Applying Goal-Driven Autonomy to StarCraft

Ben G. Weber, Michael Mateas, and **Arnav Jhala**
Expressive Intelligence Studio
UC Santa Cruz
bweber,michaelm,jhala@soe.ucsc.edu

Abstract

One of the main challenges in game AI is building agents that can intelligently react to unforeseen game situations. In real-time strategy games, players create new strategies and tactics that were not anticipated during development. In order to build agents capable of adapting to these types of events, we advocate the development of agents that reason about their goals in response to unanticipated game events. This results in a decoupling between the goal selection and goal execution logic in an agent. We present a reactive planning implementation of the Goal-Driven Autonomy conceptual model and demonstrate its application in StarCraft. Our system achieves a win rate of 73% against the built-in AI and outranks 48% of human players on a competitive ladder server.

Introduction

Developing agents capable of defeating competitive human players in Real-Time Strategy (RTS) games remains an open research challenge. Improving the capabilities of computer opponents in this area would add to the game playing experience (Buro 2003) and provides several interesting research questions for the artificial intelligence community. How can competitive agents be built that operate in complex, real-time, partially-observable domains which require performing actions at multiple scales as well as reacting to opponents and exogenous events? Current approaches to building game AI are unable to address all of these concerns in an integrated agent. To react to opponents and exogenous events in this domain, we advocate the integration of autonomy in game AI.

Goal-Driven Autonomy (GDA) is a research topic in the AI community that aims to address the problem of building intelligent agents that respond to unanticipated failures and opportunities during plan execution in complex environments (Muñoz-Avila et al. 2010). One of the main focuses of GDA is to develop agents that reason about their goals when failures occur, enabling them to react and adapt to unforeseen situations. Molineaux, Klenk, and Aha (2010) present a conceptual model for GDA which provides a framework for accomplishing this goal.

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

One of the open problems in the GDA community is building systems capable of concurrently reasoning about multiple goals. StarCraft provides an excellent testbed for research in this area, because it is multi-scale, requiring an agent to concurrently reason about and execute actions at several levels of detail (Weber et al. 2010). In StarCraft, competitive gameplay requires simultaneously reasoning about strategic, economic, and tactical goals.

We present an instantiation of the GDA conceptual model implemented using the reactive planning language ABL (Mateas and Stern 2002). Our system, EISBot, plays complete games of StarCraft and uses GDA to concurrently reason at multiple scales. We demonstrate ABL as a candidate for implementing the goal management component in a GDA system. We also show that using GDA to build game AI enables a decoupling of the goal selection and goal execution logic in an agent. Results from our experiments show that EISBot achieves a 73% win rate against the built-in AI and outranks 48% of competitive human players.

Related Work

Two common approaches for building game AI are reactive systems and planning. Reactive systems perform no look-ahead and map game states to actions or behaviors. Reactive techniques for building game AI include finite state machines (FSMs) (Rabin 2002), subsumption architectures (Yiskis 2003), and behavior trees (Isla 2005; Champandard 2008). It is difficult to build agents capable of reacting to unforeseen situations using these techniques, because they do not reason about expectations. Therefore, it is not possible for an agent to detect discrepancies between expected and actual game states. One approach to overcome this limitation is the use a blackboard to model an agent's mental state, enabling the agent to reason about expected game state. Our system differs from this approach in that EISBot explicitly represents discrepancies.

Planning is another technique for building game AI. Goal-Oriented Action Planning (GOAP) (Orkin 2003) is a planning-based approach to building AI for non-player characters in games. 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. Therefore, GOAP maps game states to goals as opposed to actions. One

of the challenges in applying GOAP to game AI is detecting invalid plans, because GOAP systems do not currently generate expectations that can be used to detect discrepancies. Additionally, GOAP architectures are usually applied to only a single level of reasoning. For example, the AI in the RTS game Empire: Total War uses GOAP for strategic decision making, while FSMs are used for individual units[1].

Goal-Driven Autonomy

The goal of GDA is to create agents capable of responding to unanticipated failures that occur during plan execution in complex, dynamic domains. GDA approaches this problem by developing agents that reason about their goals. The research area is motivated by Cox's claim that an agent should reason about itself as well as the world around it in a meaningful way in order to continuously operate with independence (Cox 2007). Games provide an excellent domain for GDA research, because they provide real-time environments with enormous decision complexities. GDA has previously been applied to decision making for FPS bots in a team domination game (Muñoz-Avila et al. 2010).

The GDA conceptual model provides a framework for online planning in autonomous agents (Molineaux, Klenk, and Aha 2010). It consists of several components which enable an agent to detect, reason about, and respond to unanticipated events. The conceptual model outlines the different components and interfaces between them, but makes no commitment to specific implementations. A simplified version of the model is introduced in this paper.

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

The components in our simplified version of the GDA conceptual model are shown in Figure 1. An agent begins with an initial goal, $g0$, which is given to the planner. The planner generates a plan consisting of a set of actions, a, and expectations, x. As actions are executed in the game world, the discrepancy detector checks that the resulting game state, s, meets the expected game state. When a discrepancy is detected, 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, in response to the explanation. The goal is then passed to the goal manager, which is responsible for selecting and executing the current active goal.

The functionality of the GDA conceptual model can be demonstrated in a StarCraft example. Consider an agent that selects an initial strategy of building ground units with a ranged attack. While this strategy is effective against most early game ground and air-based armies, it is weak against a fast expanding strategy or an opponent that focuses on producing cloakable units as fast as possible. Given the agent's selected strategy, it has expectations that the opponent will not build a fast expansion and that the opponent will not build cloakable units. During the game, the agent scouts a unit type that it has not yet encountered in the game. In response to this event, the discrepancy detector generates a discrepancy that an unexpected unit type was observed. If the scouted unit type violates the expectations of the current strategy, an explanation is generated. In this example, scouting a building that enables an opponent to train cloakable units would cause the explanation generator to create an explanation that the opponent is pursuing cloakable units. The explanation is given to the goal formulation component, which formulates the goal of building detector units in order to have vision of cloaked units. Finally, the goal is given to the planner and the agent produces a plan to train detector units. This example demonstrates that using GDA enables the agent to react to unexpected opponent actions.

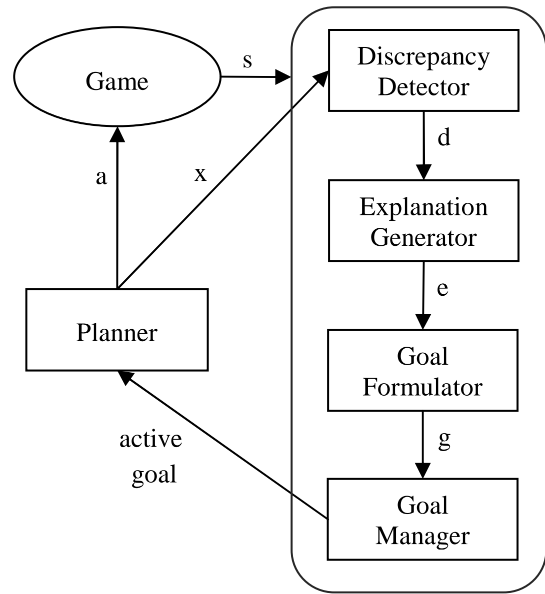

Figure 1: Components in the simplified Goal-Driven Autonomy conceptual model

Applying GDA to StarCraft

StarCraft is a science fiction RTS game developed by Blizzard Entertainment™ in which players manage an economy, produce units and buildings, and vie for control of the map with the goal of destroying all opponents. To perform well in this game, an agent must react to events at the strategic, economic and tactical levels. We applied GDA to StarCraft to determine when new goals should be selected and to decide which goals should be pursued at each of these levels.

EISBot uses the ABL reactive planning language to implement the components specified in the GDA conceptual model. ABL is well suited for building RTS game AI, because it was precisely designed to combine reactive, parallel

[1] http://worthplaying.com/article/2008/12/1/interviews/57018/

```
sequential behavior detect() {
 precondition {
  (EnemyUnit type::type)
  !(UnitTypeDiscrepancy type==type)
 }

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

Figure 2: An ABL behavior for detecting new unit types

```
sequential behavior explain() {
 precondition {
  (UnitTypeDiscrepancy type==LurkerEgg)
  !(Explanation type==EnemyCloaking)
 }

 mental_act {
  workingMemory.add(new Explanation(
      EnemyCloaking));
 }
}
```

Figure 3: A behavior that generates an explanation that the opponent is building cloaked units in response to noticing a lurker egg.

goal pursuit with long-term planfulness (Mateas and Stern 2002). Additionally, ABL supports concurrent action execution, represents event-driven behaviors, and provides a working memory enabling message passing between components.

Our system contains a collection of behaviors for each of the GDA components. The agent persistently pursues each of these behaviors concurrently, enabling the agent to quickly respond to events. The majority of the agent's functionality is contained in the goal manager component, which is responsible for executing the agent's current goals. The different components communicate using ABL's working memory as a blackboard. Each of the components is discussed in more detail below.

Discrepancy Detector

The discrepancy detector generates discrepancies when the agent's expectations are violated. In contrast to previous work that creates a new set of expectations for each generated plan, our system has a fixed set of expectations. Also, EISBot does not explicitly represent expectations, because there is a direct mapping between expectations and discrepancies in our system. Instead, the agent has a fixed set of discrepancy detectors that are always active.

Discrepancies serve the purpose of triggering the agent's goal reasoning process and provide a mechanism for responding to unanticipated events. Our system 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 2. The 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 is 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.

Explanation Generator

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

- Opponent is teching
- Opponent is building air units
- Opponent is building cloaked units
- Opponent is building detector units
- Opponent is expanding
- Agent has 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. At the strategic level, explanations are created only for discrepancies that violate the agent's current high-level strategy. An example behavior for generating explanations is shown in Figure 3. The behavior checks if the opponent has morphed any lurker eggs, which hatch units capable of cloaking. In response to detecting a lurker egg, the agent creates an explanation that the opponent is building cloakable units.

Goal Formulator

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 the 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,

```
sequential behavior formulateGoal() {
 precondition {
  (Explanation type==ForceAdvantage)
 }

 spawngoal expand();
 spawngoal attack();
}
```

Figure 4: A behavior that spawns goals for expanding and attacking the opponent in response to an explanation that the agent has a force advantage.

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 4. The behavior spawns goals to attack and expand in response to an explanation that the agent has a force advantage. In ABL, spawngoal is analogous to creating a new thread of execution, and enables the agent to pursue a new goal in addition to the currently active goal.

Goal Manager

In our system, the goal manager and planner components from Figure 1 are merged into a single component. Goals that are selected by the goal formulation component immediately begin execution by pursuing behaviors that match the spawned goal name. While our system supports concurrent goal execution, only a single goal can be active at each of the strategic, economic, and tactical levels. The agent can pursue the goal of expanding while attacking, but cannot pursue two tactical goals simultaneously. For example, the current agent cannot launch simultaneous attacks on different areas of the map.

The goal manager is based on the integrated agent architecture of McCoy and Mateas (2008). It is composed of several managers that handle distinct aspects of StarCraft gameplay. The strategy manager handles high-level decision making, which includes determining which structures to build, units to produce, and upgrades to research. The strategy manager actively pursues one of the following high-level strategies:

- Mass Zealots: focuses on tier 1 melee units
- Dragoon Range: produces tier 1 range units
- Observers: builds detectors and range units
- Carriers: focuses on building air units
- Dark Templar: produces cloaked units

The current goal to pursue is selected by the goal formulator by spawning *Execute Strategy* goals.

Expansion goals are carried out by the income manager. The manager is responsible for producing the expansion building as well as training and assigning worker units at the expansion. The attack and retreat goals are handled by the tactics manager. Given an attack goal, the manager sends all combat units to the opponent base using the attack move command. To retreat, the manager sends all combat units to the agent's base.

Agent Architecture

EISBot is implemented using the ABL reactive planning language. Our architecture builds upon the integrated agent framework (McCoy and Mateas 2008), which plays complete games of Wargus. While there are many differences between Wargus and StarCraft, the conceptual partitioning of gameplay into distinct managers transfers well between the games. We made several changes to the managers to support the StarCraft tech tree and added additional behaviors to the agent to support micromanagement of units (Weber et al. 2010). Currently, the agent plays only the Protoss race.

An overview of the agent architecture is shown in Figure 5. The ABL agent has two collections of behaviors which perform separate tasks. The GDA behaviors are responsible for reacting to events in the game world and selecting which goals should be pursued, while the manager behaviors are responsible for executing the goals selected by the GDA behaviors. The ABL components communicate using ABL's working memory as a blackboard (Isla et al. 2001). By utilizing the GDA conceptual model, we were able to cleanly separate the goal selection and goal execution logic in our agent.

The agent interacts with StarCraft using the BWAPI interface[2]. Brood War API is a recent project that exposes the underlying interface of StarCraft, allowing code to directly view game state, such as unit health and locations, and to issue orders, such as movement commands. This library is written in C++ and compiles into a dynamically linked library that is launched in the same process space as StarCraft. Our ABL agent is compiled to Java code, which runs as a separate process from StarCraft. ProxyBot is a Java component that provides a remote interface to the BWAPI library using sockets. Every frame, BWAPI sends a game state update to the ProxyBot and waits for a response containing a set of commands to execute.

Evaluation

We evaluated our GDA approach to building game AI by applying it to the task of playing complete games of StarCraft. Currently, EISBot plays only the Protoss race. The system was tested on a variety of maps against both the built-in AI of StarCraft as well as human opponents on a ladder server. We also plan on evaluating the performance of EISBot by participating in the AIIDE 2010 StarCraft AI Competition[3].

The map pool used to evaluate EISBot is the same as the pool that will be used in tournament 4 of the StarCraft AI competition. It includes maps that support two to four players and encourages a variety of play styles. For example, the mineral-only expansion on Andromeda encourages macro-focused gameplay, while the easy to defend ramps on Python encourage the use of dropships. A detailed analysis

[2] http://code.google.com/p/bwapi/

[3] http://eis.ucsc.edu/StarCraftAICompetition

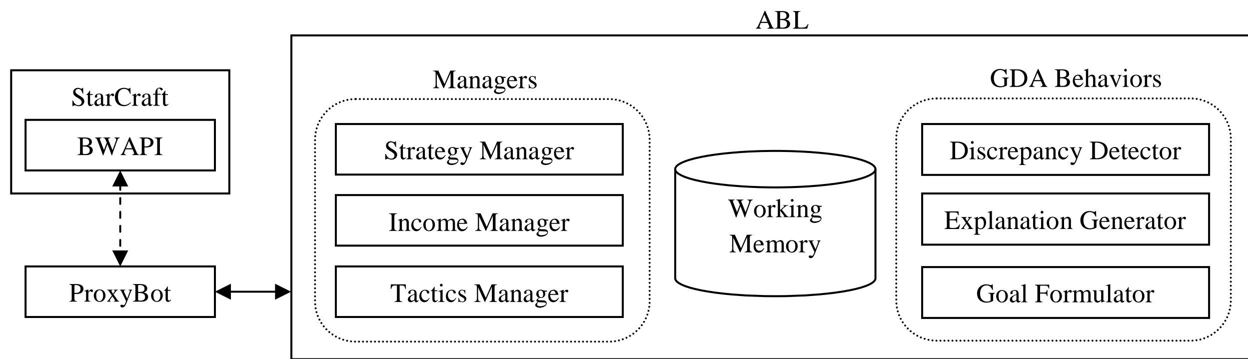

Figure 5: Agent architecture

Table 1: Win rates versus the built-in AI

	Versus			
	Protoss	Terran	Zerg	Overall
Andromeda	65%	65%	45%	58%
Destination	50%	85%	75%	70%
Heartbreak Ridge	75%	95%	85%	85%
Python	65%	90%	70%	75%
Tau Cross	65%	95%	70%	77%
Overall	64%	86%	69%	73%

Table 2: Results versus human opponents

	Versus			
	Protoss	Terran	Zerg	Overall
Win-loss record	10-23	8-21	19-19	37-63
Win ratio	30%	28%	50%	37%
ICCup Points: 1182				
ICCup Rank: 33,639 / 65,646				

of the characteristics of the maps and the gameplay styles they support is available at Liquipedia[4].

The first experiment evaluated EISBot versus the built-in AI of StarCraft. The default StarCraft AI works by selecting a specific script to run at the beginning of a match and then executing that script. For each race, there are one or more scripts that can be executed. For example, a Protoss computer opponent will either perform a mass zealot timing attack or a dark templar rush. The AI attacks in waves, which commit units to attacking the player and never retreating. Players are able to defeat the built-in AI by building sufficient defenses to defend against the initial rush while gaining an economic or strategic advantage over the AI.

Results from the first experiment are shown in Table 1. Overall, the agent achieved a win rate of 73% against the built-in AI. To ensure that a variety of scripts were executed by the AI, 20 games were run on each map for each opponent race. In total, 300 games were run against the built-in AI. EISBot performed best against Terran opponents, which execute a single fixed strategy. The agent performed worse against the other races due to well-executed timing attacks by the opponent. For example, the Zerg opponent will often 4-pool rush against the agent, which is the fastest rush possible in StarCraft.

There are two maps that had average win rates that were different from the rest of the pool. The agent performed best on Heartbreak Ridge, which resulted from the short distance between bases and lack of ramps. EISBot performed worst on Andromeda, which resulted from the large distance between bases and easy to defend ramps. EISBot performed better on smaller maps, because it was able to attack the opponent much quicker than on larger maps. Additionally, EISBot performed better on maps without ramps, due to a lack of behaviors for effectively moving units as groups.

The second experiment evaluated EISBot against human opponents. Games were hosted on the International Cyber Cup (ICCup)[5], a ladder server for competitive StarCraft players. All games on ICCup were run using the map Tau Cross, which is a three player map with no ramps. The results from the second experiment are shown in Table 2. EISBot achieved a win rate of 37% against competitive humans. The agent performed best against Zerg opponents, achieving a win rate of 50%. A screen capture of EISBot playing against a Zerg opponent is shown in Figure 6. Videos of EISBot versus human opponents are available online[6].

The International Cyber Cup has a point system similar to the Elo rating system in chess, where players gain points for winning and lose points for losing. Players start at a provisional 1,000 points. After 100 games, EISBot achieved a score of 1182 and was ranked 33,639 out of 65,646. Our system outperformed 48% of competitive players. The ability of the system to adapt to the opponent was best illustrated when humans played against EISBot multiple times. There were two instances in which a player that previously defeated EISBot lost the next game.

[4]http://wiki.teamliquid.net/starcraft/

[5]http://www.iccup.com

[6]http://www.youtube.com/user/UCSCbweber

Figure 6: EISBot (orange) attacking a human opponent

Conclusion and Future Work

We have presented an approach for integrating autonomy into a game playing agent. Our system implements the GDA conceptual model using the ABL reactive planning language. This approach enables our system to reason about and react to unanticipated game events. We provided an overview of the GDA conceptual model and discussed how each component was implemented. Rather than map states directly to actions, our approach decouples the goal selection and goal execution logic in our agent. This enables the system to incorporate additional techniques for responding to unforeseen game situations.

EISBot was evaluated against both the built-in AI of StarCraft as well as human opponents on a competitive ladder server. Against the built-in AI, EISBot achieved a win rate of 73%. Against human opponents, it achieved a win rate of 37% and outranked 48% of players after 100 games.

While our initial results are encouraging, there are a number of ways in which EISBot could be improved. Future work could focus on adding more behaviors to the strategy and tactics managers in our agent. EISBot does not currently have the capability to fully expand the tech tree. Also, several behaviors are missing from our agent, such as the ability to utilize transport units and efficiently move forces through chokepoints.

Currently, EISBot has a small library of discrepancies, explanations, and goals. Increasing the size of this library would enable our system to react to more types of events. Additional explanations could be generated by analyzing how humans describe gameplay (Metoyer et al. 2010) or utilizing richer representations (Hoang, Lee-Urban, and Muñoz-Avila 2005).

Another possible research direction is to automate the process of building discrepancies, explanations, and goals. The current implementations of the GDA components utilize trigger rules for responding to events. Future work could utilize opponent modeling techniques to build explanations of the opponent's actions (Weber and Mateas 2009), and learning from demonstration to formulate new goals to pursue.

References

Buro, M. 2003. Real-Time Strategy Games: A New AI Research Challenge. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1534–1535.

Champandard, A. 2008. Getting Started with Decision Making and Control Systems. In Rabin, S., ed., *AI Game Programming Wisdom 4*. Charles River Media. 257–264.

Cox, M. 2007. Perpetual Self-Aware Cognitive Agents. *AI Magazine* 28(1):32–45.

Hoang, H.; Lee-Urban, S.; and Muñoz-Avila, H. 2005. Hierarchical Plan Representations for Encoding Strategic Game AI. In *Proceedings of Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press.

Isla, D.; Burke, R.; Downie, M.; and Blumberg, B. 2001. A Layered Brain Architecture for Synthetic Creatures. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1051–1058.

Isla, D. 2005. Handling Complexity in the Halo 2 AI. In *Proceedings of the Game Developers Conference*.

Mateas, M., and Stern, A. 2002. A Behavior Language for Story-Based Believable Agents. *IEEE Intelligent Systems* 17(4):39–47.

McCoy, J., and Mateas, M. 2008. An Integrated Agent for Playing Real-Time Strategy Games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 1313–1318. AAAI Press.

Metoyer, R.; Stumpf, S.; Neumann, C.; Dodge, J.; Cao, J.; and Schnabel, A. 2010. Explaining How to Play Real-Time Strategy Games. *Knowledge-Based Systems* 23(4):295–301.

Molineaux, M.; Klenk, M.; and Aha, D. W. 2010. Goal-Driven Autonomy in a Navy Strategy Simulation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 1548–1554. AAAI Press.

Muñoz-Avila, H.; Aha, D. W.; Jaidee, U.; Klenk, M.; and Molineaux, M. 2010. Applying Goal Driven Autonomy to a Team Shooter Game. In *Proceedings of the Florida Artificial Intelligence Research Society Conference*, 465–470. AAAI Press.

Orkin, J. 2003. Applying Goal-Oriented Action Planning to Games. In Rabin, S., ed., *AI Game Programming Wisdom 2*. Charles River Media. 217–228.

Rabin, S. 2002. Implementing a State Machine Language. In Rabin, S., ed., *AI Game Programming Wisdom*. Charles River Media. 314–320.

Weber, B., and Mateas, M. 2009. A Data Mining Approach to Strategy Prediction. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 140–147. IEEE Press.

Weber, B.; Mawhorter, P.; Mateas, M.; and Jhala, A. 2010. Reactive Planning Idioms for Multi-Scale Game AI. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, To appear. IEEE Press.

Yiskis, E. 2003. A Subsumption Architecture for Character-Based Games. In Rabin, S., ed., *AI Game Programming Wisdom 2*. Charles River Media. 329–337.