a modular design, where new implementations of managers can be swapped into

an agent as needed. It also enables concerns to be separated across scales, such as isolating decisions about which structures to construct at the strategy scale from specific details about construction at the production scale. Partitioning agent behavior into managers also enables domain experts to author sophisticated behavior for specialized tasks, such as micromanagement, without worrying about impacting other aspects of gameplay.

EISBot includes a manager for each of the scales of reasoning identified in Section 4.3. The income manager is responsible for handling several tasks including producing worker units and additional resource facilities, harvesting minerals and vespene gas, assigning workers to new tasks, and releasing workers for use by other managers. A subset of the goals pursued by the manager are shown in Figure 4.11. The income manager implements the following contract:

- **Tasks**
  - Harvest Resources
  - Manage Idle Workers
  - Train Workers
  - Construct Resource Facilities
- **Inputs**
  - *ExpansionRequestWME*: Request to produce an expansion.
  - *WorkerRequestWME*: Request to free a worker unit.

- **Outputs**
  - *IdleWorkerWME*: a worker unit available for a new task.
- **Holds**
  - *ProbeStopWME*: suspends the production of worker units.
  - *GasHoldWME*: suspends workers from gathering vespene gas.

The income manager receives expansion requests and worker unit requests from other managers and generates as output idle workers which can be assigned new tasks. The manager also coordinates with other managers by acknowledging resource holds, such as the `ProbeStopWME`, which requests for the manager to suspend worker production. The manager assumes control of all worker units and is responsible for releasing workers for new tasks when requested from other managers in the system.

4.4.3 Message Passing

In ABL several messaging idioms are possible by using working memory as an internal blackboard [44]. Common messaging patterns in ABL are the message producer and message consumer patterns. A message producer is a behavior that adds a WME to working memory, while a message consumer removes a WME from working memory after operating on its contents. In ABL, WMEs can be added to working memory and removed by mental acts. Examples of the message producer and consumer patterns are shown in Figure 4.12.

```
sequential behavior messageProducer() {
  mental_act {
    BehavingEntity.getBehavingEntity().addWME(new MessageWME());
  }
}

sequential behavior messageConsumer() {
  precondition {
    message = (MessageWME)
  }

  subgoal processMessage(message);

  mental_act {
    BehavingEntity.getBehavingEntity().deleteWME(message);
  }
}
```

Figure 4.12: ABL supports behaviors that produce and consume messages. This example shows a polling-based consumer behavior, which can be selected for expansion only when a `MessageWME` is placed in working memory.

A message producer is a behavior that adds a working memory element to the agent's working memory. Posting WMEs to working memory enables a manager to request other managers to perform actions and provides a method for decoupling decision making and action execution concerns. In EISBot, decisions about which units to train, buildings to construct, and upgrades to research are selected by the strategy manager and the processes for executing these decisions are delegated to the other managers by posting requests to working memory.

ABL supports polling-based and event-driven message consumers. The consumer behavior in Figure 4.12 is an example of polling-based messaging, because the behavior can be selected for expansion only when a `MessageWME` is present in working memory. In order for the behavior to be selected for expansion, the agent needs to continuously pursue the subgoal to consume messages. Another approach is event-driven

consumers that wait for messages to be placed in working memory. An example event-driven consumer is shown in Figure 4.13. The first step in the behavior is a success test that suspends the execution of the behavior until a construction request is added to working memory. In response to receiving a construction request, the behavior binds the request to a behavior-scoped variable, processes the request, and then removes it from working memory. The event-driven approach can be used for consuming messages when there is a single behavior for accomplishing a goal, while the polling-based approach is useful when consuming messages should not suspend the execution of a behavior.

While using message passing can cause agent authoring to become more complex, because it adds a level of separation between action selection and execution, it provides several benefits. Message passing provides a mechanism for serializing requests, as demonstrated by the construction request processing behavior in Figure 4.13. In this example, construction requests are queued up in working memory and dispatched one at a time by the behavior. In addition to supporting communication across managers, message passing can be used in ABL agents to manage control flow, prevent behaviors from blocking, and enable multiple behaviors to be the source of messages, including external components.

4.4.4 Behavior Locking

One of the idioms used in EISBot to manage concurrency is behavior locking. Behavior locking is a design pattern for suspending the execution of a behavior using working memory. An example of behavior locking in EISBot is shown in Figure 4.14.

```
sequential behavior processConstructionRequests() {
  ConstructionRequestWME request;

  with (success_test {
    request = (ConstructionRequestWME)
  }) wait;

  with (persistent when_fails) subgoal handleConstruction(request);

  mental_act {
    BehavingEntity.getBehavingEntity().deleteWME(request);
  }
}
```

Figure 4.13: An example of an event-driven message consumer in EISBot. The behavior waits for a construction request, pursues the subgoal of handling the request until it succeeds, and deletes the request.

```
sequential behavior putWorkersOnGas() {
  AssimilatorWME assimilator;

  with (success_test {
    !(GasHoldWME)
    assimilator = (AssimilatorWME numberWorkers<3)
  }) wait;

  subgoal assignWorkers(assimilator);
}
```

Figure 4.14: An example of behavior locking in EISBot. The success test in this behavior blocks its execution when a `GasHoldWME` is in working memory.

The success test in this behavior will suspend execution if a `GasHoldWME` is placed in working memory, preventing additional worker units from being assigned to gathering vespene gas.

Behavior locking is one of many ways to suspend the execution of behaviors in ABL. ABL also supports meta-behaviors, which modify the agent's ABT at runtime [59]. Meta-behaviors can be used in an agent to suspend or resume the execution of behaviors. While meta-behaviors provide a powerful mechanism for modifying an agent's active goals, behaviors in EISBot that access shared resources can use WMEs as locks, which can be exposed to external processes.

4.4.5 Unit Subtasks

EISBot combines high-level decision making with reactive unit-level tasks. This is achieved through the use of unit subtask behaviors in ABL. Unit Subtasks are behaviors that temporarily claim a unit to perform a specific task. In EISBot unit subtasks are used to perform micromanagement behaviors, assign worker units to defend against rushes, and regroup squads into tighter formations.

The general pattern for unit subtasks is to lock a specific unit, spawn a new goal for managing the unit, and then release the lock on the unit. An example unit subtask behavior in EISBot is shown in Figure 4.15. The `dragoonDance` behavior waits for a condition in which the agent controls a damaged dragoon that is executing an attack order. When these conditions are met, the behavior marks the unit as fleeing, and spawns a new goal for the dragoon to flee, which causes the unit to move away from

```
sequential behavior dragoonDance() {
  DragoonWME unit;

  with (success_test {
    unit = (DragoonWME order==Attack task!=FIGHTER_FLEE damaged==true)
  }) wait;

  mental_act {
    unit.setTask(FIGHTER_FLEE);
  }

  spawngoal dragoonFlee(unit, 24);
}

sequential behavior dragoonFlee(DragoonWME unit, int delay) {
  act move(unit, fleeX, fleeY);
  subgoal WaitFrames(delay);

  mental_act {
    unit.setTask(FIGHTER_ATTACK);
  }

  subgoal reengage(unit);
}
```

Figure 4.15: Unit subtasks are used to implement micromanagement behaviors. The `dragoonDance` behavior temporarily commands damaged units to flee from combat and then re-engage.

combat and then be marked as attacking before re-engaging the enemy. Unit subtasks can also be used to issue commands to groups of units. The `defendBase` behavior in EISBot spawns a goal for workers to defend the Nexus, which recursively requests assistance from additional worker units until the Nexus is no longer under attack.

4.5 Design Patterns in EISBot

I have used the idioms presented in this section to author behaviors for EISBot. The agent's behavior is specified as 3,265 lines of ABL code, which includes 171 behaviors, and is supported by thousands of lines of Java code that define sensors and

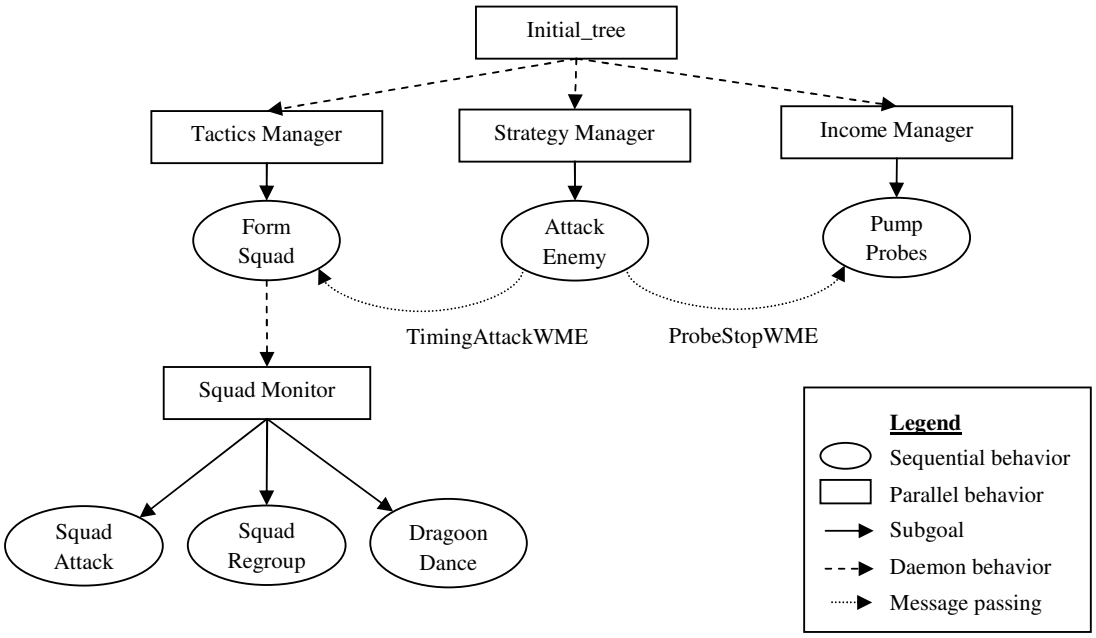

Figure 4.16: Several goals are pursued by the different managers in EISBot's ABT. This visualization shows how a behavior that satisfies the goal `AttackEnemy` communicates with other managers including the tactics manager, which consumes `TimingAttackWME` messages and spawns new squad monitor goals, and the income manager, which locks the `PumpProbes` goal when a `ProbeStopWME` is placed in working memory.

actuators, implement working memory elements, provide specialized queries for condition checks, and support mental acts. During gameplay, the agent's ABT reaches over 100 executing steps and continues to increase as new units and squads are instantiated in the game. The base ABL agent plays at the level of an amateur player and detailed results are presented in Chapter 7.

A visualization of a subset of EISBot's active behavior tree is shown in Figure 4.16. The `initial_tree` behavior starts several daemon behaviors that spawn the different managers. The managers in the agent communicate by posting and removing WMEs from working memory. The figure shows the message passing design pattern being applied by the strategy and tactics managers. The behavior satisfying the `AttackEnemy` goal posts a `TimingAttackWME` to working memory during its execution, and the behavior satisfying the `FormSquad` goal removes the WME from memory. Message passing is also used between the strategy and income manager, but in this case the income manage does not consume the message and instead suspends pursuit of the `PumpProbes` goal using the behavior locking pattern.

The visualization also shows that new daemon behaviors can be instantiated in response to receiving a message. A new squad monitor goal is spawned each time a `TimingAttackWME` is posted to memory. The behavior that satisfies this goal is responsible for grouping unassigned combat units into a squad and monitoring units in the newly formed squad. This daemon behavior handles both goals at the squad level, such as attacking opponent forces and regrouping, and the individual unit level, such as the `DragoonDance` goal that micromanages individual units. One of the benefits of using

ABL to implement agent behavior is that individual units as well as the formulation of squads are managed within a unified environment, enabling the agent to dynamically assign units to roles based on the current situation.

By using managers, message passing, and unit subtasks, EISBot is able to reason about different goals at different scales simultaneously, and to coordinate these gameplay scales in order to achieve a coherent result. Managers for different scales can not only override one another when necessary, but they can pass messages to influence or direct other managers in the system. This leads ultimately to an agent that is responsive, flexible, and extensible: an agent that is able to respond to highly specific circumstances appropriately without losing track of long-term goals [109].

4.6 Application to RoboCup

The multi-scale idioms presented in this chapter can be applied to authoring agents outside the genre of RTS games. This section explores how the idioms could be applied to a new task using a top-down approach. One of the the multi-scale AI problems I identified is RoboCup soccer, which involves decision making at individual, squad level, and team scales. In the RoboCup Small-Size League, a team of six robots are managed by a centralized system[5]. An objective of the league is to focus on multi-agent coordination. The role of the centralized system is similar to the role of the commander in an RTS game in that it manages a team of robots and can delegate actions to individuals. This league presents a multi-scale AI problem and the multi-

[5] `http://small-size.informatik.uni-bremen.de`

scale idioms introduced in this chapter can be leverage to design a system for this task.

The Small-Size League provides an environment in which a team of robots are monitored by a centralized system. Each robot can pursue individual goals, such as ball control, while the focus is on multi-robot coordination. In this environment, reactive planning can be applied to implement the centralized decision making system, while individual goals can be managed using on-board systems. The role of the reactive planner in this configuration is to identify tasks for each of the robots to perform, select joint interactions for squads of robots, and manage team goals. To coordinate gameplay across each of these scales, I propose managers for this task, identify message passing across managers, and discuss the potential uses of the multi-scale idioms in a RoboCup agent.

The first step in applying the idioms to a new problem is to decompose gameplay into distinct competencies. Each of these competencies is handled by a manager in the agent. For RoboCup soccer, I propose the following managers:

- **Offense Manager**: responsible for controlling robots on offensive.
- **Defense Manager**: handles decision making for robots on defense.
- **Keeper Manager**: responsible for controlling the goal keeper.
- **Coach Manager**: handles position assignment and level of aggressiveness.

The keeper manager operates at the individual scale, the offense and defense managers operate at the squad scale, and the coach manager operates at the team scale.

The next step is to identify cross-cutting concerns between the managers in the system. In this environment, players can be utilized by the offense, defense, and keeper managers. One way of managing players is to have the coach manager responsible for assigning players to managers. An additional concern across managers is facilitating coordination between defensive and offensive players. One way of facilitating coordination across squads is by having managers express intended actions, such as passing or blocking players. The message passing idioms can be used to express intent between players, using the following WMEs:

- **PassIntentWME**: The player intends to pass the ball to a teammate.
- **GuardIntentWME**: The player intends to guard an opponent.

The messages can be used to perform coordination actions across multiple units, which may be handled by separate managers.

The third step is to identify player subtasks, in which players are temporarily assigned to new individual tasks and are then re-assigned to a manager. Example subtasks in RoboCup soccer include:

- **Throw ins**: The player leaves the squad to throw in the ball.
- **Ball pursuit**: The player races to retrieve the ball.
- **Defense assist**: The player assists a teammate in defending an opponent.

These subtasks are short interactions that involve individual player tasks. They enable players to pursue new roles for a duration before being reassigned to squad-level goals.

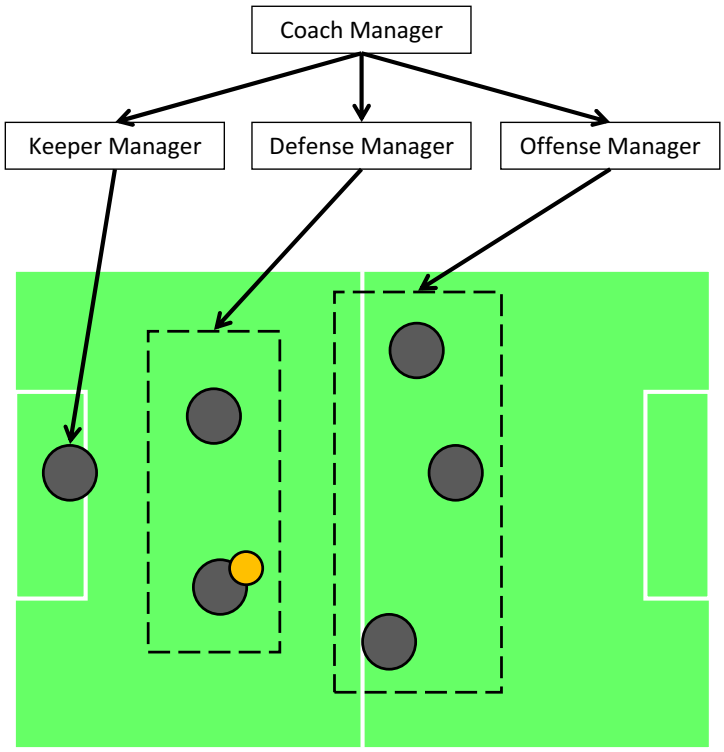

Figure 4.17: The proposed RoboCup agent includes four managers. The coach manager assigns players to each of the managers, and the other managers coordinate using message passing.

The daemon behavior and behavior locking idioms can also be applied to this environment. One of the tasks that can be performed in isolation of other tasks is identifying candidate passing lines for both teams. The daemon behavior pattern can be used to author behaviors that monitor the field and place PassingLineWMEs in working memory when relevant. The behavior locking pattern can be used to enable or disable behaviors based on the current game context. For instance, a KeeperAttack behavior could be locked unless the coach manager decides to pursue an aggressive strategy.

An overview of the proposed RoboCup agent is shown in Figure 4.17. The coach manager is responsible for assigning players to units and selecting formulations and level of aggression. The offense and defense managers each handle a squad of players. Players can express intent to perform passing and guarding actions, which enable units across squads to coordinate actions. This decomposition of gameplay enables the system to specialize in offensive and defensive tasks, while still working towards team-level goals.

4.7 Summary

One of the characteristics of RTS gameplay is that it requires the ability to simultaneously manage several interconnected tasks while pursuing higher-level goals. StarCraft presents players with multiple tasks to perform where there is not a strict separation between tasks and the performance in each task impacts other tasks. I classify domains with these characteristics as multi-scale domains. A multi-scale AI

problem is an instance of a multi-task problem in which actions are performed across multiple scales, tasks performed at each scale are interrelated, and decision making and acting are performed in real-time.

I advocate reactive planning as a technique for authoring agents that operate in multi-scale domains, because it provides several mechanisms necessary for multi-scale reasoning. The ABL reactive planning language supports agents that perform concurrent goal pursuit, dynamically spawn new goals during runtime, and manipulate working memory [59]. This chapter describes reactive planning idioms that assist in the development of multi-scale agents. These include design patterns for spawning daemon behaviors that handle isolated tasks, organizing agents as a collection of managers that are each responsible for a scale of reasoning, producing and consuming messages, locking behaviors, and dynamically creating new goals to manage individual units and squads of units.

Each of these idioms is applied in EISBot, enabling the agent to concurrently pursue high-level strategic goals while simultaneously reacting to unit-specific events. The agent includes a manager for each scale of gameplay and facilitates coordination across managers using message producer and consumer patterns. The resulting system performs actions at multiple scales, and in real-time while pursuing the high-level goal of defeating opponents. While the idioms presented here are demonstrated in the ABL reactive planning language, many of these patterns are more general and can be applied to build multi-scale AI in other agent authoring methodologies.

Chapter 5

Learning From Demonstration

StarCraft gameplay presents a number of challenges including decision making with imperfect information and tracking the evolving meta-game. To perform at an expert level, players employ adaptation, anticipation, and estimation capabilities in order to learn new strategies as gameplay evolves and to reason about unknown current and future game states. My approach for realizing these capabilities in an agent is to learn from gameplay demonstrations generated by professional players.

There are several benefits to learning gameplay behavior from professional players. As new strategies are discovered by players, they can be added to the agent's collection of strategies and reused when similar gameplay situations are encountered. Another benefit is that an agent can make predictions about the actions a player will perform based on previous observations, and predictions can be made by analyzing the prior actions of individuals or groups of players. Learning from demonstration also enables agents to harness *collective intelligence* demonstrated by thousands of players,

rather than specified by a small number of designers [79].

Using demonstrations from players presents several challenges, because demonstrations contain noisy and non-intentional actions. My approach for addressing these concerns is to focus on learning individual gameplay scales, which reduces the amount of behavior that needs to be learned from examples. Rather than attempt to learn all gameplay behavior from demonstrations, I focus on learning specific aspects of gameplay that are intractable or impossible for a domain expert to author. Decision making policies learned for individual scales are integrated into the overall agent, which supports both hand-authored behaviors and behaviors learned from demonstrations.

This chapter presents methods that enable adaptation, anticipation, and estimation capabilities in EISBot. I discuss three different ways learning from demonstration is used in the system: model training for classification and regression algorithms that perform strategy prediction, case-based goal formulation for strategy selection and opponent modeling, and parameter selection for optimizing a particle model that tracks opponent forces. These methods enable the agent to learn new strategies, anticipate the actions of opponents, and build predictions of candidate opponent locations. By utilizing these methods, EISBot emulates many of the capabilities demonstrated by expert human players.

5.1 Strategy Prediction

One of the ways that a player can gain an advantage over an opponent in StarCraft is to anticipate the opponent's actions and execute actions that counter the opponent's plan of action. In order to effectively pursue a strategy, a player needs to invest in production facilities, upgrades, and forces. This process can take a substantial amount of time and transitioning to a new strategy in the middle of a game can be detrimental to the player's overall combat strength. Therefore, selecting a strategy to enact involves a large degree of commitment. Detecting an opponent's strategic plan early on in the game can lead to several advantages, because the player can anticipate the type of army composition to expect, have an idea of when new unit types will enter play, and predict when the opponent will be aggressive and attack. These predictions can be used to select a strategy that counters the most likely plan of action performed by the opponent. Often, thwarting an opponent's plans requires executing counter measures across multiple gameplay scales.

My work on developing methods for anticipating the actions of opponents is in part motivated by capabilities demonstrated by StarCraft commentators. During a match, commentators explain potential strategies that players are pursuing based on observed actions early on in the game. Typically, players will perform actions with different timings based on the selected strategy. These timings provide hints that commentators use to build predictions of the strategies being executed by players. By studying large numbers of games and identifying when specific actions are performed

by players, commentators can determine which strategies are being pursued with a high degree of certainty. The objective of the strategy prediction method presented in this section is to emulate this anticipation capability, which is demonstrated by both players and commentators.

I represent strategy prediction as an intent recognition problem, where the goal is to identify which strategic plan an opponent is executing. Specifically, I model strategy prediction as a classification problem, where the input to the model is a description of the actions performed by a player and the output is a strategy label. To train the model, I collected thousands of StarCraft demonstrations, in the form of replays, converted them into a feature vector representation, and labeled them using a rule set that identifies common StarCraft strategies. This method uses demonstrations as examples for training classification algorithms.

Each vector corresponds to a single game and encodes the timings of strategic actions performed by each of the players. In this representation, each feature describes when a unit type, building type, or upgrade is first produced. This encoding provides a compact representation for reasoning about a player's expansion of the tech tree. One of the advantages of this representation is that it is possible to simulate different time steps during a game, by limiting the maximum feature values. A disadvantage of this representation is that it captures actions at only the strategy scale, and other scales such as economy can have a large impact on strategy selection. I developed this encoding prior to the release of BWAPI, which limited the data available for analysis to actions contained in game logs, as opposed to complete game state information.

There are two main limitations of this classification method. First, the strategies that the model can predict must be known ahead of time in order to label the feature vectors. Therefore, this method does not support tracking the evolving metagame, because the set of strategies it can predict is fixed. Another limitation of this approach is that it trains models for identifying strategies for a specific phase of the game, and supporting additional phases requires additional rule sets. Approaches for overcoming these limitations include probabilistic models [26, 95] and the opponent modeling method introduced in Section 5.2.1.

The feature vectors can also be used to train models that anticipate when specific unit types, building types, or upgrades will be produced by a player. I represent timing prediction as a regression problem, where the input to the model is a description of the actions performed by the player, and the output is a prediction of when a specific action will be performed. These predictions can be used by the system to anticipate when new unit types and building types will be produced by opponents.

In this section, I present several off-line experiments performed in order to evaluate the accuracy of different classification algorithms on the strategy prediction and timing prediction tasks. These experiments evaluate the models in simulated perfect information and imperfect information environments. The results show that an opponent's strategy can be predicted with a large degree of confidence, even when reasoning with imperfect information. Additionally, in the early stages of the game multiple classification techniques are more accurate than the rule set used to label the strategies, emulating the anticipating capability of players.

Table 5.1: The Web crawler collected thousands of StarCraft replays. This table shows the number of games collected for each of the race match-ups.

Type	**# Replays**
Protoss vs. Protoss	542
Protoss vs. Terran	1,139
Protoss vs. Zerg	1,024
Terran vs. Terran	628
Terran vs. Zerg	1,150
Zerg vs. Zerg	1,010
Total	5,493

5.1.1 Data Collection

Several websites are dedicated to collecting and sharing StarCraft replays with the gaming community, a large portion of which are from professional and high-ranked amateur matches. Therefore, it is possible to mine these websites in order to build a collection of replays that is a representative set of expert play in StarCraft. Due to the large number of replays available, it is possible to learn a variety of strategies on several maps against different play styles.

I developed a web crawler to collect StarCraft replays from GosuGamers.net and TeamLiquid.net, two popular StarCraft websites. The Web crawler downloaded collections of replays from professional tournaments including BlizzCon, World Cyber Games, MBC Starleague and the StarCraft Proleague. The crawler also collected replays from top-ranked players on ICCup.com, a popular StarCraft ladder ranking system. I limited replay collection to *1 vs. 1* matches, because it is the most common game type for professional StarCraft matches. The resulting number of game replays for the different race match-ups are shown in Table 5.1.

My goal is to build a general model for anticipating opponent actions in StarCraft. This differs from previous work, which has focused on modeling single opponents [28] or work that has been limited to at most a few hundred game logs [40]. By applying data mining to a large number of game logs, I can develop predictions models that are not limited to a single opponent, set of maps, or style of gameplay.

5.1.2 Game Encoding

StarCraft replays are stored in a proprietary, binary format. I used a third-party replay analyzer[1] to convert the replay files to game logs. A subset of an example log is shown in Table 5.2. The game logs contain only user interface actions performed by each player. Game state is not available in the logs, because replays are viewed by performing a deterministic simulation based on the user interface actions. This limits how much state can be extracted from replays, because information, such as the player's current amount of resources, is not available for analysis. However, the production of different unit types and building types can be extracted, providing sufficient information to analyze a player's build order.

The lack of game state provides a challenge for strategy prediction, because only the players' user interface actions are available for analysis. One approach to overcome this limitation is the use of a state lattice to capture the sequence in which actions are performed. State lattices have been applied to opponent modeling in Wargus [83] and StarCraft [40]. A state lattice is a directed graph without cycles, where each node

[1] `http://lmrb.net/`

Table 5.2: Game logs extracted from replays include user interface actions performed by the players. This partial game log shows production actions from the start of a Terran vs. Zerg game.

Player	Game Time (minutes)	Action
Player 2	0:00	Train Drone
Player 1	0:00	Train SCV
Player 2	1:18	Train Overlord
Player 1	1:22	Build Supply Depot
Player 1	2:04	Build Barracks
Player 2	2:25	Build Hatchery
Player 1	2:50	Build Barracks
Player 2	2:54	Build Spawning Pool
Player 1	3:18	Train Marine
Player 2	4:10	Train Zergling

represents a unique expansion of the tech tree. State lattices are built in a similar manner to constructing decision trees (see [40] for a more detailed explanation). State lattices are useful for predicting the next action given a sequence of actions. However, state lattices are unable to predict when the next action will occur.

The goal of my representation is to capture the timing aspects of a player's strategic decisions in a game, enabling the prediction of distinct strategies and the timing of the opponent's actions. My feature-vector representation contains temporal features that record when the player expands different branches of the tech tree. Each feature describes when a unit type, building type, or upgrade is first produced. For each game log, two feature vectors are constructed. Each vector represents a single player's actions for an entire game. Formally, the representation is a feature vector where each

feature f, for a player P, is defined as follows:

$$f(x) = \begin{cases} t & : \text{time when } x \text{ is first produced by } P \\ 0 & : x \text{ was not (yet) produced by } P \end{cases}$$

where x is a unit type, building type or unit upgrade. Each race has a different number of features, based on the tech tree and upgrades. For example, the Protoss representation contains 56 features. A subset of an example feature vector for a Zerg player is shown in Table 5.3. In this example, the *Hydralisk* feature has a value of 14:51, which corresponds to the game time when the first hydralisk was produced by the player, while the *Queen* feature has a value of 0:00, because a queen was not produced by the player during the game. This encoding, while not requiring complete information about the game state, captures both the tech tree expansion and timing of a player's strategy. This representation varies from related work which encodes single game traces as several cases [73, 104].

After encoding the replays into a feature vector representation, I analyzed the timing distributions of several of the features. For some race match ups, the strategy that a player is pursuing can be anticipated based on the timing of a single feature. For example, there is a wide variety of timings exhibited by Zerg players producing their first spawning pool versus Terran opponents, as shown in Figure 5.1. The spawning pool is a tier 1 production building that enables a player to produce combat units and is often delayed in order to focus on economy. The figure shows three peaks in timing production that correspond to players executing *9-pool*, *over-pool*, and *12-hatch* strategies, where *9-pool* is an aggressive build order, *over-pool* slightly delays combat units to improve

Table 5.3: A subset of a feature vector for a Zerg player shows that a hydralisk was first produced at 14 minutes and 51 seconds into the game, while no queens were produced.

Action (Attribute)	**Game Time (Value)**
Second Hatchery	2:42
Spawning Pool	3:07
Lair	5:48
Zergling	4:23
Zergling Speed	12:10
Hydralisk Den	12:52
Hydralisk	14:51
Lurker	15:39
Hydralisk Speed	16:48
Hydralisk Range	20:01
Queen	0:00

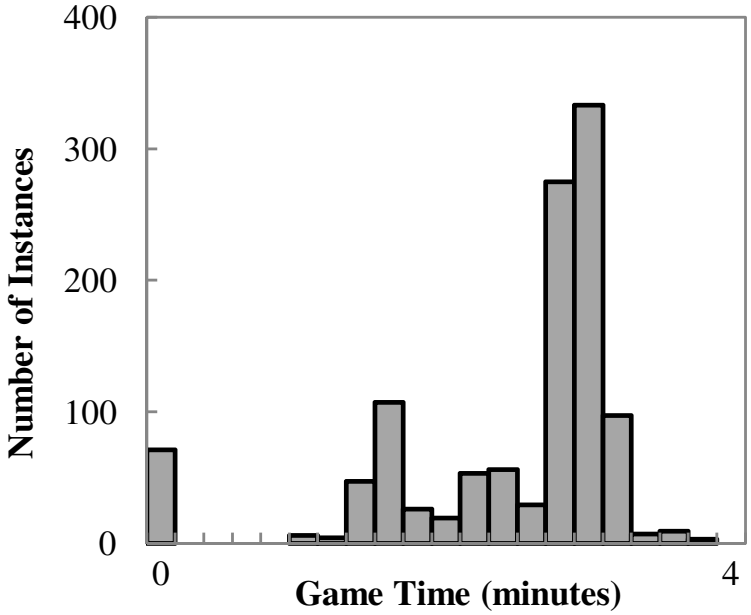

Figure 5.1: The timing distribution of Spawning Pool production in Zerg versus Terran games shows that players typically follow three openings, and prefer expansion focused build orders.

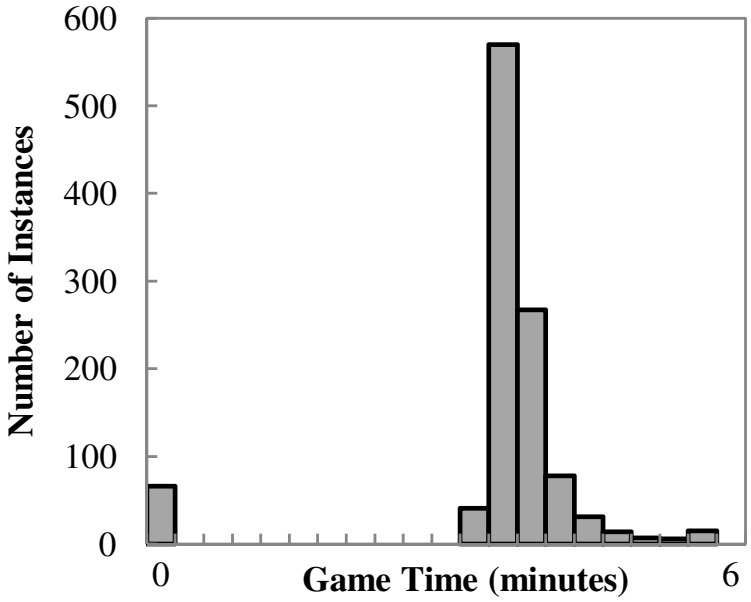

Figure 5.2: The timing distribution of Factory production in Terran versus Protoss games shows little variation, indicating that Terran strategies do not branch until after factory production.

economy, and *12-hatch* is an economy focused build order. While each of these timings can develop into different strategies later in the game, they clearly indicate whether the player is pursuing an aggressive strategy early in the game.

There are other features where the timing indicates little information about the player's strategy. The timing distribution for factory production in Terran versus Protoss games is shown in Figure 5.2. The factory is a tier 2 production building that enables players to produce combat units useful for defending against Protoss opponents. This timing distribution indicates that the general trend in Terran versus Protoss gameplay is to follow a fixed opening and then vary once a factory has started construction. In order to determine which strategy a Terran player is pursuing, it is necessary to

Table 5.4: The strategy distributions for Protoss show that several different strategies are used based on the opponent race.

Strategy	Versus Protoss	Versus Terran	Versus Zerg
Fast Legs	1%	1%	10%
Fast DT	18%	16%	8%
Fast Air	1%	1%	20%
Expand	22%	31%	46%
Reaver	9%	12%	3%
Standard	27%	32%	1%
Unknown	22%	7%	12%

incorporate the timings of several different buildings, units, and upgrades.

5.1.3 Labeling Replays

Each of the feature vectors is labeled with a strategy using rules based on analysis of expert play. A different rule set was used for labeling each of the different races. Each rule set is designed to capture the tier 2 strategies of a race. The rule sets label logs with strategies based on the order in which building types are produced. The rule set for labeling Protoss strategies is shown in Figure 5.3. In the figure, tier 1 strategy refers to strategic decisions made in the early stages of the game. The strategy of the player is not labeled until a tier 2 strategy decision is made, such as building a Stargate, which is labeled as a "Fast Air" strategy. Six strategies were created for each race. If a game does not fit any of the rules, then the strategy is labeled as unknown.

The rule sets were designed to capture a wide variety of strategies in StarCraft. Distributions of the builds versus different races are shown for Protoss in Table 5.4. In certain match-ups, such as Protoss versus Terran, a wide variety of strategies are

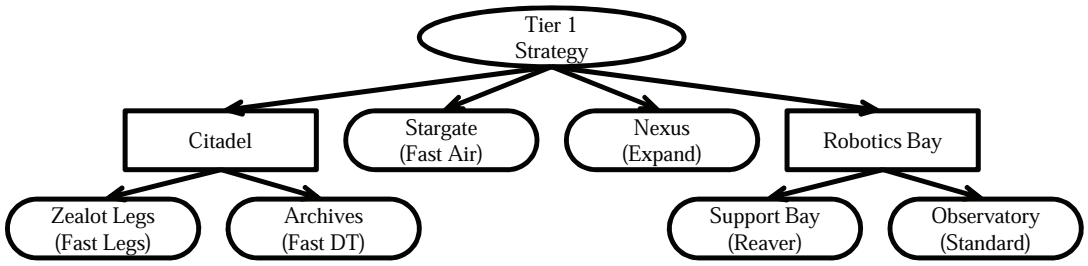

Figure 5.3: The rule set for labeling Protoss replays includes six strategies. The tree for labeling strategies is traversed by selecting the first building produced in the child nodes.

commonly used. However, strategy prediction is easier in some match-ups where a single strategy dominates. For example, the two-hatchery mutalisk strategy is used in over 70% of Zerg versus Zerg games.

5.1.4 Build-Order Prediction

I represent strategy prediction as a multi-class classification problem. The goal of the classification model is to predict the strategy an opponent is pursuing at different stages throughout the game. I applied the following algorithms to classify opponent build orders:

- J48 – C4.5 decision tree [86].
- k-NN – Nearest neighbor [2].
- NNge – Non-nested generalized exemplars [57].
- LogitBoost – Additive logistic regression [34].

I used the implementations of these algorithms provided in the Weka [112] toolkit. The LogitBoost algorithm was configured to use 100 iterations and a shrinkage rate of 0.5. All of the other algorithms used the default settings. Ten-fold cross validation was performed for all of the experiments.

In addition to the Weka algorithms, I evaluated two additional classifiers. The rule set classifier predicts strategies using the exact rules used to label the strategies. Since the replays are labeled based on tier 2 strategies, this classifier is not accurate until the opponent's strategy has been executed. I also implemented a state lattice classifier based on Hsieh and Sun's approach for comparison [40].

The algorithms were evaluated at different time steps throughout the game. I simulated different time steps by setting all features with a value greater than the current game time to 0. Time τ is simulated in a replay by applying the following transformation to the feature vector:

$$f(x,\tau) = \begin{cases} f(x) & : f(x) \leq \tau \\ 0 & : \text{otherwise} \end{cases}$$

Consider the following feature vector: $f =< 0, 1000, 2000, 3000 >$. After applying the transformation with $\tau = 1500$, the resulting vector would be: $f =< 0, 1000, 0, 0 >$. This transformation is applied to training data as well as the test data. The precisions of the algorithms versus game time for strategy prediction are shown in Figure 5.4 and Figure 5.5. A comprehensive listing of the performance of the algorithms at five and ten minutes game time is shown in Table 5.5.

Table 5.5: Precision of the strategy prediction models at 5 minutes and 10 minutes game time.

	5 minutes				10 minutes			
	NNge	**J48**	**Boosting**	**Lattice**	**NNge**	**J48**	**Boosting**	**Lattice**
PvP	**0.49**	0.43	0.47	0.39	0.80	0.81	**0.86**	0.56
PvT	**0.68**	0.63	0.61	0.45	0.89	0.91	**0.94**	0.62
PvZ	0.63	0.63	**0.66**	0.62	0.85	0.87	**0.87**	0.44
TvP	**0.76**	0.66	0.63	0.45	0.81	0.80	**0.94**	0.51
TvT	**0.82**	0.75	0.77	0.57	0.85	0.81	**0.92**	0.56
TvZ	**0.91**	0.88	0.90	0.86	**0.94**	0.90	0.86	0.60
ZvP	0.53	0.56	**0.60**	0.48	0.84	0.85	**0.87**	0.49
ZvT	**0.53**	0.50	0.49	0.41	0.87	**0.91**	0.89	0.65
ZvZ	0.83	0.82	0.83	**0.84**	0.94	0.95	**0.95**	0.82
Avg.	**0.69**	0.65	0.66	0.56	0.86	0.87	**0.91**	0.58

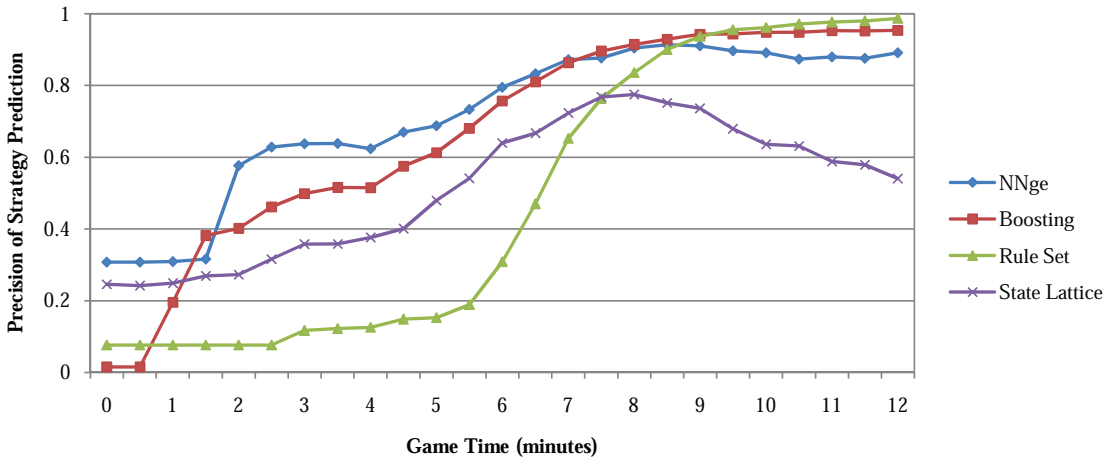

Figure 5.4: The precision of the strategy prediction classifiers show that instance-based models initially performed best, while boosting approaches outperformed the other models later in the game. This figure shows the precision of the models in Protoss versus Terran games.

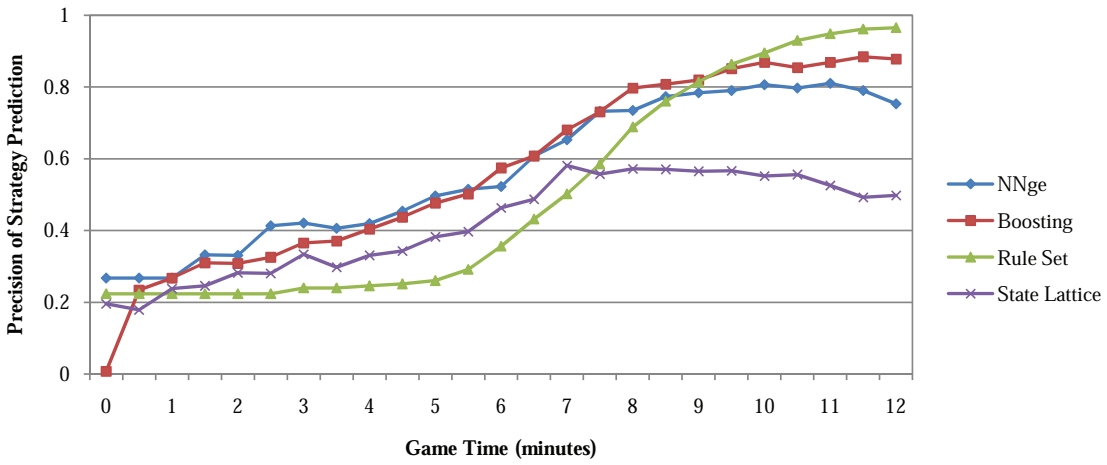

Figure 5.5: The performance across different strategy prediction classifiers was less noticeable for Protoss versus Protoss games.

The results show that different algorithms are better at different stages of the game. The instance-based algorithms (NNge and k-NN) perform well in the initial stages of the game, but degrade in the later stages of the game, while boosting performs poorly initially and improves in the later stages. All of the machine learning algorithms outperformed the state lattice classifier.

Interestingly, the machine learning algorithms had higher precision than the exact rule set during the first 8 minutes of the game. While the precision of the rule set classifier eventually reaches 100%, there is a noticeable difference between this classifier and the machine learning algorithms in the early stages of the game. These results indicate that the algorithms have "foresight" of the opponent's strategy. Here, I define "foresight" to be the area between the classification algorithm and the rule set. Given this metric, the machine learning algorithms clearly outperform the state lattice classifier.

A second set of experiments analyzed the effects of noise on the classification algorithms. These experiments simulate the *fog-of-war* in StarCraft, which limits visibility to portions of the map in which the player controls units. Delayed scouting can be simulated by adding noise to features, while inability to scout an opponent's base can be simulated by transforming a subset of the features.

The first imperfect information experiment added a uniform distribution of noise to individual features, which simulates delayed scouting in StarCraft. This transformation is applied by incrementing the value of each feature: $f'(x) = f(x) + U(0, n)$, where n represents a delay in scouting. The results for this noise transformation are shown in Figure 5.6. All of the algorithms degrade in precision as more noise is applied to the test data set. However, the precision of the k-NN algorithm does not degrade as quickly as the other algorithms. In this experiment, the precision of the state lattice decreased much more rapidly than the other algorithms.

The second imperfect information experiment tests the effects of missing features on the classification algorithms. Attributes were randomly set to 0, based on a missing attribute probability. This simulates a player that is unable to scout an opponent's base in StarCraft. The results for the missing attribute transformation are shown in Figure 5.7. The precision of the machine learning algorithms decreased linearly with respect to the ratio of missing attributes, while the precision of the state lattice classifier degrades to that of a random classifier after 20% of attributes are missing. The results of these two experiments indicate that the machine learning algorithms are more tolerant of noisy and missing features than the state lattice classifier.

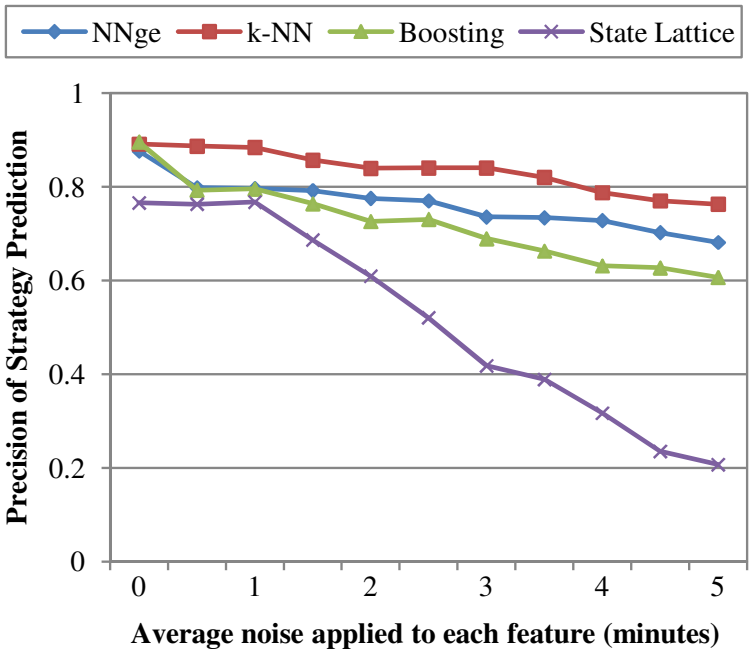

Figure 5.6: The first imperfect information experiment simulated delayed scouting by applying noise uniformly to individual attributes.

5.1.5 Timing Prediction

I represent timing prediction as a regression problem. The goal of the regression model is to predict when a specific unit type, building type, or upgrade type will be produced by the opponent. I applied the following algorithms to the timing prediction task:

- ZeroR - Estimates the mean.
- LinearRegression - Regression with Akaike criterion [4].
- M5' – Inducing model trees [102].
- AdditiveRegression - Stochastic gradient boosting [35].

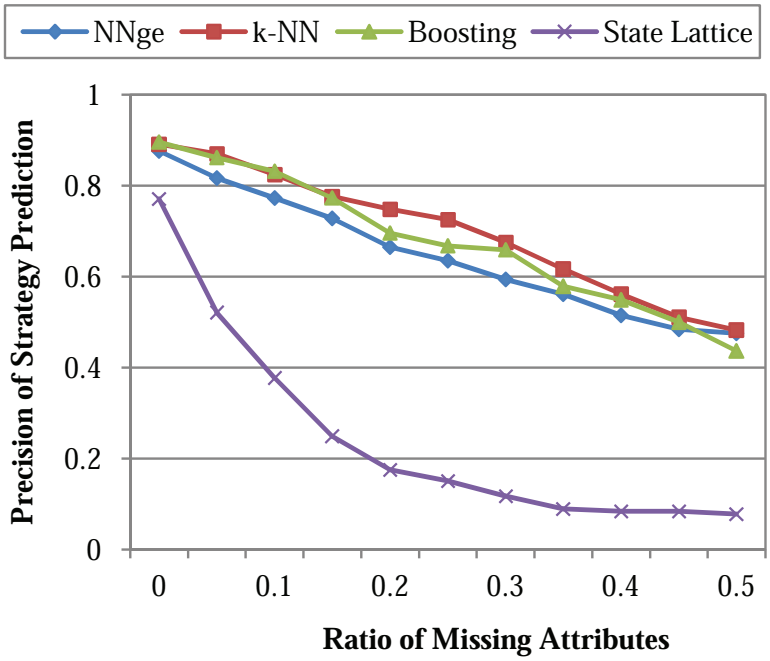

Figure 5.7: The second experiment simulated inability to scout an opponent. The missing attribute ratio specifies the probability that an attribute will be set to 0.

The additive regression algorithm was configured to use 100 iterations and a shrinkage rate of 0.5. All of the other algorithms used the default settings. Ten-fold cross validation was performed for all of the experiments.

The regression models are applied to the task of predicting the timing of individual features. The training and test data sets were transformed prior to training the algorithms. A transformation was applied to the features to simulate the game state prior to the production of the individual feature. The following transformation was applied for each feature, y, in the representation:

$$f(x,y) = \begin{cases} f(x) & : y \neq x, f(x) \leq f(y) \\ 0 & : \text{otherwise} \end{cases}$$

This transformation sets the feature vector to the game state just before the unit type, building type or unit upgrade, y, was produced. Consider the following feature vector: $f =< 0, 1000, 2000, 2000 >$. These features correspond to worker timing, barracks timing, depot timing, and marine timing. The transformation is applied for a specific feature, such as depot timing. In this example, $f(y) = f(\text{depot}) = 2000$, and applying the transformation for the depot timing feature would result in the following vector: $f =< 0, 1000, 0, 2000 >$.

A subset of the regression results for Zerg versus Terran games is shown in Table 5.6. The M5' algorithm performed the best on almost all of the features. The algorithms perform poorly on the Zergling Speed feature, which has a standard deviation of 8 minutes. While M5' predicted action timing with the smallest error, linear regression performed well on highly correlated features. For example, linear regression is able to accurately predict observer timing, because there is a strong correlation between constructing an observatory and producing an observer.

The M5' algorithm was able to predict the timing of the lair and hive structures with a mean error of less than 40 seconds. A lair enables Zerg players to build tier 2 units and buildings, while a hive enables Zerg players to build tier 3 units and buildings. Therefore, M5' can predict when the player is upgrading to the next tier of the tech tree with an average error of less than 40 seconds.

Table 5.6: Results from the timing prediction tasks show that M5' performed best on almost all of the features. These values show the average difference between predicted and actual action timings, in minutes and seconds.

Action	ZeroR	Linear Regression	M5'	Additive Regression
Spawning Pool	0:28	0:17	**0:04**	0:06
Zergling	0:42	0:48	**0:25**	0:32
Zergling Speed	4:02	3:54	3:28	**2:49**
Second Hatchery	0:44	0:35	**0:19**	0:21
Hydralisk Den	2:15	1:24	**0:45**	0:52
Lair	1:02	0:55	**0:33**	0:37
Hive	4:40	0:45	**0:39**	0:57
Consume	4:55	0:32	**0:27**	0:55

5.1.6 Conclusion

Players can gain an advantage in RTS games by identifying opponent strategies as they are being executed and developing counter measures. Commentators demonstrate the capability to anticipate strategies and can predict which build order a player is pursuing based on the timing of specific actions performed in the game. This section presents an approach for emulating this capability by collecting thousands of gameplay demonstrations, encoding the actions performed in the games into feature vectors, and training classification and regression models. These models are applied to the tasks of predicting an opponent's build order and timing of production actions. The resulting models exhibit foresight and are capable of identifying which strategy a player is executing with a high confidence.

There are several limitations of my classification approach for recognizing strategies. In order to label the feature vectors, it is necessary to specify a fixed set of

rules and strategies. This prevents the system from tracking the evolving meta-game, where new strategies are constantly discovered by players. An additional issue is that the rule set applies to a specific phase of the game and anticipating opponent actions for different phases in the game requires applying additional rule sets. The models also rely on knowing the precise timings of actions executed by the opponent, which can lead to the classifiers overfitting the data. While the methods presented here help agents to anticipate what to expect, they do not provide ways in which to respond.

5.2 Strategy Learning

Real-time strategy games have an evolving meta-game in which new strategies are constantly being discovered and exploited by players. The motivation for exploring new strategies is to find new strategies that are effective against current strategies that are popular, develop novel build orders that catch opponents off guard, and exploit new maps released by the community. Adaptation is a necessary capability for expert StarCraft gameplay, because the pool of strategies employed by players is constantly changing. Performing well in StarCraft requires not only the ability to learn new strategies as the meta-game evolves, but also the ability to learn to recognize new opponent strategies. This is a form of meta-game adaptation, where players learn new strategies across several gameplay sessions.

My approach for implementing an adaptation mechanism in a StarCraft agent is to use gameplay demonstrations to track the evolving meta-game. As new strategies

are performed by players, new replays are produced containing the actions necessary to execute them. To harness these demonstrations, I use the replays to formulate new strategic goals for the agent to pursue. Using demonstrations to select goals for the agent to accomplish eliminates the need to author specific strategies in the agent, reducing the amount of hard-coded behavior in the system. My approach uses a library of cases to perform goal formulation, which I refer to as case-based goal formulation [106]. I use case-based goal formulation in my system for two tasks: strategy selection and opponent strategy recognition. This approach enables the agent to select new strategic goals at any time based on expert gameplay demonstrations and emulates the adaptation capability shown by professional players.

One of the goals of case-based goal formulation is to overcome many of the limitations of the strategy prediction method presented in the previous section. While the strategy prediction approach was capable of identifying the opponent's strategy with a high degree of confidence early in the game, it relied on domain engineering a rule set for labeling strategies and was limited to a fixed set of strategies during a specific phase of the game. Case-based goal formulation is a more general approach for intent recognition that enables the agent to formulate an anticipation of the opponent's strategic goal at any time during the game. The main disadvantage of case-based goal formulation over the classification approach to strategy recognition is that it is an instance-based method that formulates goals based on retrieving a single gameplay demonstration, while the classification method is trained on a large number of demonstrations.

Case-based goal formulation exploits the temporal structure of demonstrations

to select goals to pursue. The general idea of the approach is to infer game states in a demonstration as the goals of a player. For example, if a player reaches a game state in which air units are produced, my approach infers that during gameplay the player formulated and accomplished a goal of producing air units. One of the advantages of this approach is that it is possible to infer the goals of a player with different planning window sizes, by retrieving game states from different time steps in a demonstration. Game states reached early in the game are goals pursued with a small planning window size, while states reached later are goals with a large planning window size. Case-based goal formulation provides a mechanism for strategy learning, because it can select new goals for the agent to pursue at any time, and it can be applied to the opponent state to anticipate future opponent state.

One of the limitations of this approach is that the agent is limited to executing and anticipating strategies that are included in the case library. This means that the agent's pool of strategies is fixed, unless the case library is actively maintained. Additionally, the fixed strategy pool means that the agent will never execute a novelty build order. Another limitation is that the agent can anticipate only strategies included in the library; if the opponent performs a new build order, the case library may not provide sufficient examples for managing the novelty strategy. One way of overcoming these limitations is to apply methods for maintaining the case library as the strategy space evolves, but this topic is outside the scope of this work.

This section introduces case-based goal formulation and applies it to the task of predicting an opponent's future game state. I present the Trace algorithm, which

uses the temporal structure of demonstrations to infer the goals of a player. The case representation for this approach builds on the feature vector representation from the previous section and also introduces features for tracking the number of different unit types produced during a game. To evaluate the approach, I performed a number of off-line experiments that predict the opponent state with various planning window sizes. As a benchmark, I compare the results to classification based approaches for intent recognition.

5.2.1 Case-Based Goal Formulation

Case-based goal formulation is a technique that retrieves goals from a library of cases. It is similar to sequential instance-based learning [27], but selects goals rather than actions. I define case-based goal formulation as follows:

> The agent's goal state, g, is a vector computed by retrieving the most similar case, q, to the world state, s, and adding the difference between q and its future state, $q\prime$, which is the state after n actions have been applied to q.

Formally:

$$q = \underset{c \in L}{\operatorname{argmin}} \ \operatorname{distance}(s, c)$$

$$g = s + (q\prime - q)$$

where g is a numerical vector representing the goal state, c is a case in the case library, L, and the distance function is a domain independent or domain specific distance metric.

I refer to the number of actions needed to achieve the generated goal state as the *planning window size*. The planning window size is a parameter that specifies how

far to look ahead during the retrieval process. A small planning window is useful for domains where plans are invalidated frequently, while a large planning window should be used in domains that require long-term plans. I discuss tuning this parameter in Chapter 6.

5.2.2 Trace Algorithm

The Trace algorithm is a technique I developed for implementing case-based goal formulation using traces of world state. A trace is a list of tuples containing world states and actions, and is a single episode demonstrating how to perform a task in a domain. For example, a game replay is a trace that contains the current game state and player actions executed in each game frame. The algorithm utilizes a case representation where each case is an unlabeled feature vector that describes the game state at a specific time.

Cases from a trace are indexed using a time step feature. This enables efficient lookup of $q\prime$ once a case, q, has been selected. Assuming that the retrieved case occurred at time t in the trace, $q\prime$ is defined by the world state at time $t+n$. Since the algorithm uses a feature vector representation, g can be computed as follows:

$$q = q_t$$

$$q\prime = q_{t+n}$$

$$g(x) = s(x) + (q\prime(x) - q(x))$$

where x is a feature in the case representation. The Trace algorithm operates by retriev-

ing the most similar case, finding the future state in the trace based on the planning window size, and adding the difference between the retrieved states to the current game state.

Consider an agent with a planning window of size 2, a Euclidean distance function, and the following game state:

$$s =< 3, 0, 1, 1 >$$

There is a single trace, consisting of the following cases:

$$q_1 =< 2, 0, 0.5, 1 >$$

$$q_2 =< 3, 0, 0.7, 1 >$$

$$q_3 =< 4, 1, 0.9, 1 >$$

$$q_4 =< 4, 1, 1.1, 2 >$$

The Trace algorithm would proceed as follows:

1. The system retrieves the most similar case: $q = q_2$
2. $q\prime$ is retrieved: $q\prime = q_{2+n} = q_4$
3. The difference is computed: $q\prime - q =< 1, 1, 0.4, 1 >$
4. g is computed: $g = s + (q\prime - q) =< 4, 1, 1.4, 2 >$

After goal formulation, the agent's goal state is set to g.

One of the ways of overcoming the brittleness of placing too much emphasis on a single trace is to formulate goals from multiple traces. The MultiTrace algorithm is an

extension of the Trace algorithm in which multiple cases are retrieved when formulating a goal state. The technique is similar to k-NN, where the k most similar cases are retrieved. The intention of combining multiple traces for goal formulation is to deal with new situations that may not be present in the case library. The algorithm is defined as follows:

$$w_j = e^{-\text{distance}(s,q_j)}$$

$$\sum_{j=1}^{k} w_j = 1$$

$$g(x) = s(x) + \sum_{j=1}^{k} w_j * (q_j\prime(x) - q_j(x))$$

where w_j is the weight assigned to a case. Each of the k retrieved cases is assigned a weight based on the distance to the current goal state. The weights are then normalized. The cases are combined into a single goal state by multiplying each retrieved case by its weight.

5.2.3 Case Representation

My case representation is a feature vector that tracks the number of units and buildings that a specific player controls. There is a feature for each unit and building type and the value of each feature is the number of that type that have been produced since the start of the game. This approach encodes only a single player's state. The system encodes the agent's state for strategy selection and the opponent's state for opponent modeling.

I collected thousands of professional-level replays from community websites

and converted them to my case representation. Replays were converted from Blizzard's proprietary binary format into text logs of game actions using a third-party tool, as discussed in Section 5.1.2. The resulting case library consists of 1,831 traces and 244,459 cases. The cases are extracted directly from the actions contained in replays and not from recorded game states, which means that this representation does not track unit deaths. One way of capturing complete game state information while generating the case library is to play the replays in StarCraft and extract game state information each time a production event occurs [110].

A subset of an example trace is shown in Table 5.2. An initial case, q_1, is generated with all values set to zero, except for the worker unit type (SCV) and command center type, which are set to 4 and 1 respectively, because the player begins with these units. A new case is generated for each action that trains a unit or produces a building. The value of a new case is initially set to the value of the previous case, then the feature corresponding to the train or build action is incremented by one. Considering a subset of the features (# SCVs, # Supply Depots, # Barracks, # Marines), the example trace would produce the following cases:

$$q_1 =< 4, 0, 0, 0 >$$

$$q_2 =< 5, 0, 0, 0 >$$

$$q_3 =< 5, 1, 0, 0 >$$

$$q_4 =< 6, 1, 0, 0 >$$

$$q_5 =< 6, 1, 1, 0 >$$

$$q_6 =< 6, 1, 1, 1 >$$

The actions in the example trace are *Train SCV*, *Build Depot*, *Train SCV*, *Build Barracks*, and *Train Marine*. The initial case, q_1, corresponds to the starting game state and each new case is generated for each unit or structure produced by the player.

I also explored case representations that include additional features used for case retrieval, but not for computing goal states. These additional features are used in the distance function and enable better situation assessment in the retrieval process. The features are based on the encoding presented in the previous section, where each feature specifies the time that a new unit, structure, or upgrade is first produced. I evaluated feature sets that include the player's timings, the opponent's timings, and both players' timings of actions in addition to the unit count features.

The retrieval process uses Euclidean distance to compute the similarity between different cases. To convert the unit count and timing features to similar ranges, the timing features were converted to correspond to game time in minutes.

5.2.4 Evaluation

I evaluated case-based goal formulation by applying it to opponent modeling in StarCraft. Opponent modeling was performed by applying goal formulation to the opponent's state. Given the opponent's current state, s, an opponent modeling algorithm builds a prediction of the opponent's future state, p, by applying n actions to s. This prediction is then compared against the opponent's actual state n actions later in the game trace, g. All experiments performed computed error using the root mean

squared error (RMSE) between the predicted goal state, p, and the opponent's actual goal state, g.

Experiments used 10-fold cross validation. A modified version of fold-slicing was utilized to prevent cross-fold trace contamination, where cases from the same trace are present in both training and testing datasets. To get around this problem, all cases from a trace are always included in the same fold. I had sufficient training data for the folds to remain relatively balanced.

I compared case-based goal formulation against classification algorithms. The classification case representation contains an action in addition to the goal state, which serves as a label for the case. In the example cases above, q_1 would be labeled with the action *Train SCV* and q_2 would be labeled with the action *Build Depot*. The following algorithm was applied to build predictions with a planning window of size n:

```
p = goal(state g, int n)

        if (n == 0) return g

        else return goal(g + c(g), n-1)
```

where *goal* is the formulation function, $c(g)$ refers to classifying an instance, and $g+c(g)$ refers to updating the goal state by applying the action contained in the case. The goal function runs the classifier, updates the state based on the prediction, and repeats until n classifications have been performed. I evaluated the following algorithms:

- Null - predicts $p = s$ and serves as a baseline.
- IB1 - uses a nearest neighbor classifier [2].

- AdaBoost - uses a boosting classifier [33].
- Trace - uses the Trace algorithm with a Euclidean distance metric.
- MultiTrace - uses the MultiTrace algorithm with a Euclidean distance metric.

Weka implementations were used for the IB1 and AdaBoost classifiers [112].

The first experiment evaluated opponent modeling on various planning window sizes at different stages in the game. The different stages in the game refer to how many train and build actions have been executed by the player so far. Different stages in the game were simulated by building predictions for the cases indexed at a specific time from the traces in the test dataset. Opponent modeling was applied to predicting a Terran player's actions in Terran versus Protoss matches. Results from the first experiment are shown in Figure 5.8. The results show that the Trace and MultiTrace algorithms outperformed the classification algorithms on all planning window sizes. Detailed results for various planning windows sizes are shown in Table 5.7.

The second experiment evaluated the effects of adding additional features to the case representation. The additional features specify the game time in which the player first produces a specific unit type or building type [103]. There is a timing feature for each of the unit count features. The different feature sets include the unit count feature set, the addition of the player timing features (timing), the addition of the opponent timing features (opponent timing), and the addition of both player and opponent timing features (both timing). Results from the second experiment are shown in Figure 5.9. The results show that adding any of the additional feature sets greatly

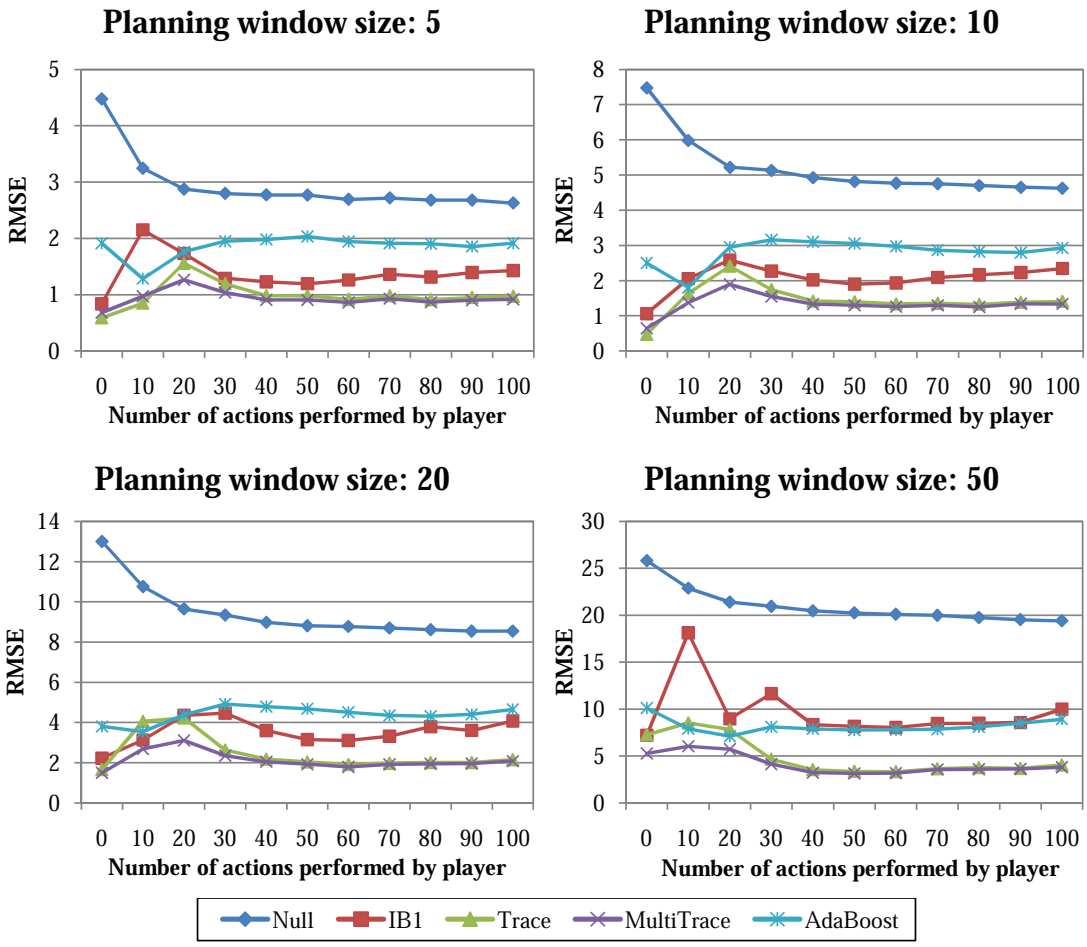

Figure 5.8: The error rates of the algorithms on various planning window sizes show that the Trace and MultiTrace algorithms performed best. In this figure, the horizontal axis refers to the number of train and build actions that have been executed by the player.

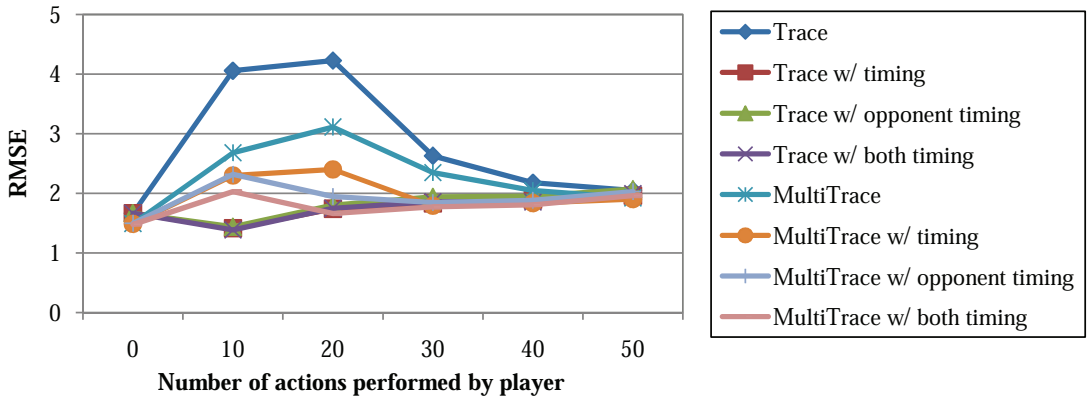

Figure 5.9: The error rates of the Trace and MultiTrace algorithms improved when utilizing additional features. Each of the algorithm were evaluated with four different feature sets that include the original features and additional timing features.

Table 5.7: The average errors of the Trace and classification algorithms on various planning window sizes show that the Trace algorithm with timing features performed best.

	Window Size			
	5	**10**	**20**	**50**
Null	2.94	5.19	9.43	20.9
IB1	1.38	2.06	3.53	9.64
AdaBoost	1.86	2.81	4.40	8.18
Trace	0.98	1.44	2.44	4.87
MultiTrace	0.93	1.32	2.12	4.12
Trace w/ Timing	**0.84**	**1.17**	**1.87**	**3.74**

improves the performance of anticipating opponent actions.

5.2.5 Conclusion

New strategies are constantly being developed in StarCraft, which results in an evolving meta-game. In order to play at an expert level, it is necessary to adapt and learn new strategies as gameplay advances. One way of emulating this capability in an agent is to learn from demonstrations. As new strategies are explored by players,

additional demonstrations can be added to the agent's collection of gameplay examples, enabling the system to learn to execute and anticipate new strategies.

I explored an approach for implementing strategy learning by representing learning as an opponent modeling task, where the goal is to predict the future state of an opponent. I presented the Trace and MultiTrace algorithms, which exploit the temporal structure of demonstrations to anticipate future game states. The algorithms were able to outperform two classification approaches and improved in performance when utilizing additional features that describe the timing of actions. The MultiTrace algorithm performed best in the early stages of the game when strategies are most divergent, while adding timing features resulted in similar performance between the Trace and MultiTrace algorithms. The resulting technique, case-based goal formation, can be used to select goals states for an agent to pursue at any time during the game and to anticipate the future state of the opponent with various planning window sizes.

The current version of the system uses a batch process to learn a collection of strategies from demonstrations. In order to track the evolving meta-game of StarCraft, the demonstration library needs to be updated as new strategies are popularized by players. This process involves case library maintenance, which my system does not capture. One way of implementing this capability is to apply methods for identifying unusual games [26], and updating the library as necessary.

5.3 State Estimation

StarCraft enforces imperfect information through a *fog-of-war* that limits a player's visibility to portions of the map where units are controlled. To acquire additional game state information, players actively scout opponents to uncover the locations of enemy forces and bases. Observations of enemy forces are used by players to identify candidate locations to attack and determine which locations need to be defended. Professional players demonstrate estimation capabilities, by tracking opponent forces and moving units into defensive positions before attacks are launched.

Tracking opponent forces is a necessary capability for expert RTS gameplay. In order to work towards realizing this capability in an agent, I investigate the task of maximizing the amount of information available to an agent given game state observations. To accomplish this goal, I propose a particle-based approach for tracking the locations of enemy units that have been scouted. My approach is inspired by the application of particle filters to state estimation in games [10]. It includes a movement model for predicting the trajectories of units and a decay function for gradually reducing the agent's confidence in predictions. One of the challenges in RTS games is accurately identifying the location and size of opponent forces, because opponents may have multiple, indistinguishable units.

To select parameters for the particle model, I represent state estimation as a function optimization problem. To implement this function, I mined a corpus of expert StarCraft replays to create a library of game states and observations, which are used to

evaluate the accuracy of a particle model. This process uses gameplay demonstrations in order to perform model optimization. Representing state estimation as an optimization problem enabled off-line evaluation of several types of particle models. My approach uses a variation of the simplex algorithm to find near-optimal parameters for the particle model. This section provides an overview of the particle model, update process, and model training approach as well as experiments that evaluate that ability of the model to track previously observed enemy units.

5.3.1 Particle Model

The goal of my model is to accurately track the positions of enemy units that have been previously observed. My approach for achieving this task is based on a simplified model of particle filters. I selected a particle-based approach instead of a space-based approach based on several properties of RTS games. It is difficult to select a suitable grid resolution for estimation, because a tile-based grid may be too fine, while higher-level abstractions may be too coarse. Also, the model needs to be able to scale to hundreds of units. Finally, the particle model should be generalizable to new maps.

The particle model is inspired by particle filters, but instead uses a single particle to track the position of a previously encountered enemy unit, instead of a cloud of particles. A single particle per unit approach was chosen to simplify the culling process, because an opponent may have multiple, indistinguishable units. Since an agent is unable to identify individuals across multiple observations, the process for culling candidate target locations becomes non-trivial. A side effect of using a cloud

of particles for tracking indistinguishable units is that the agent may overestimate the threat of a region, and this problem is present in a single particle formulation as well. I address this problem by adding a decay function to particles, which gradually reduces the agent's confidence in estimations over time.

Particles in the model are assigned a class, weight, and trajectory. The class corresponds to the unit type of the enemy unit. My system includes the following classes of units: building, worker unit, ground attacker, and air attacker. Each class has a unique set of parameters used to compute the trajectories and weights of particles.

Particles are assigned a weight that represents the agent's confidence in the prediction. A linear decay function is applied to particles in which a particle's weight is decreased by the decay amount each update. Particles with a weight equal to or less than zero are removed from the list of candidate target locations. Different decay rates are used for different classes of particles, because predictions for units with low mobility are likely to remain accurate, while predictions for units with high mobility can quickly become inaccurate.

Each particle is assigned a constant trajectory, which is computed based on a linear combination of vectors. A visualization of the different vectors is shown in Figure 5.10. The movement vector is based on observed unit movement, which is computed as the difference between the current coordinates and previous coordinates. The model also incorporates a chokepoint vector, which enables terrain features to be incorporated in the trajectory. It is found by computing the vectors between the unit's coordinates and the center point of each chokepoint in the current region, and selecting the vector

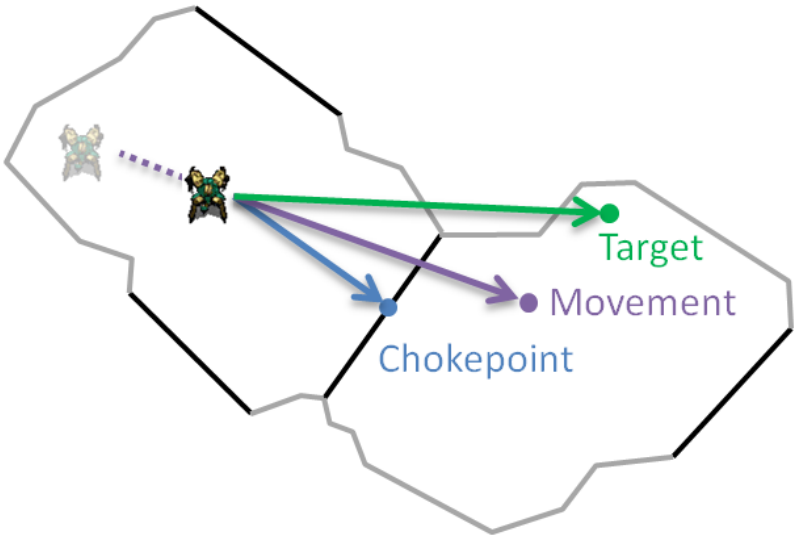

Figure 5.10: Particle trajectories are computed using a linear combination of vectors. The movement vector captures the current trajectory, while the target vector factors in the unit's destination and the chokepoint vector factors in terrain.

with the smallest angle with respect to the movement vector. The target vector is based on the unit's destination and is computed as the difference between the destination coordinates and current unit coordinates. Computing the target vector requires accessing game state information that is not available to human players.

The trajectory of a particle is computed by normalizing the vectors to unit vectors, multiplying the vectors by class-specific weights, and summing the resulting vectors. My model incorporates unique movement and target weights for each particle class, while a single weight is used for the chokepoint vector.

5.3.2 Update Process

The particle model begins with an initially empty set of candidate target locations. As new units are encountered during the course of a game, new particles are

spawned to track enemy units. The model update process consists of four sequential steps:

- **Apply movement:** updates the location of each particle by applying the particle's trajectory to its current location.
- **Apply decay:** linearly decreases the weight of each particle based on its class specific decay weight.
- **Cull particles:** removes particles that are within the agent's vision range or have a less than zero weight.
- **Spawn new particles:** creates new particles for units that were previously within the agent's vision range that are no longer within the agent's vision range.

The spawning process instantiates a new particle by computing a trajectory, assigning an initial weight of one, and placing the particle at the enemy unit's last known position.

Unlike previous work, the model does not perform a normalization process, because multiple units may be indistinguishable. Additionally, my model does not commit to a specific sampling policy. The process for determining which particles to sample is left up to higher-level agent behaviors.

A visualization of the model at different phases throughout a game is shown in Figure 5.11. In the figure, green dots are the locations of player units, red dots are the locations of enemy units, and blue dots are estimations of enemy unit locations. During the early stages of the game, the player sends a scouting unit to the opponent's

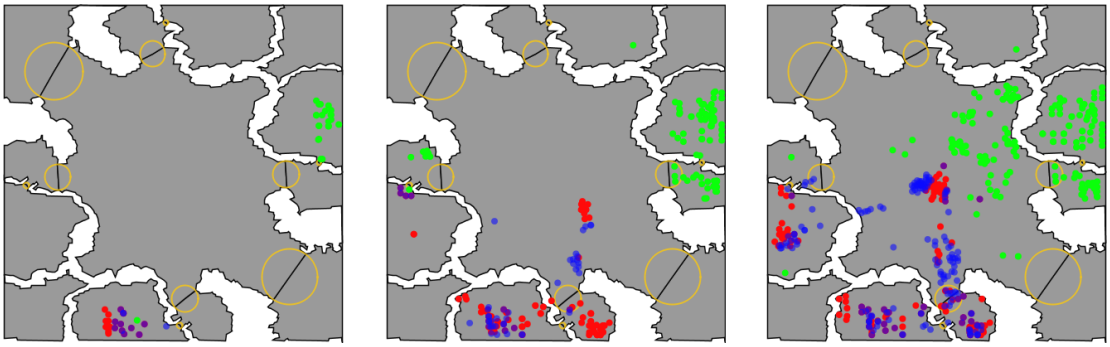

Figure 5.11: The particle model tracks units that have been observed by scouting and attacking forces. This figure shows the estimations of enemy forces during the early game (left), mid game (middle), and late game (right) phases.

base, which identifies the location of the majority of the opponent's units. As the game progresses, both players expand to new locations and the players begin attacking each other. The estimations during the middle of the game show that the model is aware of the opponent's forces in the middle of the map, but not the opponent expansion. During the late stages of the game, a majority of the map is covered by the players. While the model does not provide exact locations of enemy forces, in this example the model is able to identify the relative army strength in each of the map regions.

5.3.3 Model Training

I explored the application of particle models to StarCraft by performing off-line analysis. The goal of this work was to determine the accuracy of different model settings and to find near-optimal trajectory and decay parameters for the models. The parameters in the particle model are shown in Table 5.8. To evaluate the models, I collected a corpus of StarCraft replays, extracted game state and observation data from

Table 5.8: The parameters in the model include decay rates and movement weights for four different classes of units. The values shown here are values selected for Protoss versus Zerg games.

	Decay Rate
Building	0.00
Worker	0.00
Ground Attacker	0.04
Air Attacker	0.13
	Movement Vector
Building	0.00
Worker	5.67
Ground Attacker	5.35
Air Attacker	31.57
	Chokepoint Vector
All Classes	20.96

the replays, and simulated the ability of the models to predict the enemy threat in each region of the map at each time step.

To enable off-line analysis of particle models, I collected thousands of expert-level StarCraft replays. Replay collection is described in more detail in Section 5.1.1. I sampled the replays by randomly selecting ten replays for each unique race match up. An additional constraint applied during the sampling process was that all replays in a sample were played on distinct maps. This constraint was included to ensure that the particle models are applicable to a wide variety of maps.

I extracted game state information from the sampled replays by viewing them using the replay mode of StarCraft and querying game state with the Brood War API. My replay tool outputs a dump of the game state once every 5 seconds (120 frames), which contains the positions of all units. The extracted data provides sufficient infor-

mation for determining which enemy units are visible by the player at each time step. The resulting data set contains an average of 2,852 examples for each race match up.

I explored a region-based metric for state estimation in StarCraft, where the role of the particle model is to predict the enemy threat in each region of the map. The error function makes use of the Brood War Terrain Analyzer, which identifies regions in a StarCraft map [82]. The particle model is limited to observations made by the player, while the error function is based on complete game state.

Error in state estimation can be quantified as the difference between predicted and actual enemy threat. The particle model predicts the enemy threat in each region based on the current game state and past observations. For each region, the enemy threat is computed as the number of visible enemy units in the region (unit types are uniformly weighted), plus the summation of the weights of particles within the region. Given predictions of enemy threat at time step t, state estimation error is computed as follows:

$$\text{error}(t) = \sum_{r \in R} |p(r,t) - a(r,t)|$$

where $p(r,t)$ is the predicted enemy threat of region r at time step t, $a(r,t)$ is the actual number of enemy units present in region r at time step t, and R is the set of regions in a map. The actual threat for a region can be computed using the complete information available in the extracted replay data. The overall error for a replay is defined as follows:

$$\text{error} = \frac{1}{T} \sum_{t=1}^{T} \text{error}(t)$$

where T is the number of time steps, and error is the average state estimation error.

My proposed particle model includes several free parameters for specifying the trajectories and decay rates of particles. To select optimal parameters for the particle model, I represent state estimation as an optimization problem: the state estimation error serves as an objective function, while the input parameters provide a many-dimensional space. The set of parameters that minimizes the error function is selected as optimal parameters for my particle model.

To find a solution to the optimization problem, I applied the Nelder-Mead technique [70], which is a downhill simplex method. I used Michael Flanagan's minimization library[2] which provides a Java implementation of this algorithm. The stopping criterion for my parameter selection process was 500 iterations, which provided sufficient time for the algorithm to converge.

5.3.4 Evaluation

I compared the performance of the particle model with a baseline approach as well as a perfect prediction model. The range of values between the baseline and theoretical models provides a metric for assessing the accuracy of my approach. I evaluated the following models:

- **Null Model:** A particle model that never spawns particles, providing a baseline for worst-case performance.
- **Perfect Tracker:** A theoretical model which perfectly tracks units that have been previously observed, representing best-case performance.

[2] `http://www.ee.ucl.ac.uk/~mflanaga/java`

- **Default Model:** A model in which particles do not move and do not decay, providing a last known position.
- **Optimized Model:** The particle model with weights selected from the optimization process.

The null model and perfect tracker provide bounds for computing the accuracy of a model. Specifically, I define the accuracy of a particle model as follows:

$$\text{accuracy} = \frac{\text{error}_{\text{NullModel}} - \text{error}}{\text{error}_{\text{NullModel}} - \text{error}_{\text{PerfectTracker}}}$$

where error is the state estimation error. Accuracy provides a metric for evaluating the ability of a particle model to estimate enemy threat.

The accuracy of the default and optimized models for each of the race match ups are shown in Table 5.9. A race match up is a unique pairing of StarCraft races, such as Protoss versus Terran (PvT) or Terran versus Zerg (TvZ). The table also includes results for variations of the particle models which were provided with additional features. The Default$_I$ and Optimized$_I$ models were capable of identifying specific enemy units across observations, and the Optimized$_T$ model used the target vector while other models did not. Accuracies for the null model and perfect tracker are not included, because these values are always 0 and 1. Overall, the optimized particle model, which is limited to features available to humans, performed best. Providing additional information to the particle models did not, on average, improve the accuracy of models.

There are several factors that result in different accuracies across the race match ups. One of the biggest influences on the accuracy of the particle model is the

Table 5.9: The accuracies of the different particle models varies based on the specific race match up. Overall, the optimized particle model performed best in the off-line state estimation task. Providing the particle models with additional features, including the target vector (T) and ability to distinguish units (I), did not improve the overall accuracies.

	PvP	PvT	PvZ	TvP	TvT	TvZ	ZvP	ZvT	ZvZ	Avg.
Default	0.75	0.74	0.69	0.57	0.78	0.83	0.63	0.68	0.36	0.67
Default$_I$	0.75	0.73	0.71	**0.72**	0.76	0.767	0.68	0.70	0.56	0.71
Optimized	**0.84**	**0.77**	**0.73**	0.71	**0.81**	**0.83**	**0.70**	**0.72**	0.54	**0.74**
Optimized$_I$	0.75	0.73	0.71	**0.72**	0.76	0.77	0.68	0.70	**0.56**	0.71
Optimized$_T$	**0.84**	0.74	0.71	0.71	**0.81**	**0.83**	**0.70**	**0.72**	0.54	0.73

average game time for the particular match up. Zerg versus Zerg games frequently have the shortest game lengths in tournament play, and the particle models performed worst on this match up. Another factor in the different race match ups is army mobility. Terran players tend to group units into large forces and slowly push across the map, while Zerg and Protoss armies are more mobile. A third factor is different units that provide vision of the map. Zerg players can scout the map using Overlords, while Terran players are less reliant on scouting units. These different factors result in different accuracies across the optimized and default particle models for the race match ups.

One of the limitations of the error measurement is that it aggregates the performance of each model over complete games. I also investigated the variation in accuracy of different models over the duration of a game, which provides some insights into the scouting behavior of players. The average threat prediction errors for the different models in Terran versus Protoss games is shown in Figure 5.12. In this race match up, there was a noticeable difference between the accuracies of the default and optimized models. In the early stages of the game there is no difference between the models, because no

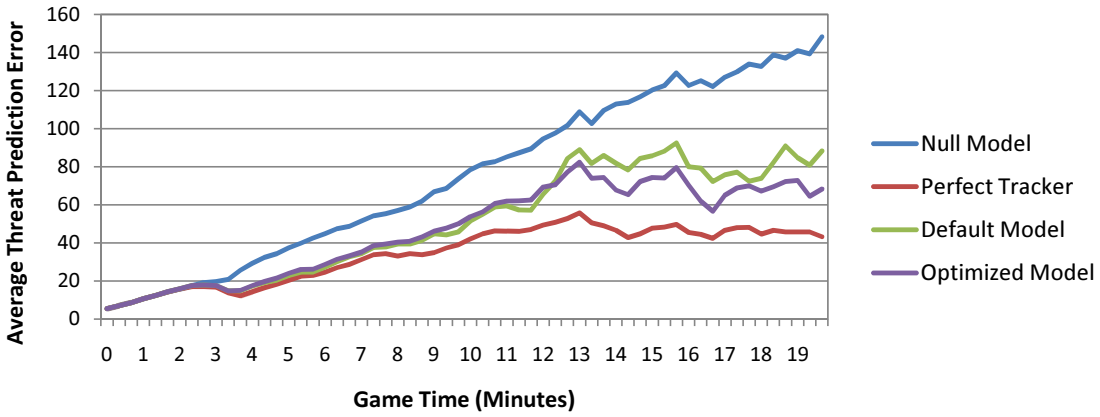

Figure 5.12: The average error of the particle models in Terran versus Protoss games vary drastically over the duration of a game. The accuracy of the particle models improve over baseline performance once enemy units are scouted. The optimized particle model noticeably outperforms the default particle model after 12 minutes.

enemy units have been observed. Players tend to scout the opponent between three and four minutes game time, which leads to improved state estimations. There is little difference between the default and optimized particle models in the first 12 minutes of the game, but the optimized model is noticeably more accurate after this period.

5.3.5 Conclusion

Players use estimations of opponent force locations to determine where to defend and identify candidate locations for assaulting opponents. Tracking opponent forces is a necessary capability for expert-level gameplay. To emulate this capability in my system, I developed a particle model that estimates the locations of opponent units that have been observed. The model contains parameters for movement and decays weights for different units classes, and values for these parameters are selected by representing state estimation as an optimization problem.

In order to evaluate the accuracy of different parameter settings, I extracted game state observations from professional replays and simulated the ability of different models to predict the locations of enemy units. This process uses demonstrations as a way to perform model optimization. To provide a baseline for comparing different models, I introduced the perfect tracker, which is a model with a priori knowledge of unit movement. Another result is that the output of the models show general patterns in StarCraft gameplay, such as the initial scout timing of professional players.

5.4 Summary

StarCraft gameplay requires a combination of deliberative and reactive reasoning capabilities. To perform at an expert level, it is necessary to build anticipations of opponent actions in order to formulate counter measures. Three capabilities demonstrated by professional players are anticipation, adaptation, and estimation. To realize these in an agent, I have explored different approaches for learning from gameplay demonstrations. Each of these approaches applies to a single scale of gameplay, and uses the demonstrations in a unique way.

To emulate the anticipation capability demonstrated by players, I used demonstrations to train classification and regression models that perform build order prediction and timing prediction tasks. The motivation for this approach was that plotting the timing distribution of individual features hinted at different opponent strategies. To train the different models, I converted thousands of replays into a feature vector representa-

tion that captures a player's expansion of the tech tree. The classification algorithms were evaluated against the rule set used to label the vectors, and the results showed that several classification models outperformed the rule set. Therefore, the classification models exhibited foresight in predicting opponent strategies. The main limitations of this approach are that it requires hand-authoring a rule set for labeling strategies and it predicts strategies for a specific phase of the game.

To emulate the adaptation capability demonstrated by players, I presented case-based goal formulation. This technique uses a collection of demonstrations in order to formulate goals for an agent to pursue and can be applied to the task of anticipating the future state of an opponent. Rather than hand-authoring specific goals for the agent to pursue, the system learns strategies from a collection of demonstrations. In case-based goal formulation, demonstrations are converted to a case library and used in a case retrieval process that exploits the temporal structure of demonstrations. The motivation for this approach is that the game state reached in later stages of a game can be inferred as a player's goals earlier in the game. One of the advantages of this approach is that it can be used at any time during a game. I evaluated case-based goal formulation on the task of predicting future opponent state and the results show that it outperformed classification algorithms.

To emulate the estimation capability demonstrated by players, I explored a particle model for tracking observed enemy units. The particle model is motivated by the application of particle filters to state estimation in games, but I used a single particle per unit approach due to unique challenges presented in RTS games. The resulting

model has several free parameters for managing the movement and decay functions of particles. To select suitable parameters for the model I represented state estimation as an optimization problem and used demonstrations to implement an error function for evaluating different model settings. In this process, demonstrations are used to provide examples of gameplay that enable off-line analysis of different model settings. To evaluate the different models, I compared the optimized particle model with a perfect tracker, which provided a lower bound of prediction error. The results showed that the optimized model outperformed the default model in off-line experiments.

Each of the methods presented in this chapter use demonstrations in order to build a model of gameplay. One of the limitations of these methods is that a batch process is used for training. In order for these approaches to adapt to the evolving meta-game of StarCraft, it is necessary to rebuild the models with additional demonstrations as gameplay changes.

Chapter 6

Integrating Learning

One of the central challenges in building an agent capable of expert level RTS gameplay is supporting heterogeneous reasoning capabilities in an integrated system. StarCraft requires both reactive and deliberative decision making, and professional players also demonstrate adaptation, anticipation, and estimation capabilities during gameplay. A system that reproduces these capabilities needs to not only support each of these processes, but to integrate them as well.

In the previous chapters I explored how existing AI methodologies can be used to support many of the capabilities necessary for RTS gameplay. Reactive planning provides mechanisms that support authoring multi-scale agents, and case-based reasoning and machine learning can be used to build models of gameplay from demonstrations. The outcome of this work is an agent that plays complete games of StarCraft with a fixed strategy, and models for anticipating the actions of an opponent, estimating the locations of opponent units, and formulating strategic goals from gameplay examples. To

integrate these capabilities in EISBot, my approach uses working memory to interface reactive planning with external components that generate plans and formulate goals.

There are a number of ways in which the ABL reactive planner can be interfaced with external reasoning processes. One way of interfacing ABL with other components is by adding additional facts to the agent's working memory. Facts generated from external components can be used for condition checks in ABL behaviors, such as using estimates of opponent locations in attack behaviors. ABL can also be interfaced with components that perform plan generation and goal formulation outside the scope of the reactive planner. In this configuration, ABL is used to manage the active goals of the agent and execute plans, while goals can be generated by the reactive planner or external components. I interface ABL with an external plan generation process to realize deliberative reasoning capabilities in EISBot.

While these integration approaches enable the reactive planning agent to interface with external components, each of the learned gameplay models is used in isolation of each other. To facilitate coordination between the different components in the system, I investigate an instantiation of the goal-driven autonomy (GDA) conceptual model [65], which outlines a number of subtasks within an agent. These subtasks include monitoring expectations, detecting discrepancies, generating explanations, and formulating goals. My system implements these tasks using examples learned from gameplay demonstrations. Using the GDA model enables the agent to integrate processes that anticipate the actions of an opponent, detect when new opponent strategies are being pursued, and formulate goals in response to opponent strategies.

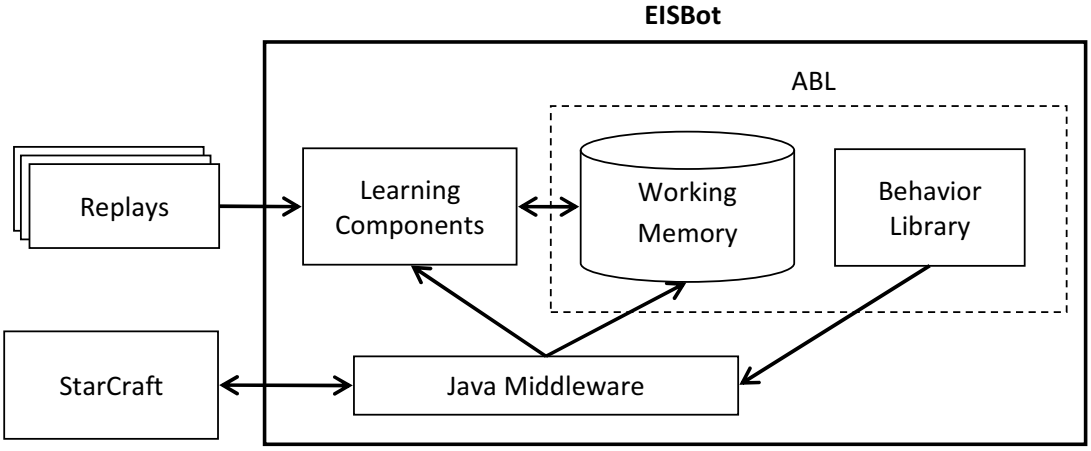

Figure 6.1: The agent architecture includes the ABL reactive planner, a Java middleware layer, and learning components. The learning components take a collection of replays as input and communicate with ABL through the agent's working memory. The Java middleware manages synchronization with StarCraft, sends game state updates to working memory and the learning components, and performs actions selected for execution by ABL.

This chapter presents methods for integrating reactive planning with external components. I start by providing an overview of the agent architecture. I then identify design patterns used to interface EISBot with gameplay models learned from demonstrations. Building on this work, I present an instantiation of the GDA model that integrates anticipation and goal formulation capabilities in EISBot.

6.1 Agent Architecture

The components in EISBot include the ABL reactive planner, a Java middleware layer, and learning components. An overview of the agent architecture is shown in Figure 6.1. The agent runs as a separate process from StarCraft and interfaces with the game through the use of a shared-memory bridge, which is presented in Section 4.3.1.

The core of the agent is implemented in the ABL reactive planning language, as a collection of hand-authored behaviors. The structure of the ABL behavior library is based on the integrated agent framework of McCoy and Mateas [61], which decomposes RTS gameplay into distinct tasks and includes a manager for performing each of these tasks. The design of the ABL agent is discussed in more detail in Section 4.3.

The Java middleware layer performs several functions. It is responsible for communicating with the StarCraft process and managing synchronization. The layer also implements sensors and actions that enable the ABL agent to interact with StarCraft. On game update events, the layer retrieves the current game state from StarCraft and updates data structures used by WME sensors and the learning components. At the end of the update, it dispatches unit orders that have been selected for execution by ABL behaviors. The middleware layer provides utility functions that can be queried by mental acts and condition checks in ABL behaviors, such as testing if a location is suitable for constructing a structure. Another function the layer provides is additional physical actions that are not directly supported via the game interface, such as ordering a squad of units to attack a target location. It also notifies other components in EISBot when a game update has occurred.

The learning components in EISBot interface with the middleware layer, working memory, and a collection of replays. Each of the learning components in the system operate independently and coordinate using messages in working memory. EISBot includes learning components that track estimations of opponent units, select strategic plans for the system to execute, and perform GDA subtasks. These components are

discussed in more detail in Section 6.2.1 and Section 6.2.2. These components interface with working memory by querying for the existence of WMEs, posting WMEs to memory, and consuming WMEs. The learning components are given a collection of replays that have been converted into the necessary data format. For most of these components, a training phase is performed offline before the start of games in order to build a gameplay model. The learning components do not select actions for execution directly, and instead delegate action execution and monitoring to ABL.

The components in EISBot use working memory as a blackboard [36], enabling external components to coordinate with ABL behaviors and query active goals. In addition to storing facts about the world, the working memory of an ABL agent includes the expansion of the active behavior tree. A variety of event-based and polling approaches are used in EISBot to facilitate coordination between ABL behaviors and the learning components.

6.2 Integration Approaches

EISBot integrates heterogeneous components using multiple integration approaches. These include adding elements to working memory, formulating goals to pursue, generating plans to execute, and providing additional conditions for behavior activation. I have used these approaches to integrate the ABL reactive planner with case-based reasoning, goal-driven autonomy, and machine learning components. These components interface with ABL behaviors through working memory design patterns.

While the methods presented here are used in a reactive planning agent, the integration approaches could be applied to other agent architectures with working memories.

One way of interfacing with external components in a reactive planner is to utilize the multi-scale idioms presented in Section 4.4. The message passing and behavior locking patterns are used to interface external components with working memory. It is also possible for external components to replace ABL managers, by implementing a manager contract. External components that interact with working memory directly, rather than through the use of sensors, need to be synchronized with the ABL decision cycle in order to avoid race conditions.

This section presents methods used in EISBot to integrate ABL with learning components. These methods are visualized in Figure 6.2. To integrate state estimation, the particle model augments working memory with estimates of opponent locations. To integrate strategy learning, ABL behaviors execute and monitor plans generated by external components. To integrate goal-driven autonomy, the agent pursues goals spawned by external processes and uses behaviors activated by external components.

6.2.1 Augmenting Working Memory

The gameplay models learned from demonstration can be integrated in EISBot by supplementing the agent's working memory with additional beliefs. These beliefs can be estimates about the game state, such as the location of opponent units, or anticipations of opponent actions, such as a prediction of the opponent build order. A belief can be added to the agents working memory by instantiating a WME and

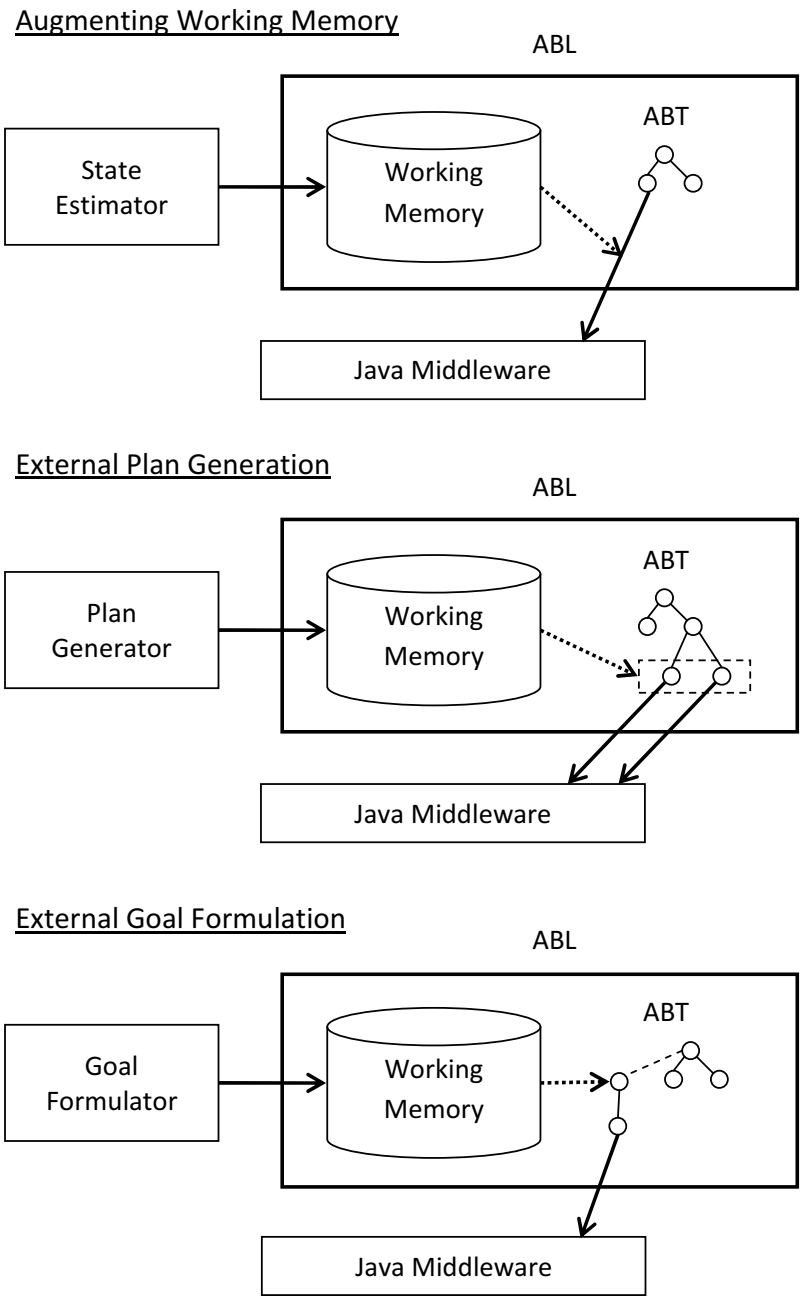

Figure 6.2: External components are integrated in EISBot using ABL's working memory. The state estimator adds additional facts to the agent's working memory that are used for parameterizing physical actions. The plan generator adds production requests to working memory that result in sequences of physical actions. The goal formulation adds requests to working memory that result in the agent spawning additional goals to pursue.

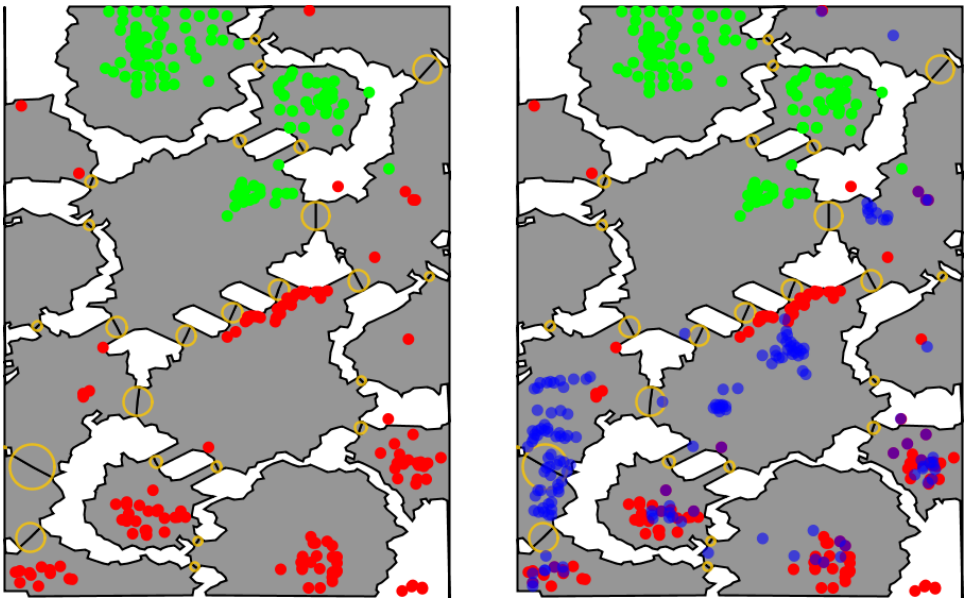

Figure 6.3: The agent's perspective of the game state is augmented with predictions of opponent locations, shown in blue. The image on the left shows an agent with complete vision of the map and the image on the right shows estimates of unit locations based on previously observed opponent units.

placing it in working memory. WMEs added to working memory by external components can be used in the condition checks of ABL behaviors including preconditions, context conditions, and success tests. In order to utilize additional WME types added to working memory, it is necessary to author additional ABL behaviors that operate on these beliefs.

One of the learning components that utilizes this integration approach is the state estimation particle model. The input to the particle model is observations of enemy unit locations, and the output of the model is estimates of enemy unit locations. A visualization of the inputs and outputs of the particle model in an example game are shown in Figure 6.3. Each enemy unit is tracked by a particle, which stores the unit type and location, applies a movement model, and manages a decay function. The

particle model is notified of game updates by the Java middleware layer, which also provides a list of enemy units that are currently visible in the game.

The state estimation component maintains a list of candidate enemy locations, represented by the particles in the model. The component instantiates a ParticleWME for each of the particles in the model, which contains the location of the particle and the unit type it is tracking. The ParticleWMEs are added to the agent's working memory through the use of a ParticleWME sensor, which queries the particle model for the list of candidate locations. Using a sensor object to modify working memory ensures that updates are synchronized with the ABL decision cycle.

EISBot includes scouting and tactical ABL behaviors that operate on ParticleWMEs. The particles provide candidate locations to attack and scout based on the estimations tracked by the model. One of the tasks that uses particles is target selection. A subset of the behaviors that implement target selection in EISBot are shown in Figure 6.4. These behaviors accomplish the goal `attack` for a specified unit and have different specificities. The behavior with the higher specificity will be selected for expansion if there are enemy units visible to the agent. If there are no enemy units visible, then the lower specificity behavior will be selected for expansion, which selects a particle location to attack. In this configuration, the agent will attack enemy units if they are visible and if no units are visible, attack locations are chosen using the particle model. The particle model is also used in EISBot to activate scouting behaviors. If no particles are present in working memory past a specific game time, the agent spawns the goal of scouting base locations.

```
sequential behavior attack(PlayerUnitWME unit) {
  precondition {
    (EnemyUnitWME x::x y::y)
  }
  specificity 2;

  act attackMove(unit, x, y);
}

sequential behavior attack(PlayerUnitWME unit) {
  precondition {
    (ParticleWME x::x y::y)
  }
  specificity 1;

  act attackMove(unit, x, y);
}
```

Figure 6.4: EISBot includes two behaviors for achieving the `attack` goal. The first behavior selects an attack target based on the location of an enemy unit currently in the agent's field of view, while the second behavior selects a target using a predicted enemy location.

6.2.2 External Plan Generation

While ABL is well suited for managing the execution of an agent, it lacks deliberative planning capabilities. One way of realizing deliberative reasoning capabilities in an ABL agent is to use external components to generate plans and apply ABL to the task of plan execution and monitoring. Plan monitoring is an important task in RTS games, because most actions are performed over a duration and actions can fail for a number of reasons. EISBot uses external components to select strategic plans to execute, which generate sequences of production actions. At the start of the game, a plan is selected from a collection of hand-authored build orders. Later in the game, new plans are generated using case-based goal formulation [106].

The external planning process replaces a collection of ABL behaviors respon-

sible for build order selection. In the initial version of EISBot, all of the logic for selecting which production actions to perform was implemented as hand-authored behaviors. These behaviors can be easily replaced with the external planning process, because message passing idioms are used to separate behaviors that select production actions to perform and behaviors that handle the details of executing production actions. The planning component implements a subset of the strategy manager and is responsible for build order selection.

The output of the external planning components is a sequence of production actions to perform. These actions include constructing buildings, training units, and researching upgrades. The planner creates a production request WME for each of the actions in the generated plan. These actions are added to working memory and consumed by ABL behaviors responsible for handling these requests. The ABL daemon behavior responsible for handling external production requests is shown in Figure 6.5. When production requests are placed in working memory, the manager retrieves the requests, pursues a subgoal for accomplishing the request, and then deletes the request upon completion. The agent includes multiple behaviors for achieving the `processPlanRequest` goal, which can involve training units, constructing buildings, and researching upgrades.

6.2.3 External Goal Formulation

Another way EISBot interfaces with components is by pursuing goals that are selected outside of the reactive planning process. Goals that are formulated external to the agent can be added to the reactive planner by modifying the structure of the active

```
sequential behavior buildOrderManager() {
  with (persistent) subgoal processPlanRequests();
}

sequential behavior processPlanRequests() {
  PlanRequestWME request;

  with (success_test {
    request = (PlanRequestWME)
  }) wait;

  subgoal processPlanRequest(request);

  mental_act {
    BehavingEntity.getBehavingEntity().deleteWME(request);
  }
}
```

Figure 6.5: The build order manager is a daemon behavior responsible for processing production actions selected by external planning components. When production requests are placed in working memory, the manager retrieves the requests, executes the production action, and then deletes the request.

behavior tree, or by triggering ABL's spawngoal functionality, which causes the agent to pursue a new goal in parallel with the currently active goals.

I have interfaced ABL with external goal formulation components using the message passing design patterns. The goal formulator determines when to pursue a specific goal, and upon activation posts a WME to working memory that requests a goal to be pursued. This WME is consumed by an ABL behavior responsible for processing external requests and spawning goals. An ABL behavior that performs this functionality for the `attack` goal is shown in Figure 6.6. Structuring the agent in this way enables the conditions for launching attacks to be decided by hand-authored ABL behaviors or external components.

My approaches for integrating external plan generation and goal formulation

```
sequential behavior monitorAttackRequests() {
  AttackGoalWME request;

  with (success_test {
    request = (AttackGoalWME)
  }) wait;

  mental_act {
    BehavingEntity.getBehavingEntity().deleteWME(request);
  }

  spawngoal attack();
}
```

Figure 6.6: Goals selected for activation by external components are spawned by ABL behaviors that implement the message consumer pattern. In this example, an external component formulates the goal of attacking and adds an attack request to working memory, while the ABL behavior consumes the request and spawns the `attack` goal.

are quite similar. Both of these approaches are implemented as behaviors that consume requests from working memory. The main difference is that behaviors that consume plan actions pursue the actions inline as a subgoal, while behaviors that consume requests for goal pursuit spawn new threads of execution for pursuing the goal. These approaches are similar in EISBot, because performing production tasks involves several behaviors that monitor the execution of production actions.

6.2.4 Behavior Activation

ABL agents can be interfaced with components that manage the activation conditions of behaviors. ABL behaviors contain zero or more preconditions, which specify activation conditions. These conditions can contain procedural preconditions [77], which query external components for activation. In EISBot, procedural preconditions are used to query the Java middleware layer for candidate locations to construct

buildings.

Behavior activation is similar to the behavior locking design pattern presented in Section 4.4.4, but enables a behavior for expansion rather than disabling it. Another way to implement behavior activation in ABL is to incorporate WME condition checks in a behavior, where the WMEs are managed by an external component. This approaches differs from the plan generation and goal formulation patterns, because the behaviors using the WME for activation do not consume the WME. Instead the external component is responsible for posting and removing the activation WME from working memory. In EISBot, behavior activation is used to enable behaviors that build detection units, which occurs when external components anticipate that the opponent is building cloaking units.

6.3 Goal-Driven Autonomy

Expert-level StarCraft gameplay requires integrating multiple reasoning capabilities in order to anticipate the actions of opponents, identify when strategies being pursued are no longer effective, and to formulate responses to opponent strategies. In the previous section I presented methods for interfacing external components with reactive planning, but each of these components worked in isolation of each other. In order to emulate the behavior of expert players, it is necessary to facilitate tighter coordination across these external components. The ability to anticipate, assess, and respond to opponent strategies involves decision making across multiple gameplay scales. To

realize these capabilities in EISBot, I apply the goal-driven autonomy (GDA) model.

The GDA conceptual model provides a framework for creating agents capable of responding to unanticipated failures during plan execution in complex, dynamic environments [65]. It is motivated by Cox's claim that agents should reason about their goals in order to continuously operate with independence [23]. The conceptual model specifies subtasks that enable an agent to detect, reason about, and respond to unanticipated events. However, it makes no commitment to specific algorithms.

The GDA model extends Nau's model of online planning [69] by identifying specific subtasks within the controller of an agent, as shown in Figure 6.7. The controller interacts with an execution environment that provides observations of world state and a planner that generates sequences of actions to achieve the agent's current active goal. In order to identify when planning failures occur, a GDA agent requires the planning component to generate an expectation of world state after executing each action in the execution environment.

A GDA agent starts by passing an initial goal to the planner. The planner generates a plan consisting of a set of actions, a, and expectations, p. As actions are performed in the execution environment, the discrepancy detector checks if the resulting world state, s, matches the expected world state. When a discrepancy is detected between the expected and actual world states, the agent creates a discrepancy, d, which is passed to the explanation generator. Given a discrepancy, the explanation generator builds an explanation, e, of why the failure occurred and passes it to the goal formulator. The goal formulator takes an explanation and formulates a goal, g,

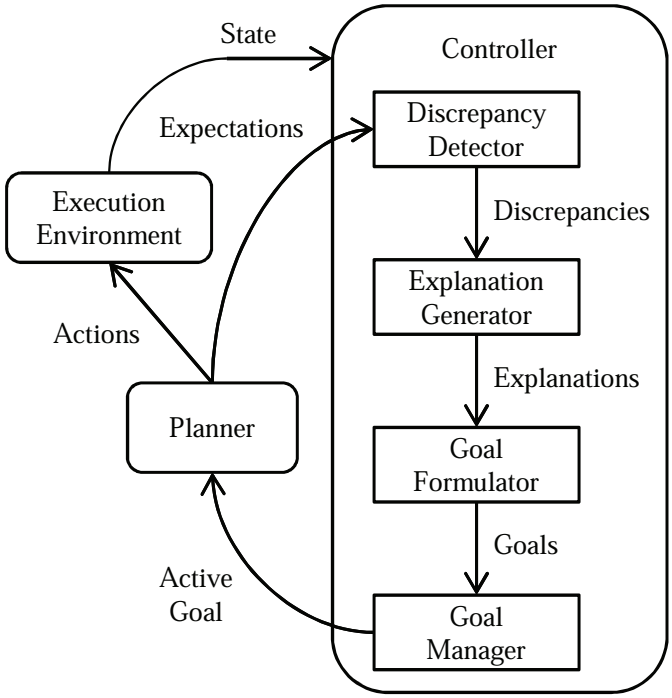

Figure 6.7: A GDA agent interacts with a planner and execution environment, and incorporates a controller with subtasks for monitoring plan execution.

in response to the explanation. The goal is then passed to the goal manager, which is responsible for selecting the agent's active goal.

I have applied the GDA model to the task of monitoring strategy execution in StarCraft. The subtasks in the model identify when the current strategy becomes invalidated due to the actions of the opponent and formulates a new strategy for the agent to pursue. I have explored two versions of the GDA model. In the first version, all of the GDA subtasks are implemented as hand-authored ABL behaviors. This version applies the GDA model in order to monitor strategies, but lacks integration with the external learning components. In the second version, three of the GDA subtasks are im-

plemented using case-based goal formulation. This version monitors strategy execution and integrates external learning components.

6.3.1 Hand-Authoring Rules

One of my goals in applying the GDA model is to decouple goal selection from goal execution logic in EISBot. The motivation for this decoupling is to enable external learning components to select goals for the agent to pursue. In the first implementation of the GDA model, I authored a collection of ABL behaviors that perform the discrepancy detection, explanation generation, and goal formulation tasks [105]. This approach is similar to previous work that implemented the GDA model using hand-authored production rules [68]. One of the key distinctions of this approach from previous work was that EISBot does not model expectations. Instead, the agent has a collection of discrepancy detection behaviors that are always active.

The discrepancy detector generates discrepancies when the agent's expectations are violated. Discrepancies serve the purpose of triggering the agent's goal reasoning process and provide a mechanism for responding to unanticipated events. EISBot generates discrepancies in response to detecting the following types of game events:

- **Unit Discrepancy:** opponent produced a new unit type.
- **Building Discrepancy:** opponent built a new building type.
- **Expansion Discrepancy:** opponent built an expansion.
- **Attack Discrepancy:** opponent attacked the agent.

- **Force Discrepancy:** there is a shift in force sizes between the agent and opponent.

The discrepancies are intentionally generic in order to enable the agent to react to a wide variety of situations. EISBot uses event-driven behaviors to detect discrepancies. An example behavior for detecting new units is shown in Figure 6.8. The `detect` behavior has a set of preconditions that checks for an enemy unit, binds its type to a variable, and checks whether a discrepancy for the unit type currently exists in working memory. If there is not currently a unit type discrepancy for the bound type, a mental act is used to place a new discrepancy in working memory.

The explanation generator takes as input a collection of discrepancies and outputs explanations. Given a discrepancy, zero or more of the following explanations are generated:

- **Cloaking:** opponent is building cloaked units.
- **Expanding:** opponent is expanding.
- **Air Units:** opponent is building air units.
- **Force Advantage:** opponent has force advantage.

The explanation generator is implemented as a set of behaviors that apply rules of the form: if d then e. An example behavior for generating explanations is the `explain` behavior shown in Figure 6.8. The behavior checks if the opponent has upgraded to a Lair and constructed a Hydralisk Den, which indicates that the opponent may be

producing units capable of cloaking. In response to detecting these conditions, the agent creates an explanation that the opponent is building units capable of cloaking.

The goal formulator spawns new goals in response to explanations. Given an explanation, one or more of the following goals are spawned:

- **Execute Strategy:** selects a strategy to execute.
- **Expand:** builds an expansion and trains worker units.
- **Attack:** attacks the opponent with all combat units.
- **Retreat:** sends all combat units back to base.

Goal formulation behaviors implement rules of the form: if e then g. EISBot contains two types of goal formulation behaviors: behaviors that directly map explanations to goals, as in the example of mapping an enemy cloaking explanation to the goal of building detector units, and behaviors that select among one of several goals in response to an explanation. An example goal formulation behavior is shown in Figure 6.8. The behavior spawns goals to retreat and build detectors in response to an explanation that the agent is producing cloaking units.

Using the GDA model enables the system to monitor the execution of strategies and to respond if the current strategy becomes invalidated. The main limitation of this approach is that all discrepancies, explanations, and goals have to be hand authored. Additionally, this approach did not reason about expectations and therefore was not capable of integrating anticipations of opponent actions. However, implementing the

```
parallel behavior gdaManager() {
  subgoal detect();
  subgoal explain();
  subgoal formulate();
}

// Discrepancy Detection
sequential behavior detect() {
  precondition {
    (EnemyUnitWME type::type)
    !(DiscrepancyWME type==type)
  }

  mental_act {
    BehavingEntity.getBehavingEntity().addWME(new DiscrepancyWME(type));
  }
}

// Explanation Generation
sequential behavior explain() {
  precondition {
    (DiscrepancyWME type==HydraliskDen)
    (DiscrepancyWME type==Lair)
    !(ExplanationWME type==Cloaking)
  }

  mental_act {
    BehavingEntity.getBehavingEntity().addWME(
      new ExplanationWME(Cloaking));
  }
}

// Goal Formulation
sequential behavior formulate() {
  precondition {
    explanation = (ExplanationWME type==Cloaking active=true)
  }

  spawngoal buildDetectors();
  spawngoal retreat();

  mental_act {
    explanation.setActive(false);
  }
}
```

Figure 6.8: I authored a collection of ABL behaviors for implementing the GDA subtasks in EISBot. The `detect` behavior generates discrepancies when new enemy unit types are observed. The `explain` behavior generates an explanation that the opponent is building cloaking units. The `formulate` behavior spawns goals for the agent to pursue in response to the cloaking explanation.

GDA subtasks in ABL enabled a decoupling between the goal section and goal execution logic in the agent, which can be exposed to external components.

6.3.2 Learning GDA Subtasks

I also explored an implementation of the GDA conceptual model that uses external learning components to perform the GDA subtasks. Rather than requiring a domain expert to specify expectations, explanations, and goals, my approach learns how to generate these objects from examples. The GDA subtasks are implemented using variations of case-based goal formulation [106]. Using the GDA model to integrate external learning components enables the agent to incorporate anticipation and adaptation capabilities.

This approach is specific to adversarial domains, in which planning failures or exogenous events occur due to the actions of other agents. I apply two case libraries: the opponent library, AL, provides examples for opponent actions and is used for intent recognition by performing goal formulation on the opponent's state, while the goal library, GL, is used to select goals for the agent to pursue. By using case-based goal formulation to retrieve goals from the adversarial case library, the agent can build predictions of the future actions of an opponent. I refer to the number of actions, m, that the retrieval mechanism looks ahead at the opponent's actions as the *look-ahead window*. This differs from the *planning window size*, which is the number of actions to look ahead when generating plans.

The GDA components utilize a numerical feature vector representation that

describes game state. The representation includes features for describing the agent's world state and a single opponent's world state. Additional discussion of the representation is presented in Section 5.2.3. The agent uses the same case library for both the opponent case library and the goal library. When retrieving cases using the opponent case library, AL, the *distance* function uses a subset of features, which describe opponent state. This feature subset is used by the expectation builder and explanation generator. The system generates expectations for features that correspond to changes in opponent state, and explanations are computed based on changes in opponent state. When retrieving cases using the goal library, GL, the complete feature set is used by the *distance* function. During goal retrieval, the system computes the goal state based on changes in agent state, while ignoring opponent state.

The methods presented here for generating expectations, explanations, and goals from demonstrations build on the Trace algorithm presented in Section 5.2.2. Expectations are used for strategy prediction, and are generated by identifying new units and structures produced in retrieved examples. To build expectations, the Trace algorithm is applied to the opponent game state. Explanations are also used for strategy prediction, and are generated by applying the Trace algorithm to goal formulation using the opponent game state. Goals are used to select new strategies for the agent to pursue, and are generated by applying the Trace algorithm to the agent game state. An example of the different applications of the Trace algorithm for the GDA subtasks is presented in Section 6.3.2.5.

6.3.2.1 Building Expectations

I define an expectation, p, as an anticipated change in a single world state feature caused by an opponent's actions. Specifically, expectations are beliefs that the opponent will perform an action that introduces a new unit type, x into the game at a specific time, τ:

$$p(x, \tau) : s_t(x) > 0, t \geq \tau$$

where s is the game state and x is a feature that describes the opponent's state. When an expectation is generated, the agent anticipates the presence of unit type x in the execution environment after time τ.

In order to build expectations, the system retrieves the most similar case from the opponent case library and identifies times in which new units are introduced into the game. The system triggers expectation generation when a new active goal is selected by the goal manager. A set of expectations, π, is generated as follows:

$$q = \underset{c \in AL}{\text{argmin}} \ \text{distance}(s, c)$$

$$\pi = \{p(x, \tau) : \tau = \min_{v}[q_v(x) > 0, v > t]$$

$$| \ \forall x, q_t(x) = 0, q_{t+m}(x) > 0\}$$

The retrieval process occurs at time t with a look-ahead window of size m. It generates an expectation for each feature describing an opponent unit type, x, in the retrieved case, q, which changes from a zero value to a positive value between time t and time $t + m$. The time selected for the expectation is the smallest time step, τ, where the

feature has a positive value. Each expectation represents an anticipated change in a single opponent feature at a specific time.

I use a modified version of the Trace algorithm to build expectations in EISBot. The Trace algorithm computes a new goal for the agent to pursue by finding the difference between two different time steps in a retrieved case. An overview of the Trace algorithm is presented in Section 5.2.2, and an example of expectation generation is presented in Section 6.3.2.5. In order to build expectations, I modified the retrieval process to identify new unit types produced by the opponent between these two time steps. The earliest time step that a new unit type is produced by the player is selected as the time, τ, to anticipate the unit type, x.

6.3.2.2 Detecting Discrepancies

A discrepancy, d, is defined as an expectation that is not met. This situation occurs when an opponent does not perform actions causing an anticipated state change. The system creates a discrepancy when the following conditions are met:

$$p(x, \tau), s_t(x) = 0, t \geq \tau + \delta$$

where δ is a discrepancy period, which provides a buffer period for observing state changes. This situation occurs when an unit of type x has not been observed in the game before time $\tau + \delta$.

There are two causes of discrepancies in this formulation. In the first case, the opponent does not produce the anticipated unit type during the discrepancy period. This can result from the opponent changing strategies or from the agent forming an

incorrect prediction. In the second case, the opponent produces the anticipated unit type during the discrepancy period, but the agent does not observe the unit type. This situation can occur due to the fog-of-war in StarCraft.

6.3.2.3 Generating Explanations

EISBot generates explanations when the active goal completes or a discrepancy is detected. It creates explanations by applying goal formulation to the opponent, using the opponent case library. I define an explanation, e, as a prediction of future world state resulting from the execution of an opponent's actions. The agent uses explanations to attempt to identify how the opponent will change world state. This definition differs from previous work in GDA, which uses explanations in order to identify why discrepancies occur using current world state [65]. Both my approach and previous work use explanations as input to the goal formulation subtask. The agent uses case-based goal formulation to create explanations as follows:

$$q = \underset{c \in AL}{\operatorname{argmin}} \ \operatorname{distance}(s, c)$$

$$e = s + (q_{t+m} - q_t)$$

where AL is the opponent goal library, m is the number of opponent actions to look ahead, and e is an anticipated future world state. An example of explanation generation is shown in Section 6.3.2.5.

The agent uses explanations in order to incorporate predictions about adversarial actions into the goal formulation process. This approach differs from previous

work [68], in which explanations are used to explain the cause of discrepancies. One of the reasons for this difference is that actions performed in StarCraft usually do not immediately invalidate a strategy. Additionally, the agent does not have complete state information and therefore it may not be possible to identify the cause of a discrepancy in StarCraft. My approach uses demonstrations to reduce the amount of domain engineering required to implement the GDA subtasks, while trading off the explanation capabilities of the agent.

The main difference between expectations and explanations in EISBot is that an explanation is an estimation of the future opponent state, while an expectation is an anticipation of a specific opponent action. Expectations are used to trigger strategy invalidation in the agent, because if an expected unit type is not produced by the opponent, it is likely that the predicted strategy of the opponent is incorrect. Expectations provide a way to monitor the execution of an opponent's actions and track anticipated opponent actions.

6.3.2.4 Formulating Goals

I define a goal, g, as a future world state for the agent to achieve. The input to the goal formulation component is the current world state and an explanation, e, which contains the anticipated future world state of the opponent. Goal formulation is triggered when the agent's active goal completes or a discrepancy is detected. The

agent formulates goals as follows:

$$q = \underset{c \in GL}{\text{argmin}}\ \texttt{distance}(e, c)$$

$$g = s + (q_{t+n} - q_t)$$

where GL is the goal case library and the input to the case retrieval similarity metric is an explanation of future world state. The output of the formulation process is a goal that is passed to the goal manager, which is responsible for selecting the agent's active goal. The goal formulator can also be used without the explanation generator, by providing the current game state as input to the retrieval process.

In EISBot, the goal formulator selects plans for the strategy manager to execute. To implement plan selection, the Trace algorithm retrieves the most similar trace, and returns the sequence of actions performed in the trace. The output of this process is a strategic plan that enables the agent to pursue the goal state selected by the goal formulation process. Previous GDA implementations have used separate goal formulation and plan generation processes.

6.3.2.5 GDA Example

The GDA subtasks are invoked when the agent selects an initial goal, the current goal is accomplished, or a discrepancy is detected. During this invocation, the expectation generation subtask selects a new set of expectations to monitor, the explanation generator creates an anticipation of future opponent state, and the goal formulator produces a new goal for the agent to accomplish. Consider an agent with a

look-ahead window of size 3, a Euclidean distance function, and the following opponent game state:

$$s =< 5, 0, 0, 0 >$$

In this representation, the features describe enemy unit counts for workers, depots, barracks, and marines. The adversary case library, AL, is a single replay, consisting of the following cases:

$$q_1 =< 4, 0, 0, 0 >$$

$$q_2 =< 5, 0, 0, 0 >$$

$$q_3 =< 5, 1, 0, 0 >$$

$$q_4 =< 6, 1, 0, 0 >$$

$$q_5 =< 6, 1, 1, 0 >$$

$$q_6 =< 6, 1, 1, 1 >$$

During invocation, the following subtasks are performed. First, expectation generation would proceed as follows:

1. The system retrieves the most similar case: $q = q_2$
2. $q\prime$ is retrieved: $q\prime = q_{2+m} = q_5$
3. New unit types are identified: depot @ $t = 3$, barracks @ $t = 5$
4. Expectations are generated: $\pi = [p(\text{depot}, 3), p(\text{barracks}, 5)]$

The result of the expectation generation process is that the agent anticipates that the opponent will produce a depot at time $t = 3$ and a barracks at time $t = 5$. These expectations remain active until a discrepancy is detected or the GDA process is invoked again. Second, explanation generation would proceed as follows:

1. The system retrieves the most similar case: $q = q_2$
2. $q\prime$ is retrieved: $q\prime = q_{2+m} = q_5$
3. The difference is computed: $q\prime - q =< 1, 1, 1, 0 >$
4. e is computed: $e = s + (q\prime - q) =< 6, 1, 1, 0 >$

The result of the explanation generation process is that the agent anticipates that the opponent will perform actions to reach the opponent state of $< 6, 1, 1, 0 >$.

Third, goal formulation is performed by the agent using the current game state and explanation as input. Consider the following goal case library, GL, which includes features for describing the player state and the opponent state. The first four features correspond to player workers, depots, barracks, and marines, and the second four features correspond to opponent workers, depots, barracks, and marines:

$$p_3 =< 7, 0, 0, 0, 5, 1, 0, 0 >$$

$$p_4 =< 7, 0, 1, 0, 6, 1, 0, 0 >$$

$$p_5 =< 7, 0, 1, 0, 6, 1, 1, 0 >$$

$$p_6 =< 8, 0, 1, 0, 6, 1, 1, 0 >$$

$$p_7 =< 8, 0, 1, 1, 6, 1, 1, 1 >$$

The goal formulation process selects the most similar case using both the current player state and explanation, while selecting a new goal using only the difference in the player state. Given a player state of $< 7, 0, 1, 0 >$, and a planning window of size 2, goal formulation would proceed as follow.

1. The system retrieves the most similar case: $q = p_5$
2. $q\prime$ is retrieved: $q\prime = p_{5+n} = p_7$
3. The difference is computed for the player state: $q\prime - q =< 1, 0, 0, 1 >$
4. g is computed: $g = s + (q\prime - q) =< 8, 0, 1, 1 >$

The result of this process is that the agent selects $< 8, 0, 1, 1 >$ as the new goal state to accomplish. During the goal retrieval process, the Trace algorithm also retrieves the actions performed in the trace. In this example, the retrieved plan is to train a worker unit and then train a marine. While executing these actions, the agent has expectations for the opponent to produce a depot and to produce a barracks.

6.4 Implementation

I have used the reactive planning idioms and integration approaches presented in this thesis to build a complete game-playing StarCraft agent that incorporates estimation, adaptation, and anticipation capabilities. EISBot integrates hand-authored reactive planning behaviors with gameplay models learned from demonstrations. The

system is a multi-scale agent that performs concurrent tasks across different scales of gameplay, which work towards the goal of defeating all opponents. The system builds on previous work for authoring multi-scale agents [61, 109], and extends it in a number of ways.

The integration approaches are applied in EISBot to incorporate gameplay models learned from demonstrations. The augmenting working memory pattern is used by the particle model, which adds candidate unit locations to working memory. The external plan generation pattern is used by the GDA subtasks, which select strategic plans for the agent to pursue and trigger replanning if the current plan is invalidated. The external goal formulation and behavior activation patterns are used by the agent to respond to anticipated opponent actions, such as building detector units in response to cloaked units. Each of these approaches integrates a learned model at a single scale, while the supporting reactive planning behaviors operate across multiple scales.

In the current version of the system, EISBot plays only the Protoss race. I focused on a single race, because each race has different gameplay characteristics and requires substantially different behavior libraries. To reduce the domain engineering effort of building a complete game-playing agent, I authored a behavior library specifically for the Protoss race. My motivation for selecting Protoss was to limit the number of competencies that needed to be implemented in order to investigate learning capabilities in an agent. Generally, Protoss is less susceptible to early aggression than Terran or Zerg, which reduces the amount of authoring necessary to build a system capable of surviving until the mid-game phase. In order to explore capabilities such as strategy adaptation,

it is necessary for matches to progress into the later stages of the game. While the behavior library is specific to the Protoss race, a number of the learning capabilities are applicable to the other races. The main challenge in applying EISBot to play a different StarCraft race is authoring the ABL behavior library to support race-specific gameplay.

The EISBot implementation interfaces the ABL reactive planner with external components that implement the GDA subtasks and track enemy units. The components in the system are shown in Figure 6.9. The reactive planner manages gameplay across all gameplay scales and is responsible for executing actions selected by the external components. The GDA subtasks manage strategy selection and monitoring, and operate at the strategy scale, while the particle model manages state estimation and operates at the reconnaissance scale. Each of the components operates asynchronously and coordinates through working memory. The resulting system integrates estimation, adaptation, and anticipation capabilities across multiple scales.

6.4.1 Behavior Library

The core of EISBot's functionality is implemented in the ABL planning language. An ABL agent selects actions to perform and goals to pursue by retrieving behaviors from the behavior library. The behavior library is a hand-authored collection of behaviors that specify how to accomplish goals. EISBot's top-level goal is accomplished by a root behavior that spawns a number of managers. Each manager actively pursues a number of subgoals, which are accomplished through further subgoaling. The roles of the different managers are described in Section 4.3. A visualization of a subset

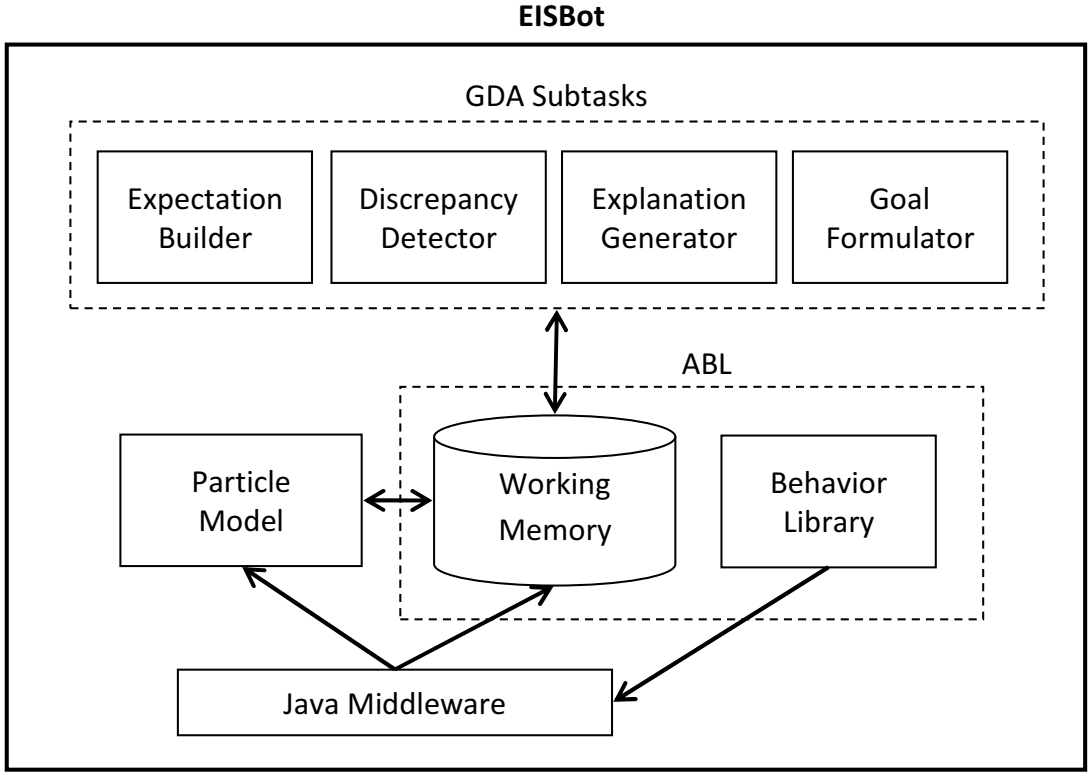

Figure 6.9: The components in the EISBot implementation include the ABL reactive planner, a particle model for state estimation, and GDA subtasks for strategy selection and monitoring. The external components operate asynchronously from ABL and coordinate using working memory. All decisions made by external components are executed through ABL behaviors.

of EISBot's goal hierarchy is shown in Figure 6.10. It shows that the agent's top-level goal is accomplished by a number of managers which further decompose subgoals into specialized tasks. A number of managers in EISBot are implemented as behaviors that encode rules-of-thumb, such as worker production. McCoy and Mateas identify several of these rules [61].

The ABL managers can operate independently of the external components. In the base agent configuration, ABL behaviors are responsible for all decision making aspects of gameplay. In configurations with external components, the ABL behaviors that duplicate tasks performed by external components are disabled using the behavior locking pattern. Additional details on the different configurations of EISBot are presented in Section 7.1.

One of the issues that arises when authoring an ABL agent is determining when to stop authoring additional behaviors. Expert StarCraft gameplay requires a variety of different strategies and tactics, and a large number of behaviors need to be authored to in order to encode these actions. Capturing all expert gameplay behavior in StarCraft, for even small tasks, is a substantial authorial burden. My approach for authoring the behavior library was to first develop a base agent capable of playing complete games, and then focus on subtasks that require additional attention. This section describe managers in EISBot that involved substantial authoring effort.

The *Strategy Manager* is responsible for strategy selection, which involves deciding which buildings to construct, units to produce, and upgrades to research. During start up, the agent selects a build order from a collection of opening strategies. The

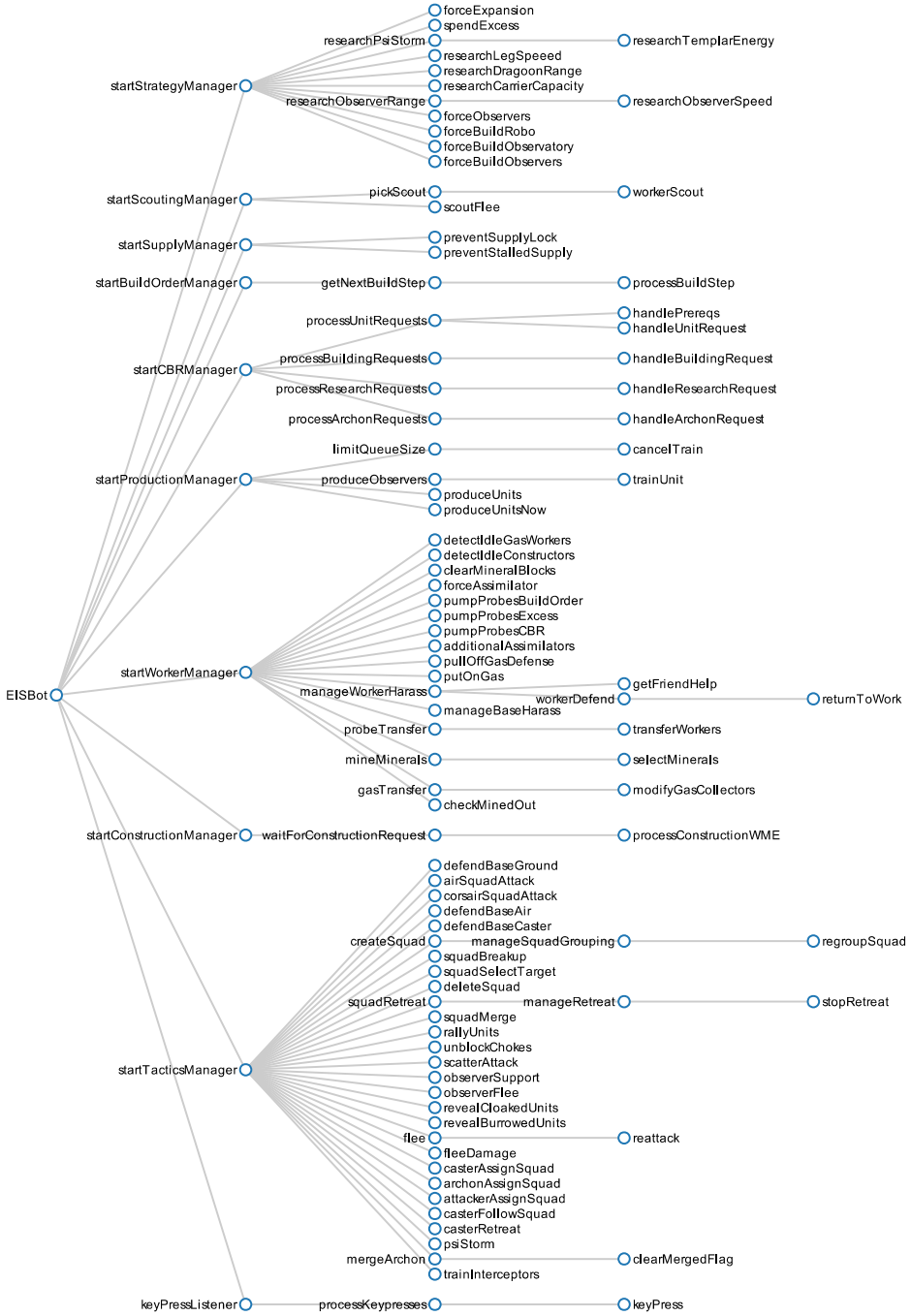

Figure 6.10: The system decomposes gameplay into subtasks, where each task is handled by a manager. This visualization shows a subset of EISBot's goal hierarchy. The ABT instantiated during runtime is much larger, because new goals are spawned during gameplay and some goals are pursued recursively.

build order contains a sequence of production actions, where each production action is indexed by a supply count. The manager also includes behaviors that are performed if the agent accumulates excess resources. A majority of the strategy manager is disabled when the GDA subtasks are used, because strategic plan selection is performed by the external components.

The primary task of the *Tactics Manager* is to create and manage squads of combat units. The manager forms a new squad when a target number of idle combat units are present in the game. Each squad is managed independently and attacks and retreats as a group. Squads are also monitored to ensure that the units do not get too spread out while moving to destinations. A squad formed during gameplay is shown in Figure 6.11. An attacking squad is ordered to retreat if the enemy threat in the immediate area is too large. Squads retreat toward one of the agent's bases and can be merged with newly formed squads. The manager also handles caster and support units, such as high templar and observers. Support and caster units are not assigned to squads, but are assigned orders to follow squads. Caster units perform spells when enemy units are in proximity, and observers move to reveal nearby cloaked units. Micromanagement behaviors are performed by the tactics manager and implemented using the unit subtask pattern. However, the manager does not support transport units.

The *Income Manager* is responsible for worker units, which perform a variety of different tasks. Worker units are used for gathering resources, constructing structures, scouting, and defending. The unit subtask pattern is used to temporarily assign worker units to new tasks. In order to prevent worker units from being assigned conflicting

tasks, the manager tracks the current task of each worker and follows a set of rules that defines which tasks can be interrupted. The manager also includes behaviors for producing additional worker units. The manager continues to produce worker units at a base until a quota is met, or resources are depleted.

The *Reconnaissance Manager* performs scouting behaviors and tracks enemy locations. The manager interacts with the particle model in order to determine if scouting is necessary and to find candidate locations to attack. The manager performs scouting by requesting a worker unit from the income manager and assigning it a set of locations to visit. As the scouting unit encounters enemy units, new enemy unit observations are added to the particle model. A visualization of the particle model during gameplay is shown in Figure 6.12. The particles track the locations and types of enemy units. The manager also implements scattering behaviors for combat units if there are no candidate locations to attack.

A subset of the behaviors in EISBot are used to constrain the actions selected by external components. One of the constraints enforced by the production manager is that duplicate technology buildings are not produced. This situation can occur during external plan retrieval and is generally a wasteful utilization of resources, because producing a duplicate technology structure does not enable new unit types, buildings, or upgrades. One of the reasons that the retrieval mechanism results in duplicate structures is that gameplay demonstrations are applied to new game situations, and the agent retrieves from multiple demonstrations during a single game. Another constraint enforced by the income manager is requiring a minimum time between expansions, to

Figure 6.11: Combat units in EISBot are assigned to squads, which attack and retreat as a group. Support units are not assigned to squads directly, and instead follow active squads.

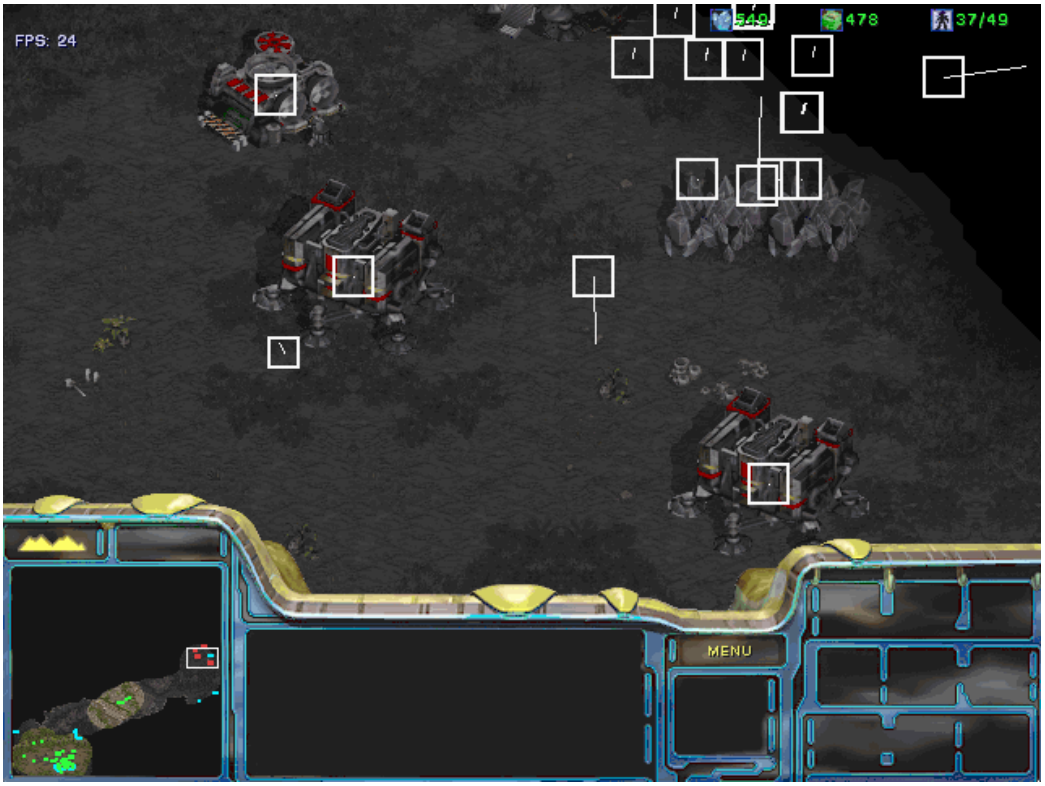

Figure 6.12: The agent maintains candidate enemy locations using a particle model. These estimations include the locations of structures and enemy units.

ensure that the agent does not invest excessive resources in expanding. These behaviors enable EISBot to interface with learned gameplay models, while avoiding duplicate actions that would decrease the gameplay performance of the system. Applying these constraints is a form of knowledge engineering, where the types of actions retrieved for execution are constrained based on background knowledge of StarCraft gameplay.

6.4.2 Goal-Driven Autonomy

The agent starts a game by selecting an initial strategic goal, known as the build order, which becomes the agent's active goal. In the current implementation, the initial goal is selected by the strategy manager, but can also be specified via a Java system property. The active goal is passed to the goal manager, which is implemented in the ABL reactive planning language [105]. The different components in the system communicate using the reactive planner's working memory as a blackboard [36]. The reactive planner is responsible for performing the actions in the game necessary to achieve the goal state. EISBot retrieves these actions directly from traces, rather than using a generative planner.

EISBot invokes the GDA subtasks upon completion of the initial goal. During this process, the agent generates an explanation of the opponent's future game state and builds expectations of anticipated unit and building types. For example, the agent may expect the opponent to construct a *Starport*, and train a fleet of *Valkyrie* aircraft. This anticipated game state is passed to the goal formulator, which selects the next strategic goal for the agent to pursue. In this example, the agent would retrieve similar

game situations in which air units are produced by the opponent and formulate a goal to counter air units. The retrieved goal state is then passed to the goal manager, becoming the current active goal, and the expectation of an opponent *Starport* is passed to the discrepancy detector.

After formulating a new goal and set of expectations, the agent enters a plan monitoring phase. During this process, the agent executes actions to pursue the active goal and checks for discrepancies. For example, if the agent observes an opponent *Starport*, then the expectation is validated and the agent continues to pursue its current goal. However, if the agent does not observe an opponent *Starport* after a discrepancy period has expired, then the expectation becomes invalid, triggering a discrepancy. Upon detecting a discrepancy or achieving the current goal, the agent invokes the explanation generation subtask. Because StarCraft enforces partial observability through a *fog-of-war*, it is possible for the system to incorrectly detect discrepancies due to insufficient vision of the opponent state.

There are three free parameters in the GDA implementation: the planning window size, n, specifies the size of plans resulting from goal formulation, the look-ahead window size, m, specifies how far to look-ahead at opponent actions, and the discrepancy period, δ, specifies a buffer period for detecting discrepancies. In order to find suitable values for these parameters, I ran an ablation study in which GDA subtasks are individually integrated into the agent.

6.5 Summary

This chapter explores methods for integrating gameplay models learned from demonstration with reactive planning. The agent architecture includes the ABL reactive planner, a Java middleware layer, and learning components. I present four ways to interface external components with ABL: augmenting working memory, external plan generation, external goal formulation, and behavior activation. Each of these integration approaches uses ABL's working memory to facilitate communication between different components in the agent. The resulting system, EISBot, incorporates a library of hand-authored ABL behaviors, a particle model for tracking enemy units, and an implementation of the GDA conceptual model for strategy selection and monitoring.

In order to support coordination between different learning components in the system, I explored two versions of the GDA model. In the first version, each of the GDA subtasks is implemented as a collection of hand-authored ABL behaviors. In the second version, each of the GDA subtasks is implemented using models learned from demonstrations. Applying the GDA model enables the agent to build expectations of opponent actions, detect when discrepancies occur, generate explanations of future opponent actions, and formulate new goals to pursue. The GDA subtasks are used for selecting strategies for EISBot to pursue and to detect when new strategies need to be formulated.

Chapter 7

Evaluation

The main objective of this thesis is to identify the capabilities necessary for expert StarCraft gameplay and to realize these capabilities in a game-playing agent. I argue that players demonstrate estimation, adaptation, and anticipation capabilities during gameplay, and have applied AI methodologies to perform each of these tasks. In addition to emulating these individual capabilities, a complete game-playing agent needs to also integrate these capabilities in real time. My hypothesis is that expert StarCraft players utilize several distinct reasoning capabilities during gameplay. In order to evaluate my hypothesis, I make the following claim:

> Reproducing expert-level StarCraft gameplay requires integrating heterogeneous reasoning capabilities.

To test this claim, I evaluate ablations of EISBot against human and computer opponents.

In order to determine the performance of EISBot with respect to human experts, the system is evaluated in complete games of StarCraft. The BWAPI interface

that EISBot uses resembles the GUI presented to human players, and provides the same affordances. The agent is not given any additional game state information that is not presented to players, but the agent is capable of perceiving all map state information without the need to pan the screen. I use two metrics for evaluating the performance of EISBot. Versus computer opponents, win ratio is used to evaluate the performance of the system. One of the problems in using this metric is that different pools of opponent bots result in different win ratios for the system. To account for this problem, versus human opponents a ladder score is used to evaluate EISBot. The ladder score provides an intrinsic rating, which is described in Section 7.2.

A number of EISBot experiments are presented in previous work [106, 105, 107]. Each of these experiments focused on integrating an additional component with the reactive planner. One of the limitations of this analysis is that EISBot was evaluated against only computer opponents. This limitation of evaluating against only computer opponents is a general characteristic of prior RTS research, and therefore related work makes no claims of human-level performance. Additionally, with the exception of the evaluation of different particle models [107], my previous experiments provided the agent with complete game state information. In more recent work, I evaluated EISBot against human opponents while enforcing imperfect information [108], but did not incorporate the learning GDA subtasks. The experiments presented in this section evaluate the complete EISBot system, integrated with external learning components, in an imperfect information environment, against human and computer opponents.

This chapter presents two sets of experiments that evaluate the performance of EISBot. In the first experiment, ablations of EISBot are tested against other StarCraft bots. The first set of experiments are also used to select parameters for components in the GDA subtasks. In the second experiment, ablations of EISBot are tested against human opponents on a StarCraft ladder server. The results of these experiments are used to compare the performance of EISBot with expert players.

7.1 Ablation Studies

To evaluate my claim that StarCraft requires integrating distinct reasoning capabilities, I performed ablation studies of EISBot in which external reasoning components are disabled. The goal of these experiments is to demonstrate that integrating additional reasoning capabilities into the agent improves its win ratio. The first study evaluates different configurations of the particle model, while the second study evaluates different configurations of the GDA subtasks. In these studies, EISBot is evaluated against computer opponents that execute relatively fixed strategies. These experiments are used to evaluate EISBot, as well as select parameters for the GDA components.

There are two sources of computer opponents for StarCraft: the built-in AI, and bots developed by AI researchers and hobbyists. One of the motivations for third-party StarCraft AI is the StarCraft AI Competition that is organized as part of the AIIDE conference[1]. Over the past two years, over two dozen bots have been submitted to the competition, resulting in a collection of bots available for analysis. For the

[1] `http://www.StarCraftAICompetition.com`

experiments presented here, I evaluated EISBot against the following bots from the 2011 StarCraft AI competition: *Aiur*, *SPAR*, *Cromulent*, *Nova*, and *BTHAI*. These are bots that demonstrated similar win ratios to EISBot, and this subset of bots does not include the best performing or worst performing bots from the 2011 competition. I also evaluated EISBot versus all three races of the built-in AI.

The first study evaluated the performance of EISBot with four different particle model configurations. These mirror the models used in off-line analysis in Section 5.3.4 with one modification: the perfect tracker was replaced by a perfect information model, which is granted complete game state information. With the exception of the particle model policy, all other EISBot settings and behaviors were held fixed. I evaluated the following models:

- **Null Model:** A particle model that never spawns particles.
- **Default Model:** A model in which particles do not move and do not decay, providing a last known position.
- **Optimized Model:** A model with weights selected from the optimization process.
- **Perfect Information:** A particle model with complete game state information.

Each model was evaluated against all opponent and map permutations in three game match ups. The map pool consisted of a subset of the maps used in the 2011 AI competition: Python and Tau Cross. Win ratios for the different models are shown in

Table 7.1: The first study evaluated the performance of different particle model configurations. The win ratios against the built-in AI and competition bots show that the optimized model performed best overall.

	Versus Protoss	**Versus Terran**	**Versus Zerg**	**Overall**
Null Model	0.50	0.75	0.75	0.67
Default Model	0.58	0.75	0.67	0.67
Optimized Model	**0.75**	0.75	**0.83**	**0.78**
Perfect Information	0.50	**0.83**	0.67	0.67

Table 7.1. Overall, the optimized particle model had the highest win ratio by over 10%. Given these results, the optimized particle model was selected as the best performing model and used in all further experiments.

A surprising result was that the perfect information model did not perform best, since it has the most accurate information about the positions of enemy forces. The most likely cause of this result was the lack of scouting behavior performed when utilizing this model. Since the agent has perfect information, it does not need to scout in order to populate working memory with particles. Scouting units improved the win rate of the agent by distracting the opponent, such as diverting rush attacks.

The second study evaluated the performance of EISBot with different GDA subtasks enabled. The ablation study evaluated four versions of the agent. In each experiment, I introduced an additional subtask of the GDA model into the agent. I evaluated the following configurations of the system:

- **Base Agent:** A base version of the agent with a fixed strategy goal and no GDA subtasks. All strategy selection and monitoring is implemented as ABL behaviors.

ABL behaviors that operate on expectations are disabled.

- **Formulator:** A version of the agent that integrates the goal formulation subtask, with no look ahead or discrepancy detection. The input to the goal formulator is the current game state, rather than an anticipation of future game state caused by the opponent. Goal formulation is triggered only when the current plan completes. ABL behaviors that operate on expectations are disabled.

- **Predictor:** A version of the agent that incorporates explanation generation and goal formulation, but no discrepancy detection. Goal formulation is triggered only when the current plan completes.

- **GDA Agent:** A version of the agent implementing the complete GDA model. Goal formulation is triggered by plan completion and discrepancy detection.

Each experiment introduces a new GDA subtask into the agent. I used a greedy hill-climbing approach for selecting parameter values for the GDA subtasks in the system. The map pool consisted of six maps from the competition, which support two to four players and encourage a variety of play styles. For each trial in the experiments, I evaluated EISBot against all permutations of bots and maps, resulting in 48 games.

In each GDA experiment, I selected a different initial build order for the agent. I selected different initial build orders in order to demonstrate improvements in the gameplay performance of the system using the GDA model. There are game situations in which using the GDA model does not improve the gameplay performance of the system, because the outcome of the game is determined before any of the GDA subtasks

are invoked. This situation can arise in StarCraft, because the game enforces partial observability through the *fog-of-war* and the agent's initial goal may be dominated by the opponent's initial goal. In StarCraft, losing a game based solely on initial goal selection is referred to as a *build-order loss*. One of the ways expert players deal with this aspect of gameplay is by building expectations of opponent build orders, and by executing build orders that are unlikely to result in a build order loss. To prevent the agent's win rate from plateauing due to this aspect of RTS gameplay, I evaluated a variety of initial goals in the experiments. Because different initial goals are assigned to the agent, the win ratios reported in each experiment should not be directly compared to win ratios in other experiments. A direct comparison of the different agent configurations utilizing the same initial build order is presented in the fourth experiment. Additionally, the experiments presented in the user study compare the agent using the same build order.

In the first experiment, I evaluated the formulator agent with various planning window sizes. I also evaluated the base agent, represented by the formulator agent with a planning window of size 0. Results from the experiment are shown in Table 7.2. The agent performed best with a planning window of size 15. With smaller window sizes the agent often retrieved duplicate production actions resulting in wasted resources. With larger window sizes, goal formulation is performed less frequently.

The second experiment evaluated the predictor agent with various look-ahead window sizes and a fixed planning window size of 15. I also evaluated the formulator agent, represented by the predictor agent with a look-ahead window of size 0. Results from the experiment are shown in Table 7.3. The agent performed best with a look-ahead

window size of 10. Incorporating predictions helped the agent prepare for opponent strategies. With smaller window sizes, the agent was unable to anticipate opponent actions in time to develop counter strategies, while large window sizes resulted in the agent employing counter strategies too early, resulting in wasted resources.

In the third experiment, I evaluated the complete GDA agent with various discrepancy period sizes, and planning window and look-ahead windows of size 15 and 10. I also evaluated the predictor agent, represented by the GDA agent with a discrepancy period of ∞. Results from the experiment are shown in Table 7.4. The agent performed best with a discrepancy period of 30 seconds. Discrepancy detection enabled the agent to respond to critical game situations, such as formulating goals to build detector units in order to reveal cloaked units. When using larger discrepancy periods, the agent rarely detected discrepancies that trigger new goals, while smaller discrepancy periods resulted in the same problems as small planning window sizes.

The fourth experiment evaluated the different configurations of the agent using the same initial goal, providing a direct comparison of the different agent configurations. Results from the experiment are shown in Table 7.5. The results show that each additional GDA subtask integrated into the agent improved the overall system performance; the agent has the highest win ratio when utilizing the complete GDA model.

EISBot had varying success against each of the opponent agents. Win rates versus individual opponents during the fourth experiment are shown in Table 7.6. Five of the opponents were dominated by EISBot, and all ablations of the system achieved 100% win rates versus these opponents. Against these opponents, the GDA subtasks

Table 7.2: Win ratios from the goal formulation experiment show that the agent performed best with a planning window size of 15 actions.

Planning Window Size (n)	**Win Ratio**
0	0.73
1	0.79
5	0.88
10	0.85
15	0.91
20	0.88

Table 7.3: Win ratios from the opponent prediction experiment show that the agent performed best using a look-ahead window of 10 actions.

Look-Ahead Window Size (m)	**Win Ratio**
0	0.71
5	0.75
10	0.79
15	0.73
20	0.69

Table 7.4: Win ratios from the discrepancy detection experiment show that the agent performed best with a discrepancy period of 30 seconds.

Discrepancy Period (δ)	**Win Ratio**
∞	0.81
60	0.83
30	0.92
15	0.81

Table 7.5: The fourth experiment evaluated all of the agent configurations using the same initial goal. The agent performed best when using the complete GDA model.

Agent	Win Ratio
Base	0.73
Formulator	0.77
Predictor	0.81
GDA	0.92

for anticipating opponent actions and triggering goal formulation were not necessary to win. Out of the five opponents that were dominated by EISBot, only one of the opponent bots lost as a direct result of build order selection. The BTHAI bot rushes for early cloaking units and was defeated in each game before these units could be utilized. In the other match ups, the opponent bots lost primarily due to a lack of economic and technology expansion, and weak tactics. Against these opponents, the ability to perform micromanagement behaviors while also pursuing technology and economic expansion goals enabled EISBot to consistently win games.

Against three of the opponents, incorporating additional GDA subtasks improved the performance of EISBot. The largest win ratio margin occurred versus the Blizzard Protoss AI. This opponent demonstrated the widest variety of strategies, and using the GDA model enabled the agent to effectively respond to most of the encountered situations. Nova executes a strategy that required the agent to build detection units, and incorporating explanation generation enabled EISBot to anticipate this situation. The largest improvement was demonstrated versus Aiur with discrepancy detection enabled. However, the results versus the Blizzard Protoss AI show that each of the GDA

Table 7.6: EISBot had varying success against individual opponents in the fourth experiment. The base version performed worst, while the complete GDA version improved in performance versus three of the opponent agents.

	Base	**Form.**	**Pred.**	**GDA**
Blizzard Protoss	0.14	0.33	0.50	0.83
Blizzard Terran	1.00	1.00	1.00	1.00
Blizzard Zerg	1.00	1.00	1.00	1.00
Aiur	0.00	0.00	0.00	0.50
SPAR	1.00	1.00	1.00	1.00
Cromulent	1.00	1.00	1.00	1.00
Nova	0.67	0.83	1.00	1.00
BTHAI	1.00	1.00	1.00	1.00
Overall	0.73	0.77	0.81	0.92

subtasks improved the performance of EISBot.

The outcomes of the ablation studies show that EISBot performed best when using the optimized particle model and all of the GDA subtasks. Adding each GDA subtask to the agent individually also provided a method for selecting parameters for each of the GDA components. The parameters that performed best are used in all further experiments. One of the issues in this evaluation is that different build orders were used for evaluating each of the GDA subtasks. An additional problem is that using different sets of opponents for evaluation result in different win ratios for EISBot. While the fourth experiment compared each of the GDA configurations using the same build order, the reported win ratio is relative to the opponent bots used for evaluation. A key limitation of evaluating against AI opponents is that there is a small set of bots available for analysis and the performance of these bots versus human players is unknown. To determine the performance of the agent with respect to expert players, it is necessary to also test the system against human participants.

7.2 User Study

My central claim is that reproducing expert-level performance for RTS gameplay requires integrating heterogeneous reasoning capabilities. Currently, humans are the only source of expert gameplay in this domain. In order to support the claim that a StarCraft agent performs at an expert level, it is necessary to evaluate the performance of the system with human players. The objective of evaluating EISBot versus human opponents is also motivated by expressive AI [58], which strives to understand the interaction between AI systems and players.

In order to test my claim that integrating heterogeneous reasoning capabilities is necessary for expert RTS gameplay, I conducted an ablated user study. An *ablated user study* is an analysis of a system that integrates human participants in the evaluation. To perform the study, I evaluated different ablations of EISBot versus human participants. The outcome of this study enables ablations of the system to be compared with respect to expert human players. My hypothesis is that versions of EISBot that integrate additional reasoning components will perform better than the ablated versions versus human players.

To perform the user study, I hosted matches on the International Cyber Cup (ICCup)[2]. ICCup is a third-party ladder server that assigns players a score based on wins and loses. The ICCup score is similar to the Elo score used to rank players in chess. It is an intrinsic rating and enables players to discuss their level of skill independent of specific opponents. Additionally, the ICCup score can be used to directly compare the

[2] `http://ICCup.com`

gameplay performance of humans and agents. I use ICCup score as the primary metric for evaluating EISBot versus human players, because it reduces the impact of specific opponents on the overall evaluation of the system. On the ICCup server, players are assigned a letter grade based on the ICCup score. At the lower ranges, the following ranks are assigned to players:

- **E:** 0-500 points, a novice player.
- **D-:** 500-100 points, an amateur player.
- **D:** 1000-2000 points, a competitive amateur player.
- **D+:** 2000-3000 points, a competitive amateur player.

The upper bound of the rating scale is 15,000, which is reached only by top professional players. Players begin with a provisional rating of 1,000 points and gain 100 points for a win against an evenly ranked opponent and lose 50 points for a loss against an evenly ranked opponent. The server implements a season system, which resets player scores every three months.

I developed a script for automating the process of running games on ICCup. The script launches StarCraft, logs in to ICCup, hosts a match, waits for an opponent, and then starts the game. The system has no control over the race selected by the opponent and does not restrict the rank of opponent players. At the start of the match, the system announces that it is a bot and the opponent player has the option of leaving the match if they do not want to participate in the experiment. I also authored behaviors

Table 7.7: The first trial evaluated EISBot on Longinus. This table shows the minimum, maximum, final, and average ICCup scores achieved by each of the ablations. On Longinus, the Predictor version performed best, with an average score of 1111.

	Minimum	**Maximum**	**Final**	**Average**
Base	629	1196	1023	942
Formulator	761	1134	761	980
Predictor	729	1369	1169	1111
GDA	631	1207	1021	952

in EISBot to support well mannered gameplay. The agent ends the game if there are less than three worker units or the primary command center is destroyed. If a defeat situation is detected by EISBot, the agent acknowledges defeat and ends the game.

The user study consisted of three trials. Each trial evaluated EISBot on a different map, and all other variables were held fixed. The map pool is a subset of the maps used in the 2011 StarCraft AI competition. During each trial, the four different GDA ablations were evaluated over 50 games. The same build order was used across all maps, ablations, and opponent races. The build order is a two gateway opening that produces six zealots while upgrading to the second technology tier. If the agent crashes during a game, the game is counted as a loss by the server.

The first trial evaluated EISBot on the map Longinus. Longinus is a three-player map with notable features including a high middle ground, double gas expansions, and easily-wallable natural expansions. Results from the first trial are shown in Table 7.7 and Figure 7.1. In this trial, the Predictor version performed best with an average score of 1111, while the Base version performed worst with an average score of 942. Overall, there was little variation among the performance of the different versions of EISBot.

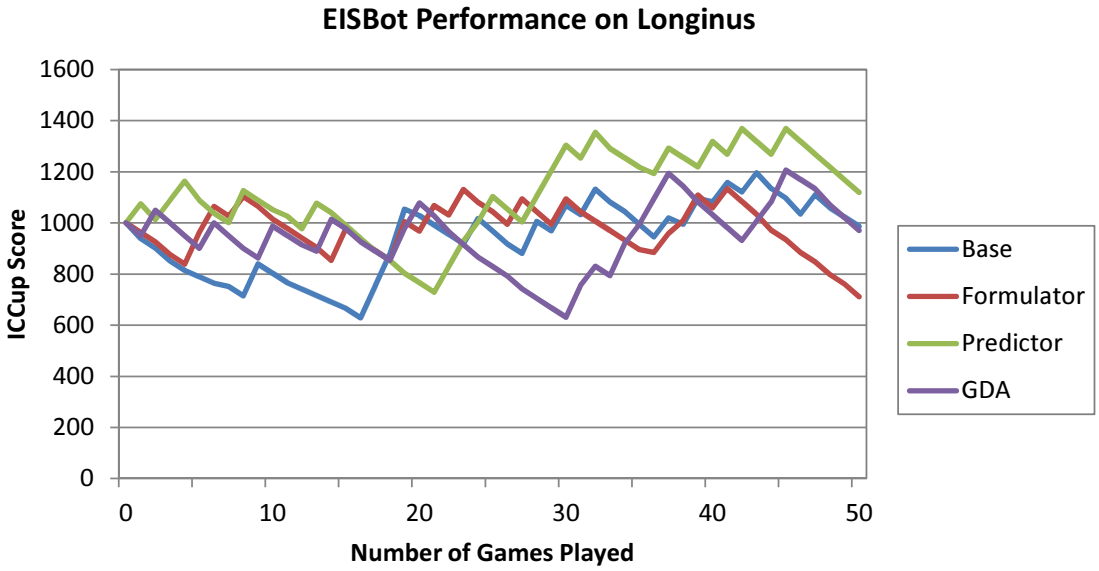

Figure 7.1: In the first trial, each of the GDA ablations was evaluated on the map Longinus over 50 games. The results show that the Predictor version performed best and the Formulator version performed worse. Overall, there was little variation among the performances of the different EISBot versions.

Table 7.8: The results from the second trial show that EISBot overall performed worse on the map Python. The final rating for each of the versions was **E**, except for the complete EISBot version, which achieved a rating of **D-**.

	Minimum	Maximum	Final	Average
Base	268	1075	268	599
Formulator	330	1015	330	718
Predictor	292	1000	509	555
GDA	595	1109	872	860

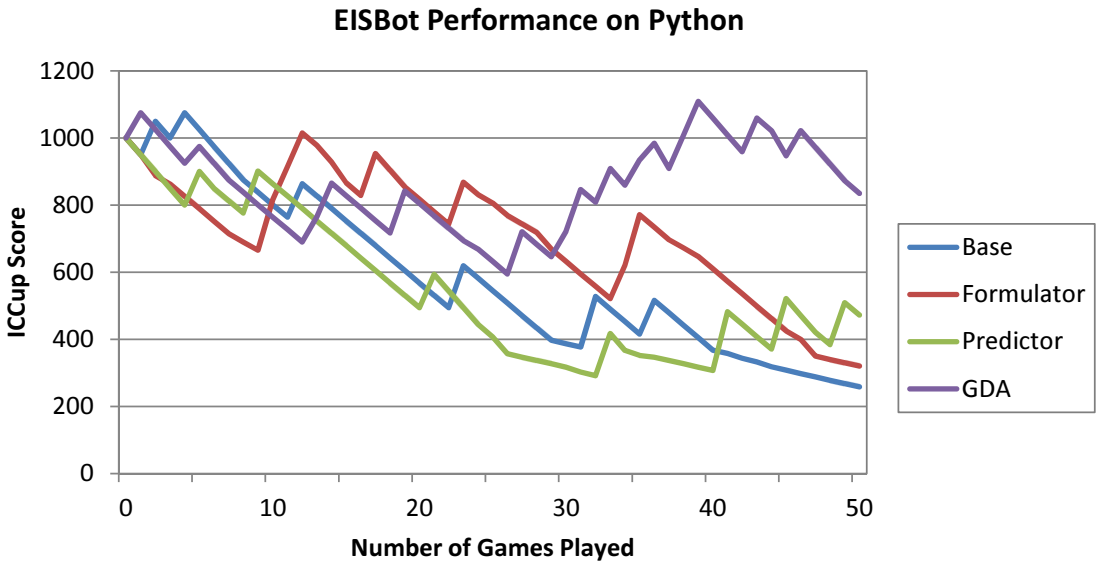

Figure 7.2: The second trial evaluated each of the GDA ablations on the map Python over 50 games. The results shows that each version of EISBot that incorporated additional GDA subtasks achieved a higher final score. In this trial, there was a noticeable difference in performance between the complete EISBot version and ablated versions.

The second trial evaluated EISBot on Python. Python is a four-player map with easy to defend ramps, wide expansion entrances, and two island expansions. Results from the second trial are shown in Table 7.8 and Figure 7.2. During this trial, the complete version outperformed the ablated versions. The final rating for each of the versions was **E**, except for the complete EISBot version, which achieved a rating of **D-**. Overall, EISBot performed worse during this trial than the previous trial. One of the reasons for this decrease in performance is that the main bases on Python have easy to defend ramps, and EISBot lacks sufficient ABL behaviors for effectively attacking up ramps. Another issue encountered during this trial was that it was performed at the start of an ICCup season. Each player's score is reset at the start of an ICCup season, which means that EISBot was often playing against opponents with a much higher skill

Table 7.9: ICCup scores from the third trial show that EISBot performed best with all GDA subtasks. All versions greatly outperformed the base agent.

	Minimum	**Maximum**	**Final**	**Average**
Base	312	1150	462	669
Formulator	841	1364	1266	1078
Predictor	878	1338	1191	1145
GDA	938	1801	1502	1293

than what their score reflected.

The third trial evaluated EISBot on the map Tau Cross. Tau Cross is a three-player map with notable features including an open and buildable center, slightly longer rush distances, and a small natural choke point. Results from the third trial are shown in Table 7.9 and Figure 7.3. EISBot overall performed better in this trial, and all ablations achieved a **D** rating, expect for the base version. There was a large difference between the base version and other ablations. The Predictor and Formulator versions maintained a score close to the starting score, the Base version consistently decreased in score, and the complete version consistently increased in score.

The overall results for the user study are shown in Table 7.10. I computed an overall score for each ablation as the mean of the average score achieved during each trial. The goal of averaging over several trials is to compensate for the lack of control over opponent races and skill levels, as well as factors such as the season system that resets player scores. Overall, the ablations achieved a ranking of **D-**, which corresponds to an amateur StarCraft player. The results shown here are for a specific build order, and different build orders would result in different scores. Overall, the GDA version performed best with an average ICCup score of 906, while the base version performed

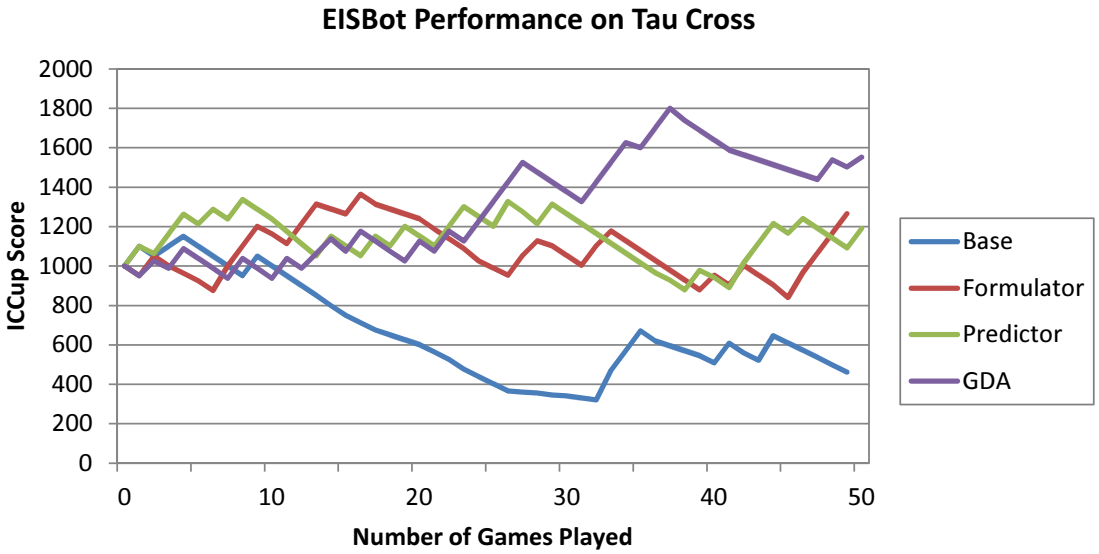

Figure 7.3: The third trial evaluated each of the GDA ablations on the map Tau Cross over 50 games. In this trial, there was a noticeable difference between the base version and the other ablations. The complete GDA agent performed best, while the Formulator and Predictor versions achieved similar ratings.

worst with a score of 771.

During the user study, 17,670 players were active on ICCup. The percentile ranks achieved by EISBot using the complete GDA model are shown in Table 7.11. The percentile rank is computed using the final score reached after 50 games. In the best performing trial, EISBot outperformed 66% of the players on the server. Based on the

Table 7.10: The overall results of the ablated user study show that the versions of EISBot that incorporated additional GDA subtasks performed better versus human opponents.

Agent	Longinus	Python	Tau Cross	Overall
Base	942	599	669	737
Formulator	980	718	1078	925
Predictor	1111	555	1145	937
GDA	952	860	1293	1035

Table 7.11: Percentile ranks achieved by the complete agent on ICCup after 50 games.

Trial	**Final Score**	**Percentile Rank**
Longinus	1021	32^{nd}
Python	872	8^{th}
Tau Cross	1502	66^{th}
Average	1131	48^{th}

average final score over three trials, EISBot outperformed 48% of competitive human players on ICCup.

To test my hypothesis that expert StarCraft gameplay requires integrating heterogeneous reasoning capabilities, I performed an ablated user study involving human participants on the ICCup ladder server. The results from these experiments show that adding GDA subtasks to the agent did improve the performance of the system. While these results support my claim, the system currently performs at a much lower ranking than expert players. One of the challenges in evaluating complete game-playing agents is that there are several ways that the system can fail, and performing well in this domain requires achieving mastery in a variety of different skills. While EISBot currently performs at only the level of an amateur player, the results presented here support the claim that adding anticipation, adaptation, and estimation capabilities to the agent improve the performance of the system in a complete game-playing task, against human opponents.

The results presented in this thesis provide a direct comparison of EISBot with competitive human players. To the best of my knowledge, no previous RTS agents in the literature have evaluated performance with respect to human players. Therefore,

the results presented here provide a high-water mark for machine play.

7.3 Summary

To test my claim that StarCraft requires integrating heterogeneous reasoning capabilities, I performed two ablation studies and a user study. In the first ablation study, I evaluated different particle model configurations. The second ablation study evaluated EISBot versions with different GDA subtasks enabled. Both of these studies evaluated performance against the build-in AI of StarCraft and bots from the StarCraft AI Competition. The results showed that the agent performed best using the optimized particle model with all of the GDA subtasks enabled. Another purpose of the ablation studies was that it provided a method for selecting GDA parameters.

The user study evaluated different EISBot configurations versus human opponents on the ICCup StarCraft ladder server. The server assigns players a score based on their win and loss record, which provides an intrinsic measurement of gameplay performance. I performed three trials, where each trial evaluated the GDA ablations on a different map. The results showed that including additional GDA subtasks in the agent improved its performance versus human opponents. Currently, EISBot performs at the level of an amateur player.

Chapter 8

Conclusions

In this thesis, I explored methods for building human-level AI for StarCraft. StarCraft gameplay is a complex process and involves managing several distinct tasks. I claim that reproducing expert-level StarCraft gameplay requires integrating heterogeneous reasoning capabilities. In order to support these capabilities in a real-time agent, I integrated a reactive planning agent with gameplay models learned from demonstrations. The resulting system, EISBot, plays complete games of StarCraft and emulates the adaptation, anticipation, and estimation capabilities demonstrated by human players.

I investigated three research questions, with the objective of identifying and realizing the capabilities necessary for expert RTS gameplay in an agent:

1. What competencies are required for expert RTS gameplay?

2. Which competencies can be learned from demonstrations?

3. How can distinct competencies be integrated in a real-time agent?

The contributions of this work are idioms for authoring multi-scale agents, techniques for learning from gameplay demonstrations, and methods for integrating learning algorithms in a reactive planning agent. An additional outcome of this work is the first evaluation with competitive human participants, providing a high-water mark for machine play.

My approach for answering the first question was to analyze professional-level StarCraft gameplay and commentaries, study player created repositories of strategies and tactics, and build upon previous work. Expert RTS gameplay is complex, and involves a combination of deliberative and reactive decision making. Additionally, StarCraft requires the ability to simultaneously manage several interconnected tasks while pursuing higher-level goals. To support these capabilities in an agent, I extended an integrated agent framework that decomposes RTS gameplay into domains of competence[61]. Each manager is responsible for handling a specific aspect of gameplay and coordinating with interrelated managers in the system.

I classify StarCraft gameplay as a multi-scale AI problem. It is a multi-task domain in which actions are performed across different scales, tasks are interrelated and performance in each task impacts other tasks, and actions and decision making occur in real-time. I advocate reactive planning as a method for authoring agents that operate in multi-scale domains, because it provides many of the mechanisms necessary for multi-scale reasoning. To support the authoring of multi-scale agents, I propose reactive planning idioms that build upon the ABL planning language. These idioms

provide constructs for spawning daemon behaviors, organizing agents as a collection of managers, producing and consuming messages, and locking behaviors. I have applied these idioms to author a StarCraft agent that integrates high-level strategic reasoning with reactive tactical reasoning.

My approach for answering the second question was to develop methods for learning from StarCraft replays. The goal of these methods is to enable estimation, anticipation, and adaptation capabilities in an agent. I presented three applications of gameplay demonstrations: model training for classification and regression algorithms that identify the strategy an opponent is performing and estimate when specific technologies will be unlocked by a player, case-based goal formulation for selecting strategies for the agent to pursue and anticipating the goals of opponents, and model optimization for a particle model that tracks opponent forces. To evaluate the different models, I performed offline experiments using replays collected from professional players. One of the interesting outcomes of this analysis was that the strategy prediction models exhibited foresight.

To answer the third research question, I investigated methods for integrating learned gameplay models with reactive planning. I presented four ways to interface external components with the ABL agents: augmenting working memory, external plan generation, external goal formulation, and behavior activation. Each of these integration approaches uses ABL's working memory to facilitate communication between different components in the agent. An additional way in which components are integrated in EISBot is through the goal-driven autonomy conceptual model.

The resulting system, EISBot, incorporates a library of hand-authored ABL behaviors, a particle model for tracking enemy units, and an implementation of the GDA conceptual model for strategy selection and monitoring. In order to support coordination between different learning components in the agent, I explored methods for learning GDA subtasks from gameplay demonstrations. Applying the GDA model enables the agent to build expectations of opponent actions, detect when discrepancies occur, generate explanations of future opponent actions, and formulate new goals to pursue. Each of the integrated components enables the agent to realize additional capabilities: the reactive planner enables the system to select and monitor plans, the particle model enables the agent to build estimations of opponent locations, and the GDA subtasks enable the agent to generate anticipations of opponent actions and adapt to new gameplay situations.

To test my claim that StarCraft requires integrating heterogeneous reasoning capabilities, I performed two ablation studies and a user study. The results of the ablation studies showed that the agent performed best using the optimized particle model with all of the GDA subtasks enabled. The user study evaluated different EISBot configurations versus human opponents on the ICCup StarCraft ladder server. The outcome of the user study demonstrated that incorporating additional capabilities into the system improved the agent's performance against human players. The results also demonstrated that EISBot currently plays at the level of a amateur player.

8.1 Discussion

A result of this work is a game-playing agent that emulates a subset of human capabilities and performs at the level of a competitive amateur player. However, there are several capabilities that the system does not capture. Additionally, existing bots often outperform EISBot, and the system has advantages over human opponents.

Human players demonstrate a broad range of reasoning capabilities during RTS gameplay. While I have explored methods for emulating estimation, anticipation, and adaptation capabilities, players also exhibit additional capabilities. In tournament matches, in which players play multiple matches versus the same opponent, players select build orders based on the opponent's anticipated strategy and build orders executed during previous games. It is common for players to alternate build orders in tournament games to prevent the opponent from formulating a dominant counter strategy. Tournament gameplay involves a form of meta-game adaptation that is not currently captured by EISBot, because EISBot maintains no history of previous matches. One way of emulating this capability in EISBot is to record previous matches and to inform the strategy selection process based on the performance of previously executed strategies. Additionally, a boredom heuristic [100] could be used to prevent the system from repeating recently selected strategies.

I evaluated the performance of EISBot versus human and computer opponents. While the agent outperformed most of the bots in the ablation studies, the study did not include the top performing bots from the StarCraft AI Competition. In the 2011 Star-

Craft AI Competition, the submitted version of EISBot, which most closely resembles the base agent configuration, placed 5$^{\text{th}}$ out of 13 participants. The tournament used a round-robin format in which each bot played 30 games against each of the opponent bots. The winning submission, *Skynet*, demonstrated excellent economy management, squad management, and attack timing. In the 2011 competition, Skynet defeated EISBot in 28 out of 30 games. While the results show that Skynet outperformed EISBot and all other participants in the competition, it is unknown how well the agent performs against human players.

The results from the competition show that existing AI methodologies can be leveraged to author StarCraft agents with strong gameplay capabilities. In order to achieve this result, the winning bot required a huge amount of knowledge engineering. One of the limitations of this approach is that most of the agent's behavior is hard coded, including building locations. While generalization is not a direct objective of the competition, a subset of the participants have explored methods for generalizing RTS capabilities. Luke Perkins introduced a map analysis method that identifies regions and chokepoints, enabling agents to generalize to new maps [82]. The *Overmind* team introduced a threat-aware A* heuristic enabling the agent to generalize to new tactical situations [12]. The *UAlbertaBot* team explored the application of deliberative planning to strategy selection, potentially enabling the agent to generalize to all three StarCraft races [22].

One of the objectives of this work was to develop an agent that interfaces with the game at the same level as a human player. Currently, the system has two

advantages over human players: it can sense all visible game state without panning the camera, and it can simultaneously issue orders to all units across the map. One way of limiting the first advantage would be to provide the agent with screen capture data rather than direct game state access [55]. Dealing with this constraint would require a mechanism for focusing attention on particular areas of the map. One way of limiting the second advantage would be to restrict the number of actions, or APM, allowed by the agent. Enforcing this constraint would prevent agents from exploiting unit behaviors and minimize the advantage provided to bots.

8.2 Future Work

While I managed to build a system capable of amateur-level StarCraft gameplay, there is still a large margin between the capabilities of EISBot and expert players. One way of closing this gap is to author additional reactive planning behaviors, such as transport behaviors in EISBot. Another approach is to learn additional competencies from demonstrations and explore methods for adaptation across multiple scales. I identify the following directions for future work:

- *Online Learning:* One of the limitations of the methods presented in this thesis is that a batch process is used for learning from demonstrations. In order to track the evolving meta-game of StarCraft it is necessary to constantly learn new strategies and tactics. The methods for building gameplay models presented in this thesis require an expert to collect and maintain a case library. To overcome

this limitation, future work could explore methods for learning during gameplay or learning after each game. To effectively incorporate new gameplay demonstrations, the system would need to be able to idenitfy novel gameplay.

- *Multi-Task GDA:* I investigated the application of the GDA model to a single gameplay task in EISBot. Future work could explore the application of goal-driven autonomy to multiple gameplay tasks including strategy selection, economy management, unit production, and scouting.

- *Stylized Gameplay:* Learning from demonstration can be applied to individuals or groups of players. The models presented in this thesis are trained using demonstrations from a large group of players, but the models could also be trained using demonstrations from individual players. Future work could explore methods for capturing gameplay behavior of specific players, in order to emulate stylized gameplay of professional players.

One of the limitations of my approach is that all examples in the case library are weighted equally. One way to potentially improve case retrieval is to incorporate a performance measure into the retrieval process. Cases that lead to improved game states are marked as more relevant than cases that do not improve game state. One of the challenges in adding a performance measure to cases is formulating an objective function that evaluates game state based on partial information. Aha et al. explore a metric for RTS games based on difference in score [3], which provides a measure of gameplay performance. Another approach is to incorporate human experts into the

evaluation process, in which the experts score retrieved cases based on relevance.

Another direction for future work is to use GDA for identifying opportunistic goals, rather than just responding to planning failures. The current system triggers goal formulation when an expectation is violated, which enables the agent to respond to the situation with a new goal. Opportunistic goals are promoted for execution when particular situations arise, such as identifying a weakness in an opponent's strategy. The use of opportunistic goals fits well with the application of GDA to multiple subtasks. For example, during goal formulation for strategy selection, the agent could promote economy management goals such as expanding.

Future work could also explore the use of learning from demonstration to enable new types of gameplay experiences, such as emulating the gameplay style of a professional player. Rather than learning from a collection of players, the agent could build gameplay models using demonstrations from a particular player. In order for the system to resemble the player's behavior, it would be necessary to learn gameplay patterns across multiple subtasks. One of the challenges in implementing this capability is learning from a much smaller demonstration library. Additionally, the agent should be tuned to the player's skill level in order to provide an engaging gameplay experience.

Bibliography

[1] A. Aamodt and E. Plaza. Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AI Communications*, 7(1):39–59, 1994.

[2] D. W. Aha, D. Kibler, and M. K. Albert. Instance-Based Learning Algorithms. *Machine Learning*, 6(1):37–66, 1991.

[3] D. W. Aha, M. Molineaux, and M. Ponsen. Learning to Win: Case-Based Plan Selection in a Real-Time Strategy Game. In *Proceedings of ICCBR*, pages 5–20. Springer, 2005.

[4] H. Akaike. A New Look at the Statistical Model Identification. *IEEE Transactions on Automatic Control*, 19(6):716–723, 1974.

[5] D. W. Albrecht, I. Zukerman, and A. E. Nicholson. Bayesian Models for Keyhole Plan Recognition in an Adventure Game. *User Modeling and User-Adapted Interaction*, 8(1):5–47, 1998.

[6] S. Bakkes, P. Kerbusch, P. Spronck, and J. van den Herik. Predicting Success in an

Imperfect-Information Game. In *Proceedings of the Computer Games Workshop*, pages 219–230, 2007.

[7] S. Bakkes, P. Spronck, and J. van den Herik. Rapid Adaptation of Video Game AI. In *Proceedings of IEEE CIG*, pages 79–86, 2008.

[8] R. K. Balla and A. Fern. UCT for Tactical Assault Planning in Real-Time Strategy Games. In *Proceedings of IJCAI*, pages 40–45, 2009.

[9] R. Baumgarten, S. Colton, and M. Morris. Combining AI Methods for Learning Bots in a Real-Time Strategy Game. *International Journal on Computer Game Technologies*, 2009.

[10] C. Bererton. State Estimation for Game AI Using Particle Filters. In *Proceedings of the AAAI Workshop on Challenges in Game AI*, 2004.

[11] N. Blaylock and J. Allen. Corpus-based, Statistical Goal Recognition. In *Proceedings of IJCAI*, pages 1303–1308, 2003.

[12] D. Burkett, D. Hall, and D. Klein. Optimal Graph Search with Iterated Graph Cuts. In *Proceedings of AAAI*, pages 12–17, 2011.

[13] M. Buro. Real-Time Strategy Games: A New AI Research Challenge. In *Proceedings of IJCAI*, pages 1534–1535, 2003.

[14] S. Butler and Y. Demiris. Partial Observability During Predictions of the Opponent's Movements in an RTS Game. In *Proceedings of IEEE CIG*, pages 46–53, 2010.

[15] S. Carberry. Techniques for Plan Recognition. *User Modeling and User-Adapted Interaction*, 11(1-2):31–48, 2001.

[16] A. Champandard. Getting Started with Decision Making and Control Systems. In S. Rabin, editor, *AI Game Programming Wisdom 4*, pages 257–264. Charles River Media, 2008.

[17] E. Charniak and R. P. Goldman. A Bayesian Model of Plan Recognition. *Artificial Intelligence*, 64(1):53–79, 1993.

[18] F. Chee. Understanding Korean experiences of online game hype, identity, and the menace of the "Wang-tta". In *Proceedings of DiGRA*, 2005.

[19] D. C. Cheng and R. Thawonmas. Case-Based Plan Recognition for Real-Time Strategy Games. In *Proceedings of the Game-On Conference*, pages 36–40, 2004.

[20] D. Choi. Reactive Goal Management in a Cognitive Architecture. *Cognitive Systems Research*, 12(3–4):293–308, 2011.

[21] D. Choi, T. Konik, N. Nejati, C. Park, and P. Langley. A Believable Agent for First-Person Shooter Games. In *Proceedings of AIIDE*, pages 71–73, 2007.

[22] D. Churchill and M. Buro. Build Order Optimization in StarCraft. In *Proceedings of AIIDE*, pages 14–19, 2011.

[23] M. T. Cox. Perpetual Self-Aware Cognitive Agents. *AI Magazine*, 28(1):32–45, 2007.

[24] M. T. Cox, H. Muñoz-Avila, and R. Bergmann. Case-based planning. *The Knowledge Engineering Review*, 20(3):283–287, 2006.

[25] C. Darken and B. Anderegg. *Game AI Programming Wisdom 4*, chapter Particle Filters and Simulacra for More Realistic Opponent Tracking. Charles River, S. Rabin editor, 2008.

[26] E. Dereszynski, J. Hostetler, A. Fern, T. Dietterich, T. T. Hoang, and M. Udarbe. Learning Probabilistic Behavior Models in Real-time Strategy Games. In *Proceedings of AIIDE*, pages 20–25, 2011.

[27] S. Epstein and J. Shih. Sequential Instance-Based Learning. In R. Mercer and E. Neufeld, editors, *Lecture Notes in Computer Science*, volume 1418, pages 442–454. Springer, 1998.

[28] M. Fagan and P. Cunningham. Case-Based Plan Recognition in Computer Games. In *Proceedings of ICCBR*, pages 161–170, 2003.

[29] H. Fernlund, A. J. Gonzalez, M. Georgiopoulos, and R. F. DeMara. Learning Tactical Human Behavior through Observation of Human Performance. *IEEE Transactions on Systems, Man, and Cybernetics*, 36(1):128–140, 2006.

[30] R. E. Fikes and N. J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2:189–208, 1971.

[31] G. Flórez-Puga, M. Gomez-Martin, B. Diaz-Agudo, and P. Gonzalez-Calero. Dynamic Expansion of Behaviour Trees. In *Proceedings of AIIDE*, pages 36–41, 2008.

[32] K. Forbus, J. V. Mahoney, and K. Dill. How Qualitative Spatial Reasoning Can Improve Strategy Game AIs. *IEEE Intelligent Systems*, 17(4):25–30, 2002.

[33] Y. Freund and R. Schapire. Experiments with a New Boosting Algorithm. In *Proceedings of the International Conference on Machine Learning*, pages 148–156, 1996.

[34] J. Friedman, T. Hastie, and R. Tibshirani. Additive Logistic Regression: A Statistical View of Boosting. *Annals of Statistics*, 28(2):337–374, 2000.

[35] J. H. Friedman. Stochastic Gradient Boosting. *Computational Statistics and Data Analysis*, 38(4):367–378, 2002.

[36] B. Hayes-Roth. A Blackboard Architecture for Control. *Artificial Intelligence*, 26(3):251–321, 1985.

[37] S. Hladky and V. Bulitko. An Evaluation of Models for Predicting Opponent Positions in First-Person Shooter Video Games. In *Proceedings of IEEE CIG*, pages 39–46, 2008.

[38] H. Hoang, S. Lee-Urban, and H. Muñoz-Avila. Hierarchical Plan Representations for Encoding Strategic Game AI. In *Proceedings of AIIDE*, 2005.

[39] C. Hogg, H. Muñoz-Avila, and U. Kuter. HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required. In *Proceedings of AAAI*, 2008.

[40] J. L. Hsieh and C. T. Sun. Building a Player Strategy Model by Analyzing Replays of Real-Time Strategy Games. In *Proceedings of the International Joint Conference on Neural Networks*, pages 3106–3111, 2008.

[41] D. Isla. Handling Complexity in the Halo 2 AI. In *Proceedings of the Game Developers Conference*, 2005.

[42] D. Isla. *Game AI Programming Wisdom 3*, chapter Probabilistic Target Tracking and Search Using Occupancy Maps, pages 379–388. Charles River Media, 2006.

[43] D. Isla. Halo 3-Building a Better Battle. In *Game Developers Conference*, 2008.

[44] D. Isla, R. Burke, M. Downie, and B. Blumberg. A Layered Brain Architecture for Synthetic Creatures. In *Proceedings of ICJAI*, pages 1051–1058, 2001.

[45] U. Jaidee, H. Muñoz-Avila, and D. W. Aha. Case-Based Learning in Goal-Driven Autonomy Agents for Real-Time Strategy Combat Tasks. In *Proceedings of the ICCBR Workshop on Computer Games*, pages 43–52, 2011.

[46] U. Jaidee, H. Muñoz-Avila, and D. W. Aha. Integrated Learning for Goal-Driven Autonomy. In *Proceedings of IJCAI*, pages 2450–2455, 2011.

[47] D. Josyula. *A Unified Theory of Acting and Agency for a Universal Interfacing Agent*. PhD thesis, University of Maryland, 2005.

[48] J. E. Laird. Research in Human-Level AI using Computer Games. *Communications of the ACM*, 45(1):32–35, 2002.

[49] J. E. Laird and M. van Lent. Human-Level AI's Killer Application: Interactive Computer Games. *AI magazine*, 22(2):15–25, 2001.

[50] P. Langley. Artificial Intelligence and Cognitive Systems. *AISB Quarterly*, 2011.

[51] P. Langley and D. Choi. A Unified Cognitive Architecture for Physical Agents. In *Proceedings of AAAI*, pages 1469–1474, 2006.

[52] P. Langley, J. E. Laird, and S. Rogers. Cognitive architectures: Research issues and challenges. *Cognitive Systems Research*, 10(2):141–160, 2009.

[53] J. F. Lehman, J. E. Laird, and P. Rosenbloom. A Gentle Introduction To Soar, An Architecture for Human Cognition. *An Invitation to Cognitive Science: Methods, Models, and Conceptual Issues*, 4:211–253, 1998.

[54] A. B. Loyall. *Believable Agents: Building Interactive Personalities*. PhD thesis, Carnegie Mellon University, 1997.

[55] S. Lucas. Ms Pac-Man Competition. *SIGEVOlution*, 2:37–38, 2007.

[56] B. Marthi, S. Russell, and D. Latham. Writing Stratagus-Playing Agents in Concurrent ALisp. In *Proceedings of the IJCAI Workshop on Reasoning, Representation and Learning in Computer Games*, 2005.

[57] B. Martin. Instance-Based Learning: Nearest Neighbour with Generalisation. Master's thesis, University of Waikato, Hamilton, New Zealand, 1995.

[58] M. Mateas. *Interactive Drama, Art and Artificial Intelligence.* PhD thesis, Carnegie Mellon University, 2002.

[59] M. Mateas and A. Stern. A Behavior Language for Story-Based Believable Agents. *IEEE Intelligent Systems*, 17(4):39–47, 2002.

[60] M. Mateas and A. Stern. Façade: An Experiment in Building a Fully-Realized Interactive Drama. In *Game Developers Conference*, 2003.

[61] J. McCoy and M. Mateas. An Integrated Agent for Playing Real-Time Strategy Games. In *Proceedings of AAAI*, pages 1313–1318, 2008.

[62] R. Metoyer, S. Stumpf, C. Neumann, J. Dodge, J. Cao, and A. Schnabel. Explaining How to Play Real-Time Strategy Games. *Research and Development in Intelligent Systems*, pages 249–262, 2010.

[63] M. Molineaux, D. W. Aha, and P. Moore. Learning Continuous Action Models in a Real-Time Strategy Environment. In *Proceedings of FLAIRS*, pages 257–262, 2008.

[64] M. Molineaux, D. W. Aha, and M. Ponsen. Defeating Novel Opponents in a Real-Time Strategy Game. In *Proceedings of the IJCAI Workshop on Reasoning, Representation and Learning in Computer Games*, 2005.

[65] M. Molineaux, M. Klenk, and D. W. Aha. Goal-Driven Autonomy in a Navy Strategy Simulation. In *Proceedings of AAAI*, pages 1548–1553, 2010.

[66] H. Muñoz-Avila and D. W. Aha. A Case Study of Goal-Driven Autonomy in Domination Games. In *Proceedings of the AAAI Workshop on Goal-Directed Autonomy*, 2010.

[67] H. Muñoz-Avila, D. W. Aha, U. Jaidee, and E. Carter. Goal-Driven Autonomy with Case-Based Reasoning. In *Proceedings of ICCBR*, pages 228–241, 2010.

[68] H. Muñoz-Avila, D. W. Aha, U. Jaidee, M. Klenk, and M. Molineaux. Applying Goal Driven Autonomy to a Team Shooter Game. In *Proceedings of FLAIRS*, pages 465–470, 2010.

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

[70] J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7(4):308–313, 1965.

[71] S. Ontañón, K. Bonnette, P. Mahindrakar, M. A. Gómez-Martín, K. Long, J. Radhakrishnan, R. Shah, and A. Ram. Learning from Human Demonstrations for Real-Time Case-Based Planning. In *Proceedings of the IJCAI Workshop on Learning Structural Knowledge from Observations*, 2009.

[72] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram. Case-Based Planning and Execution for Real-Time Strategy Games. In *Proceedings of ICCBR*, pages 164–178, 2007.

[73] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram. Learning from Demonstra-

tion and Case-Based Planning for Real-Time Strategy Games. *Soft Computing Applications in Industry*, pages 293–310, 2008.

[74] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram. On-Line Case-Based Planning. *Computational Intelligence*, 26(1):84–119, 2010.

[75] S. Ontañón, J. Montana, and A. Gonzalez. Towards a Unified Framework for Learning from Observation. In *Proceedings of the IJCAI Workshop on Agents Learning Interactively from Human Teachers*, 2011.

[76] J. Orkin. Applying Goal-Oriented Action Planning to Games. In S. Rabin, editor, *AI Game Programming Wisdom 2*, pages 217–228. Charles River Media, 2003.

[77] J. Orkin. Three States and a Plan: The AI of FEAR. In *Proceedings of the Game Developers Conference*, 2006.

[78] J. Orkin and D. Roy. The Restaurant Game: Learning Social Behavior and Language from Thousands of Players Online. *Journal of Game Development*, 3(1):39–60, 2007.

[79] J. Orkin and D. Roy. Automatic Learning and Generation of Social Behavior from Collective Human Gameplay. In *Proceedings of AAMAS*, pages 385–392, 2009.

[80] J. Orkin, T. Smith, and D. Roy. Behavior Compilation for AI in Games. In *Proceedings of AIIDE*, pages 162–167, 2010.

[81] R. Palma, P. A. González-Calero, M. A. Gómez-Martın, and P. P. Gómez-Martın.

Extending Case-Based Planning with Behavior Trees. In *Proceedings of FLAIRS*, pages 407–412, 2011.

[82] L. Perkins. Terrain Analysis in Real-Time Strategy Games: Choke Point Detection and Region Decomposition. In *Proceedings of AIIDE*, pages 168–173, 2010.

[83] M. Ponsen. Improving Adaptive Game AI with Evolutionary Learning. Master's thesis, Delft University of Technology, Delft, the Netherlands, 2004.

[84] M. Ponsen, H. Muñoz-Avila, P. Spronck, and D. W. Aha. Automatically Acquiring Domain Knowledge For Adaptive Game AI Using Evolutionary Learning. In *Proceedings of AAAI*, pages 1535–1540, 2005.

[85] L. Pryor and G. Collins. Planning for Contingencies: A Decision-Based Approach. *Journal of Artificial Intelligence Research*, 1996.

[86] R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, California, 1993.

[87] S. Rabin. Implementing a State Machine Language. In S. Rabin, editor, *AI Game Programming Wisdom*, pages 314–320. Charles River Media, 2002.

[88] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice hall, 2009.

[89] F. Schadd, S. Bakkes, and P. Spronck. Opponent Modeling in Real-Time Strategy Games. In *Proceedings of the International Conference on Intelligent Games and Simulation*, pages 61–68, 2007.

[90] J. Schrum and R. Miikkulainen. Evolving Multimodal Networks for Multitask Games. In *Proceedings of IEEE CIG*, pages 102–109, 2011.

[91] C. E. Shannon. Programming a Computer for Playing Chess. *Philosophical magazine*, 41(7):256–275, 1950.

[92] M. Sharma, M. Holmes, J. Santamaria, A. Irani, C. Isbell, and A. Ram. Transfer Learning in Real-Time Strategy Games using Hybrid CBR/RL. In *Proceedings of IJCAI*, 2007.

[93] H. A. Simon. Rational choice and the structure of the environment. *Psychological review*, 63(2), 1956.

[94] C. Simpkins, S. Bhat, C. Isbell Jr, and M. Mateas. Towards Adaptive Programming: Integrating Reinforcement Learning into a Programming Language. In *Proceedings of OOPSLA*, pages 603–614, 2008.

[95] G. Synnaeve and P. Bessiere. A Bayesian Model for Plan Recognition in RTS Games applied to StarCraft. In *Proceedings of AIIDE*, pages 79–84, 2011.

[96] T. Szczepański and A. Aamodt. Case-Based Reasoning for Improved Micromanagement in Real-Time Strategy Games. In *Proceedings of the ICCBR Workshop on Computer Games*, 2009.

[97] S. Thrun. Particle Filters in Robotics. In *Proceedings of the Conference on Uncertainty in AI*, 2002.

[98] C. Thurau. *Behavior Acquisition in Artificial Agents.* PhD thesis, Bielefeld University, 2006.

[99] P. Tozour. *Game AI Programming Wisdom 2*, chapter Using a Spatial Database for Runtime Spatial Analysis, pages 381–390. Charles River Media, 2004.

[100] S. R. Turner. *The Creative Process: A Computer Model of Storytelling and Creativity.* Lawrence Erlbaum Associates, 1994.

[101] H. J. van den Herik, H. Donkers, and P. Spronck. Opponent Modelling and Commercial Games. In *Proceedings of IEEE CIG*, pages 15–25, 2005.

[102] Y. Wang and I. Witten. Inducing Model Trees for Continuous Classes. In *Proceedings of the European Conference on Machine Learning*, pages 128–137, 1997.

[103] B. G. Weber and M. Mateas. A Data Mining Approach to Strategy Prediction. In *Proceedings of IEEE CIG*, pages 140–147, 2009.

[104] B. G. Weber and M. Mateas. Conceptual Neighborhoods for Retrieval in Case-Based Reasoning. In *Proceedings of ICCBR*, pages 343–357, 2009.

[105] B. G. Weber, M. Mateas, and A. Jhala. Applying Goal-Driven Autonomy to StarCraft. In *Proceedings of AIIDE*, pages 101–106, 2010.

[106] B. G. Weber, M. Mateas, and A. Jhala. Case-Based Goal Formulation. In *Proceedings of the AAAI Workshop on Goal-Directed Autonomy*, 2010.

[107] B. G. Weber, M. Mateas, and A. Jhala. A Particle Model for State Estimation in Real-Time Strategy Games. In *Proceedings of AIIDE*, 2011.

[108] B. G. Weber, M. Mateas, and A. Jhala. Building Human-Level AI for Real-Time Strategy Games. In *Proceedings of the AAAI Fall Symposium on Advances in Cognitive Systems*, pages 329–336, 2011.

[109] B. G. Weber, P. Mawhorter, M. Mateas, and A. Jhala. Reactive Planning Idioms for Multi-Scale Game AI. In *Proceedings of IEEE CIG*, pages 115–122, 2010.

[110] B. G. Weber and S. Ontañón. Using Automated Replay Annotation for Case-Based Planning in Games. In *Proceedings of the ICCBR Workshop on Computer Games*, 2010.

[111] S. Wintermute, J. Xu, and J. E. Laird. SORTS: A Human-Level Approach to Real-Time Strategy AI. In *Proceedings of AIIDE*, pages 55–60, 2007.

[112] I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Francisco, California, 2005.

[113] M. J. Wooldridge. *An Introduction to MultiAgent Systems*. Wiley, 2002.

[114] E. Yiskis. A Subsumption Architecture for Character-Based Games. In S. Rabin, editor, *AI Game Programming Wisdom 2*, pages 329–337. Charles River Media, 2003.