Reactive Planning Idioms for Multi-Scale Game AI

Ben G. Weber, Peter Mawhorter, Michael Mateas, and Arnav Jhala

Abstract— **Many modern games provide environments in which agents perform decision making at several levels of granularity. In the domain of real-time strategy games, an effective agent must make high-level strategic decisions while simultaneously controlling individual units in battle. We advocate reactive planning as a powerful technique for building multi-scale game AI and demonstrate that it enables the specification of complex, real-time agents in a unified agent architecture. We present several idioms used to enable authoring of an agent that concurrently pursues strategic and tactical goals, and an agent for playing the real-time strategy game StarCraft that uses these design patterns.**

I. Introduction

Game AI should exhibit behaviors that demonstrate intelligent decision making and which work towards long-term goals. In the domain of real-time strategy (RTS) games, an agent must also reason at multiple granularities, making intelligent high-level strategic decisions while simultaneously micromanaging units in combat scenarios. In fact, many modern video games require agents that act at several levels of coordination, behaving both individually and cooperatively. We explore the use of reactive planning as a tool for developing agents that can exhibit this type of behavior in complex, real-time game environments.

A multi-scale game AI is a system that reasons and executes actions at several granularities. To achieve effective multi-scale game AI, a system must be able to reason about goals across different granularities, and be able to reason about multiple goals simultaneously, including both dependent and independent goals. In RTS games, a competitive agent is required to perform micromanagement and macromanagement tasks, which makes this a suitable domain for the application of multi-scale systems. At the micromanagement level, individual units are meticulously controlled in combat scenarios to maximize their effectiveness. At the macromanagement level, the agent works towards long-term goals, such as building a strong economy and developing strategies to counter opponents. Similar distinctions between different scales of reasoning can be made in other genres of games as well, such as the difference between squad management and individual unit behavior in first person shooters [1]. One of the major challenges in building game AI is developing systems capable of multi-scale reasoning.

A common approach to this problem is to abstract the different scales of reasoning into separate layers and build interfaces between the layers, or simply implement one layer as complex actions within another. These layered architectures raise difficulties when there is not a clear separation between the different scales of reasoning, or when decision processes at one scale need to communicate with another. In RTS games, a unit may participate in both individual micromanagement and squad-based actions, which forces different systems to coordinate their actions. Another issue that arises from a layered architecture is that different layers may compete for access to shared in-game resources, resulting in complicated inter-layer messaging that breaks abstraction boundaries. Because of these issues, we claim that multi-scale reasoning in game AI is most productive when a unified architecture is used.

Reactive planners provide such an architecture for expressing all of the processing aspects of an agent [2]. Unified agent architectures are effective for building multi-scale game AI, because they deal with multiple goals across scales, execute sequences of actions as well as reactive tasks, and can re-use the same reasoning methods in different contexts [3].

We introduce several AI design patterns (of both classic and more novel varieties), programmed in the reactive planning language ABL [4], which address the problems of multi-scale reasoning. By explaining these idioms here, we hope to both demonstrate their usefulness in solving multi-scale AI problems, and to give illustrative, concrete examples so that others can employ these idioms. In order to validate our use of these idioms and present concrete examples of them, we also present our current progress on the EISBot, an AI system that uses the presented design patterns to play the real-time strategy game StarCraft.

II. Related Work

Techniques such as finite state machines (FSMs) [5], behavior trees [6], subsumption architectures [7], and planning [8] have been applied to game AI. FSMs are widely used for authoring agents due to their efficiency, simplicity, and expressivity. However, FSMs do not allow logic to be reused across contexts and designers end up duplicating state or adding complex transitions [9]. Additionally, finite state machines are generally constrained to a single active node at any point in time, which limits their ability to support parallel decision making and independent reasoning about multiple goals.

Behavior trees provide a method for dealing with the complexity of modern games [6], and have a more modular structure than finite state machines. An agent is specified as a hierarchical structuring of behaviors that can be performed, and agents make immediate decisions about actions to pursue using the tree as a reasoning structure. The main limitation of behavior trees is that they are designed to specify

Ben Weber, Peter Mawhorter, Michael Mateas, and Arnav Jhala are with the Expressive Intelligence Studio at the University of California, Santa Cruz. 1156 High Street, Santa Cruz, CA, USA (email: bweber,pmawhort,michaelm,jhala@soe.ucsc.edu).

behavior for a single unit, which complicates attempts to reason about multiple goals simultaneously. Achieving squad behaviors with behavior trees requires additional components for assigning units to squads, as well as complicated per-unit behaviors that activate only within a squad context [1]. This approach is not suitable for general multi-scale game AI, because a separate mechanism is required to communicate between behavior trees for various levels of reasoning, which introduces the problems of layered game AI systems.

In subsumption architectures, lower levels take care of immediate goals and higher levels manage long term goals [7]. The levels are unified by a common set of inputs and outputs, and each level acts as a function between them, overriding higher levels. This approach is good for reasoning at multiple granularities, but not good at reasoning about multiple separate goals simultaneously. Because the RTS domain requires both of these capabilities, subsumption architectures are not ideal.

Planning is another approach for authoring agent behavior. Using classical planning requires formally modeling the domain and defining operators for game actions with preconditions and postconditions. The AI in F.E.A.R. builds plans using heuristic-guided forward search through such operators [8]. One of the difficulties in using planning is specifying the game state logically and determining the preconditions and effects of operators. Classical planning is challenging to apply to game AI, because plans can quickly become invalidated in the game world. F.E.A.R. overcomes this by intermixing planning and execution and continuously replanning. Multi-scale game AI is difficult to implement with classical planning, because plan invalidation can occur at multiple levels of detail within a global plan, and separate plans for specific goals would suffer from the same synchronization problems as a layered architecture.

Reactive planning avoids the problems with classical planning by being decompositional rather than generative. Similar to hierarchical task networks [10], a reactive planner decomposes tasks into more specific sub-tasks, which combine into an ultimate plan of action. In a reactive planner, however, task decomposition is done incrementally in real time, and task execution occurs simultaneously. Reactive planning has been used successfully to author complex multi-agent AI in Façade [11], and has been used to build an integrated agent for the real-time strategy game Wargus [12].

Another promising approach is cognitive architectures, which are based on a human-modeling paradigm. SOAR [13], [14] and ICARUS [15] are cognitive architectures that have been applied to game AI. These systems are examples of unified agent architectures that address the multi-scale game AI problem. The main difference between these systems and our implementation is that cognitive architectures make strong claims about modeling human cognitive processes, while our approach, although it encodes the domain knowledge of human experts, does not make such claims. However, the idioms and techniques for multi-scale RTS AI described in this paper could be fruitfully applied in architectures such as SOAR and ICARUS.

In the domain of RTS games, computational intelligence has been applied to tactics using Monte Carlo planning [16], strategy selection using neuroevolution of augmenting topologies (NEAT) [17], and resource allocation using coevolution of influence map trees [18]. However, each of these approaches reasons at only a single level of granularity and needs to be integrated with additional techniques to perform multi-scale reasoning.

III. StarCraft

One of the most notoriously complex games that requires multi-scale reasoning is the real-time strategy game StarCraft[1]. StarCraft is a game in which players manage groups of units to vie for control of the map by gathering resources to produce buildings and more units, and by researching technologies that unlock more advanced buildings and units. Building agents that perform well in this domain is challenging due to the large decision complexity [19]. StarCraft is also a very fast-paced game, with top players exceeding 300 actions per minute during peak play [12]. This means that a competitive agent for StarCraft must reason quickly at multiple granularities in order to demonstrate intelligent decision making.

Our choice of StarCraft as a domain has additional motivating factors: Despite being more than 10 years old, the game still has an ardent fanbase, and there is even a professional league of StarCraft players in Korea[2]. This indicates that the game has depth of skill, and makes evaluation against human players not only possible, but interesting.

Real-time strategy games in general (and StarCraft in particular) provide an excellent environment for multi-scale reasoning, because they involve low-level tactical decisions that must complement high-level strategic reasoning. At the strategic level, StarCraft requires decision-making about long-term resource and technology management. For example, if the agent is able to control a large portion of the map, it gains access to more resources, which is useful in the long term. However, to gain map control, the agent must have a strong combat force, which requires more immediate spending on military units, and thus less spending on economic units in the short term.

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

At the tactical level, effective StarCraft gameplay requires both micromanagement of individual units in small-scale combat scenarios and squad-based tactics such as formations. In micromanagement scenarios, units are controlled individually to maximize their utility in combat. For example, a

[1]StarCraft and its expansion StarCraft: Brood War were developed by Blizzard Entertainment™

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

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. At the same time, these goals cannot simply be imposed on the unit: in order to properly position the cheap sacrificial units, knowledge about the location of the more expensive units must be processed.

StarCraft gameplay requires simultaneous decision making at both small and large scales. Building layers and interfaces between these scales is difficult for StarCraft, because different layers may compete for shared resources, such as control of a unit. Additionally, a single unit may be concurrently pursuing local, group and global goals in coordination with other units. Besides multiple scales, StarCraft also involves multiple independent goals: two harassing units on opposite sides of the map may need to pursue similar local objectives completely independently.

These challenges motivate the use of reactive planning, which can concurrently pursue many goals at multiple granularities. Our agent, the EISBot, uses reactive planning to manage and coordinate simultaneous low-level tactical actions and high-level strategic reasoning.

IV. ABL

Our agent, the EISBot, is implemented in ABL [4]. ABL (A Behavior Language), a reactive planning language based on Hap [2], adds significant features to the original Hap semantics. These include first-class support for meta-behaviors (behaviors that manipulate the runtime state of other behaviors) and for joint intentions across teams of multiple agents [20]. ABL is effective for building multi-scale game AI, because it enables agents to pursue multiple goals concurrently and provides mechanisms for facilitating communication between behaviors. Importantly, it also supports the design patterns presented in this paper as relatively simple code structures.

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 an authored collection. A behavior contains a set of preconditions which specify whether it can be executed given the current world state. There is also an optional specificity associated with behaviors that assigns a priority, and behaviors with higher specificities are selected for execution before considering lower specificity behaviors.

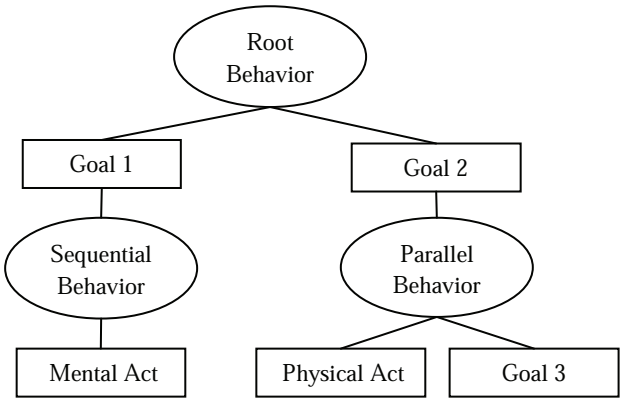

Fig. 1. An example active behavior tree (ABT)

During the execution of an ABL agent, all of the goals an agent is pursuing are stored in the active behavior tree (ABT) [2]. 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 other behaviors. When a node is selected, its component steps are placed in the ABT as the children of the goal. An example ABT is shown in Figure 1.

A core component of ABL agents is working memory. ABL's working memory serves as a blackboard for maintaining the 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 system [21]. Using working memory as a blackboard enables several idioms for authoring agents in ABL.

One of the benefits of using ABL to author game AI is that scheduling of actions is handled by the planner. 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 (steps that are part of parallel behaviors in the ABT continue). Therefore, a separate scheduling component is not necessary for scheduling the actions selected by an ABL agent.

V. ABL Semantics

In this section, we introduce ABL semantics in order to familiarize the reader with concepts in reactive planning and build a foundation for discussing the concrete implementation of AI design patterns in ABL.

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. Goals are represented by

behaviors in ABL: each behavior represents actions (and/or other behaviors) 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 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 2. 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 performing of a physical act that prints to the console.

```
initial_tree {
  subgoal sayHello();
}

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

Fig. 2. An agent with the goal of saying hello

ABL behaviors can be sequential or parallel. When a behavior is selected for expansion, its components 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.

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

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

Fig. 3. Behavior preconditions

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 3. 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 (WMEs) 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 health. Behavior parameterization is a powerful tool for reusing behaviors across multiple contexts.

Behaviors can perform mental and physical acts. Mental acts are small chunks of agent processing 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. Physical acts are actual actions performed by the agent in the game. For example, the attackUnit act in Figure 3 will cause the player's unit to attack an enemy unit. 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.

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

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

Fig. 4. Spawngoal and persistent keywords

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. 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 [2], 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 suc-

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

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

Fig. 5. Success tests and context conditions

ceeds. 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 5. 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.

VI. Design Patterns

We present several design patterns that enable authoring of multi-scale game AI. These design patterns facilitate development of agents that are capable of reasoning at many granularities while simultaneously reacting to events. Each of these idioms is realized in the EISBot, enabling the agent to concurrently pursue high-level strategic goals while simultaneously reacting to unit-specific events. These patterns, however, are general and could be used to facilitate robust multi-scale AI development in other AI systems. We discuss them here as programmed in ABL in order to give concrete examples of their instantiation and use.

A. Daemon Behaviors

A multi-scale system must be able to reason about several goals simultaneously. In ABL, this is 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.

The EISBot uses daemon behaviors to spawn new threads of execution for managing subtasks. An example daemon behavior for managing worker units is shown in Figure 4. The initializeAgent behavior spawns the daemon behavior incomeManager that continuously pursues resource collection in parallel with the agent's other goals.

B. Messaging

Communication is necessary to facilitate coordination between different behaviors in a multi-scale AI system. In ABL, several messaging idioms are possible by using working memory as an internal mental blackboard [22]. The EISBot uses message passing idioms to support the decoupling of different components.

Common messaging patterns in ABL are the message producer and message consumer idioms. 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 manipulated by mental acts. An example of the message producer and message consumer idioms are shown in Figure 6. The strategyManager behavior is a message producer that adds a construction WME to working memory and the constructionManager behavior is a message consumer that removes the construction WME from working memory.

C. Managers

One of the challenges of building multi-scale game AI is authoring several different aspects of a game within a single agent. 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 managing a distinct subset of the agent's behavior. Managers can use message passing to coordinate behaviors between their various domains. This partitioning helps to take advantage of sophisticated domain knowledge developed by human players, and also increases code modularity which eases development [12].

The EISBot is split into several managers based on analysis of expert StarCraft gameplay. For example, our agent has a high-level strategy manager that makes decisions about what buildings to build, but does not reason about where to place them or how to issue orders to build them. This separation of different reasoning levels between different managers makes the reasoning easier to author. When writing rules for high-level strategic decisions, the programmer does not have to think about the details of building placement or order sequences.

D. Micromanagement Behaviors

The EISBot combines high-level decision making with reactive unit-level tasks. This is achieved through the use of micromanagement behaviors in ABL. Micromanagement behaviors are an idiom for implementing highly compartmentalized behaviors in an agent. While managers perform high-level decision making, micromanagement behaviors perform reactive low-level tasks, and are especially useful for specifying per-unit behavior in a domain where an agent controls multiple units.

A micromanagement task is instantiated by using spawngoal to create a new goal for managing a specific unit. An example behavior for micromanaging vultures is shown in Figure 7. The spawnMicroTask behavior waits for new

```
sequential behavior strategyManager() {
  // precondition check

  mental_act {
    WorkingMemory.add(
      new ConstructionWME(factory));
  }
}

sequential behavior constructionManager() {
  precondition {
    construction = (ConstructionWME)
  }

  mental_act {
    WorkingMemory.delete(construction);
  }

  // construct factory
}
```

Fig. 6. An example of a manager that uses message passing

```
sequential behavior spawnMicroTask() {
  with success_test {
    vulture = (VultureWME)
  }
  wait;

  spawngoal micromanageVulture(vulture);
}

sequential behavior micromanageVulture(
      VultureWME  vulture) {

  context_condition{
    (vulture.isAlive())
  }

  mental_act {
    WorkingMemory.delete(vulture);
  }

  // micromanage vulture
}
```

Fig. 7. An example micromanagement task

vultures to appear in the game and spawns a new micromanageVulture for each vulture. This is accomplished by parameterizing the micromanagement behavior, and passing it a reference to the discovered unit when it is spawned. The context condition is specified so the behavior terminates if the unit is no longer alive.

VII. EISBot

The EISBot is an agent that plays StarCraft: Brood War. By developing a competitive agent in a difficult domain, we aim to fully explore the capabilities of reactive planning, and to find effective ways to use this technique to build a multi-scale agent. EISBot is also a concrete instantiation of the design patterns discussed above, and to the extent that it works, validates their applicability to multi-scale AI.

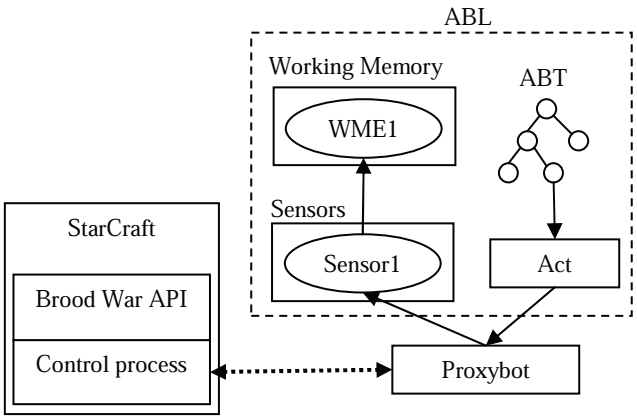

Fig. 8. EISBot StarCraft interface

A. Connecting EISBot to StarCraft

Our agent consists of a bridge between a game instance and a control process, as well as an ABL-based agent that senses game state and issues commands to the game. The bridge has two main components. The first, Brood War API, is a recent project that exposes the underlying interface of the game, allowing code to directly view game state, such as unit health and locations, and to issue orders, such as movement commands. This is written in C++, which compiles into a dynamically linked library. Within this library, our system has hooks that export relevant data and convey commands. These hooks use a socket to connect to the ProxyBot, our Java-based agent. The ProxyBot handles game start and end events, and marshals the incoming information from the game process to make it available to ABL as a collection of working memory elements. Our agent, compiled from ABL into Java code, runs on these elements, and issues orders through the ProxyBot back over the socket to the Brood War API running in the game process. The interface between the ABL agent and StarCraft is shown in Figure 8.

B. Agent Architecture

Our agent architecture is based on the integrated agent framework of McCoy and Mateas [12], 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. Our agent is composed of several managers: a strategy manager is responsible for high-level strategic decisions such as when to initiate an attack, an income manager is responsible for managing the agent's economy and worker population, a production manager constructs production buildings and trains units and the tactics manager manages squads of units. The main difference between our agent and the Wargus agent is the addition of squad-based tactics and unit micromanagement techniques. We also added a construction manager to handle the complexity of building construction in StarCraft.

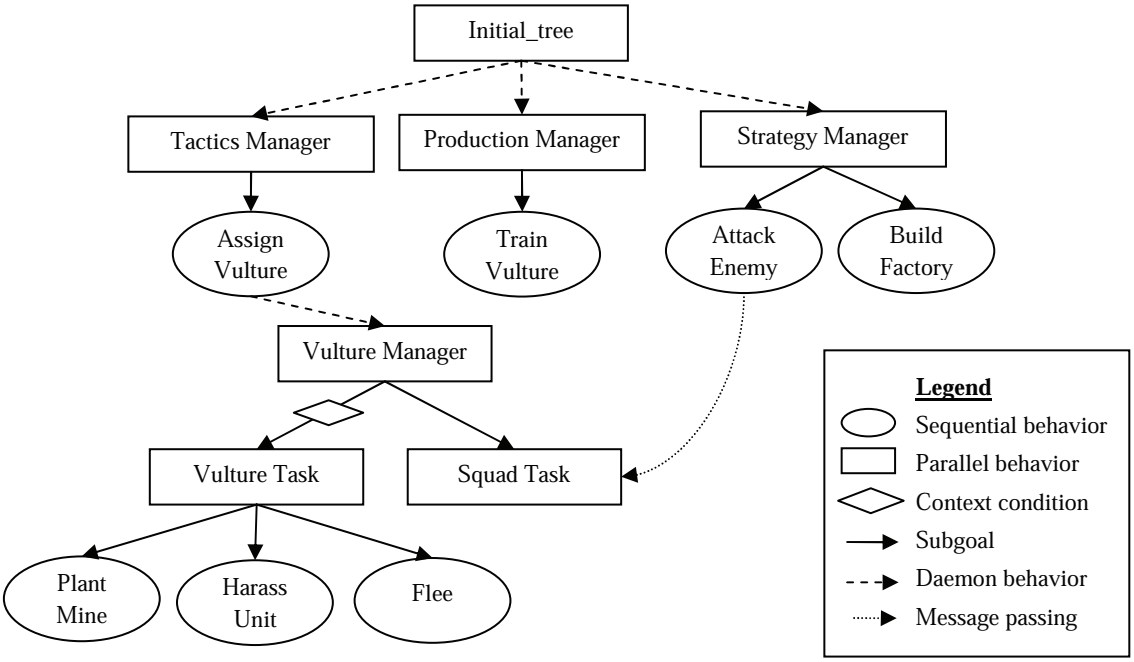

Fig. 9. A subset of the agent's behaviors. The root behavior starts several daemon processes which manage distinct subgoals of the agent. The assign vulture behavior spawns micromanagement behaviors for individual vultures. The messaging passing arrow shows the communication of squad behaviors between the strategy manager and individual units.

C. Design Patterns in Our Agent

As discussed in section 6, we used ABL design patterns in our agent to separate high-level and low-level reasoning, and to isolate behaviors related to different in-game systems, such as resource management and combat. But we also used these idioms to unify systems in some cases, and to manage different aspects of the game at appropriate levels.

To address high-level strategic reasoning needs, our strategy manager makes decisions about what buildings to build. However, the construction manager handles the details of building placement and specific unit orders when instructed to build something by the strategy manager. Thus we employ manager and message-passing patterns to help make the code more modular and to effectively reason cooperatively about a task like construction.

Squad-based tactics and micromanagement are handled by the tactics manager. For some units, each individual unit has its own behavior hierarchy that directs its actions, authored using the micromanagement behavior pattern. This strategy is effective for quick harassing units that operate independently. However, it becomes cumbersome when coordinated tactics are required, because individual units cannot reason efficiently about the context of the entire battle. For this reason, some units are managed in groups, using behaviors written at the squad level.

In StarCraft, vultures are a versatile unit effective for harassing enemy melee units, laying mine fields and supporting tanks. These tasks require different levels of cooperation. When harassing enemy forces, vultures are controlled at a per-unit level to avoid taking damage from enemy melee units. When supporting tanks, vultures work as a squad and provide the first line of defense. These dual roles exemplify the problem of multi-scale reasoning.

A subset of the agent's behaviors is shown in Figure 9. The root behavior starts several daemon behaviors that spawn the different managers. Each manager then continuously pursues a set of subgoals concurrently. In this example, the strategy manager is responsible for deciding when to produce factories and when to attack the opponent, the production manager constantly trains vultures, and the tactics manager spawns micromanagement behaviors for produced vultures.

The agent coordinates squad behavior through the use of squad WMEs. The attack enemy behavior is a message producer that adds a squad WME to working memory when executed. The squad task behavior is an event-driven behavior that reacts to squad WMEs. Upon retrieval of a squad WME, a vulture will abort any micromanagement task that it is engaged in, and defer to orders issued by a squad behavior. This is accomplished by a context condition within the micromanagement behavior that suspends it when the vulture is assigned to a squad. The key difference between this scheme and one where a squad-specific behavior was implemented within the micromanagement behavior is that the squad behavior reasons at a higher level than the individual unit, and so it can give an order to a particular vulture based on a larger context. The EISBot manages individual units as well as the formulation of squads within a unified environment, enabling the agent to dynamically assign units to roles based on the current situation.

By using managers, daemon behaviors, and message passing, our agent is able to reason about different goals at different scales simultaneously, and to coordinate that reasoning

TABLE I

WIN RATES ON THE MAP POOL OVER 20 TRIALS

	Versus		
	Protoss	Terran	Zerg
Andromeda	85%	55%	75%
Destination	60%	60%	45%
Heartbreak Ridge	70%	70%	75%
Overall	72%	62%	65%

to achieve a coherent result. Goals at different scales can not only override one another when necessary, but they can pass messages to influence or direct each others' behavior. 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.

D. Evaluation

We have evaluated our agent against the built-in StarCraft AI. The agent was tested against all three races on three professional gaming maps that encourage different styles of gameplay. The results are shown in Table 1. The agent achieved a win rate of over 60% against all of the races. Additionally, analysis of the agent's replays demonstrates that the agent performed over 200 game actions per minute on average, which shows that the agent was able to combine highly reactive unit micromanagement tasks with high-level strategic reasoning.

VIII. CONCLUSIONS AND FUTURE WORK

Using StarCraft as an application domain, we have implemented an agent in ABL to demonstrate the ability of reactive planning to support authoring of complex intelligent agents. By presenting new idioms, we have shown concretely how a reactive planning language like ABL provides the structure required to support multi-scale game AI.

Our reactive planning agent reasons at multiple scales and across many concurrent goals in order to perform well in the StarCraft domain. Different threads communicate with each other and cooperate to give rise to effective unit control in which complicated multi-unit behaviors are explicit rather than emergent. The explicit property of our higher-level tactics allows complicated strategic reasoning processes to deal directly with these tactics, instead of trying to manipulate the state of individual units to give rise to some desired emergent behavior. Using this unified reasoning architecture, our multi-scale agent is able to play competitively against the built-in StarCraft AI.

We have extensive plans for future work on the agent. We have not yet addressed some of the most interesting challenges about the domain of StarCraft, and our agent performs poorly against even moderately skilled human players. Problems such as spatial reasoning about building placement and knowledge management for hidden-information play have not been addressed in our current implementation, but are high on our list of priorities. By implementing these distinct reasoning processes in ABL, we will be able to easily integrate them within the context of our multi-scale agent. Future work also includes evaluating the performance of EISBot in the AIIDE 2010 StarCraft AI competition.

REFERENCES

[1] D. Isla, "Halo 3-Building a Better Battle," in *Game Developers Conference*, 2008.

[2] A. B. Loyall, "Believable Agents: Building Interactive Personalities," Ph.D. dissertation, Carnegie Mellon University, 1997.

[3] P. Langley and D. Choi, "A Unified Cognitive Architecture for Physical Agents," in *Proceedings of AAAI*. AAAI Press, 2006, pp. 1469–1474.

[4] M. Mateas, "Interactive Drama, Art and Artificial Intelligence," Ph.D. dissertation, Carnegie Mellon University, 2002.

[5] S. Rabin, "Implementing a State Machine Language," in *AI Game Programming Wisdom*, S. Rabin, Ed. Charles River Media, 2002, pp. 314–320.

[6] D. Isla, "Handling Complexity in the Halo 2 AI," in *Game Developers Conference*, 2005.

[7] E. Yiskis, "A Subsumption Architecture for Character-Based Games," in *AI Game Programming Wisdom 2*, S. Rabin, Ed. Charles River Media, 2003, pp. 329–337.

[8] J. Orkin, "Three States and a Plan: The AI of F.E.A.R." in *Game Developers Conference*, 2006.

[9] G. Flórez-Puga, M. Gomez-Martin, B. Diaz-Agudo, and P. Gonzalez-Calero, "Dynamic Expansion of Behaviour Trees," in *Proceedings of Artificial Intelligence and Interactive Digital Entertainment Conference*. AAAI Press, 2008, pp. 36–41.

[10] H. Hoang, S. Lee-Urban, and H. Muñoz-Avila, "Hierarchical Plan Representations for Encoding Strategic Game AI," in *Proceedings of Artificial Intelligence and Interactive Digital Entertainment Conference*. AAAI Press, 2005.

[11] M. Mateas and A. Stern, "Façade: An Experiment in Building a Fully-Realized Interactive Drama," in *Game Developers Conference*, 2003.

[12] J. McCoy and M. Mateas, "An Integrated Agent for Playing Real-Time Strategy Games," in *Proceedings of AAAI*. AAAI Press, 2008, pp. 1313–1318.

[13] J. Laird, "Using a Computer Game to Develop Advanced AI," *Computer*, vol. 34, no. 7, pp. 70–75, 2001.

[14] S. Wintermute, J. Xu, and J. Laird, "SORTS: A Human-Level Approach to Real-Time Strategy AI," in *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference*. AAAI, 2007, pp. 55–60.

[15] D. Choi, T. Konik, N. Nejati, C. Park, and P. Langley, "A Believable Agent for First-Person Shooter Games," in *Proceedings of Artificial Intelligence and Interactive Digital Entertainment Conference*. AAAI Press, 2007, pp. 71–73.

[16] M. Chung, M. Buro, and J. Schaeffer, "Monte Carlo Planning in RTS Games," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*. IEEE Press, 2005, pp. 117–124.

[17] S. Jang, J. Yoon, and S. Cho, "Optimal Strategy Selection of Non-Player Character on Real Time Strategy Game using a Speciated Evolutionary Algorithm," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*. IEEE Press, 2009, pp. 75–79.

[18] C. Miles, J. Quiroz, R. Leigh, and S. Louis, "Co-Evolving Influence Map Tree Based Strategy Game Players," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*. IEEE Press, 2007, pp. 88–95.

[19] D. W. Aha, M. Molineaux, and M. Ponsen, "Learning to Win: Case-Based Plan Selection in a Real-Time Strategy Game," in *Proceedings of the International Conference on Case-Based Reasoning*. Springer, 2005, pp. 5–20.

[20] M. Mateas and A. Stern, "A Behavior Language for Story-Based Believable Agents," *IEEE Intelligent Systems*, vol. 17, no. 4, pp. 39–47, 2002.

[21] B. Weber and M. Mateas, "Conceptual Neighborhoods for Retrieval in Case-Based Reasoning," in *Proceedings of the International Conference on Case-Based Reasoning*. Springer, 2009, pp. 343–357.

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