Game Over: You have been beaten by a GRUE

Elizabeth Gordon and Brian Logan
School of Computer Science and IT
University of Nottingham, UK
esg@cs.nott.ac.uk, bsl@cs.nott.ac.uk

Abstract

We describe GRUE, an architecture for game agents. GRUE facilitates the development of flexible agents capable of balancing competing goals and responding appropriately to their environment. We briefly describe GRUE's representation of the agent's environment and sketch the algorithm used to select between competing goals. We present some preliminary results from an evaluation of GRUE in a Tileworld environment which suggest that the features of GRUE are useful in dynamic environments. We conclude with a brief description of future work.

Introduction

Modern computer games provide dynamic, compelling simulated worlds. However, the characters in these games often exhibit formulaic, unrealistic behaviour. As game developers try to draw in new audiences and tell more sophisticated stories, they will require characters capable of handling complex situations in believable ways.

We believe that developing believable characters for games is easier using a general architecture capable of supporting flexible goal prioritisation (see, e.g., (O'Brien 2002)). A key problem with goal-based architectures is *goal arbitration*, i.e., deciding which goal or goals to work on next. For example, a bot in a first person shooter game may have a goal to defend a teammate under attack (triggered by a request from the teammate), a goal to maintain its health level (innate), and a goal to obtain a better weapon (autonomously generated). Ideally the agent should work to achieve as many goals as possible, while respecting any priority ordering over goals and the limitations imposed by its environment and the actions it can perform. The agent should be able to respond to opportunities and threats as they arise, while continuing to work towards its existing goals (to the extent to which this is possible). However, many existing goal-based agent architectures only allow an agent to work towards a single goal at a time.

In this paper we give a brief overview of GRUE (Gordon & Logan 2004), an architecture for game agents which are capable of balancing competing goals while responding appropriately to their environment. Building on the teleo-reactive programming framework originally developed in robotics, we introduce the notion of a *resource* which represents a condition which must be true for the safe concurrent execution of a durative action. Resources provide a rich representational framework for the kinds of objects that typically occur in game worlds, including time, information, and continuous quantities such as money, magic power, ammunition etc. They allow the game developer to specify which game objects are required to achieve a given goal, which objects are preferred for a goal, and whether game objects can be shared between competing goals. By varying these specifications, the developer can add personality to game characters. We briefly outline the GRUE architecture, and describe a goal arbitration algorithm for teleo-reactive programs with resources which allows a GRUE agent to respond flexibly to multiple competing goals with conflicting resource requirements.

The remainder of this paper is organised as follows. In the next two sections we give a brief description of the teleo-reactive programming framework and highlight some of the problems with implementing goal based game agents. We then introduce the notion of a resource and give an overview of the GRUE architecture before going on to present some preliminary results from an evaluation of GRUE in Tileworld (Pollack & Ringuette 1990). Finally, we briefly discuss some related work, and present our conclusions.

Taking Inspiration from Robotics

Our approach is based on teleo-reactive programs (TRPs) as described in (Benson & Nilsson 1995), which were developed for controlling robots. Each TRP brings about a particular goal, which we refer to as the success condition of the TRP. TRPs consist of a series of rules, each of which consists of some number of conditions and actions. The rules are ordered: rules later in the order have actions that tend to bring about the conditions of rules earlier in the order. The first rule matches the success condition and has a null action. A TRP is run by evaluating the conditions of the rules in order and executing the actions of the first rule whose conditions evaluate to true when matched against a world model stored in the agent's memory. The actions can be durative, in which case the action continues as long as its condition is true. Our agents use TRPs as pre-written plans for achieving goals. As in (Benson & Nilsson 1995; Benson 1996), multiple plans (programs) can be executed simultaneously by an arbitrator. However, our system uses a novel goal arbitration algorithm (described below) rather

than that described in (Benson & Nilsson 1995).

The teleo-reactive architecture described by Benson and Nilsson is not completely autonomous. Goals are proposed by a human user who is also responsible for determining the reward for achieving a goal. During each execution cycle, the arbitrator runs the program with the best reward/time ratio. The reward is the expected reward for achieving the goal (which may or may not actually be received) and the time is the estimated time necessary to reach a point where the program can safely be stopped. This allows the agent to take small amounts of time to achieve less rewarding goals while it is also working on a more time-consuming but more rewarding goal.

Game Agents

Requiring a human user to provide goals is reasonable in some situations, but impractical in most computer games. A common approach in computer games is to create an agent with a fixed set of goals, the priorities of which are set by the programmer and do not change.

We used this approach to build a simple teleo-reactive agent which tries to achieve a set goals specified by the programmer. Programmer-specified priorities replaced reward/time ratios and the Benson & Nilsson goal arbitration algorithm was not used. The agent plays 'Capture the Flag', one of the game types provided by Unreal Tournament. Unreal Tournament (UT) is a 'first person shooter' game in which players compete in a map or level to achieve specific game objectives. In Capture the Flag, each player has a flag and can gain points by stealing their opponent's flag. The game is combat based, with players using weapons to attack each other. Our agent uses a basic strategy of always guarding its own flag. It has goals for regaining health, attacking an opponent, and returning to its flag. Each goal has a corresponding teleo-reactive program, but only one program can run during each execution cycle. The goal for regaining health has the highest priority, which means that if the agent's health is low, only the program for regaining health will run. The problem is that once a health pack has been used it takes some time for another one to appear. While the agent is pursuing the 'regain health' goal (waiting for a health pack to appear), it ignores approaching opponents. This is not effective or realistic behaviour. While it is possible to modify the TRPs or adjust the goal priorities to avoid this particular problem, most fixed goal orderings will have similar types of problems. A more general solution is to allow the agent to towards multiple goals simultaneously, giving greatest priority to those goals which are most important in the current situation.

We have therefore developed a new teleo-reactive architecture, GRUE (Goal and Resource Using architecturE) with the following features:

- it provides support for goal generators, allowing an agent to generate new top-level goals in response to the current game situation and assign priorities to goals based on the current situation; and
- it uses a new arbitration algorithm which allows multiple programs to run actions in parallel during each cycle where this is possible.

Resources

A key idea in GRUE is the notion of a resource. Resources such as game objects and information are represented in terms of properties, allowing the agent to share resources between goals.

Informally, a *resource* is anything necessary for a rule in a plan to run successfully. Resources are stored in the agent's world model and represented as lists of lists, where each sublist contains a label and one or more values associated with that label. Each of these sublists represents a property. All resources have an ID and a TYPE properties and may list additional properties as required by the application. For physical objects, these might include location, colour, or shape. For example, weapons could be represented as the following resources:

```
(Weapon1
  [TYPE Weapon]
  [SUBTYPE MachineGun]
  [CARRIED True])

(Weapon2
  [TYPE Weapon]
  [SUBTYPE SniperRifle]
  [CARRIED True])

(Weapon3
  [TYPE Weapon]
  [SUBTYPE RocketLauncher]
  [CARRIED False])
```

Many objects in real and simulated worlds are divisible. Resources capture this reality by allowing properties to have divisible values, denoted by the DIV keyword. Resources with divisible properties can be split into two or more parts, as required by the agent's current goals. For example,

```
(Gold1 [TYPE money] [Amount DIV 20])
```

represents twenty gold coins. We can use part of this resource to pay for something, and hold onto the rest.

Conditions in GRUE TRPs match against resources in the agent's world model using resource variables. A *resource variable* is a 3-tuple containing an identifier for the variable, a set of required properties, and a set of preferred properties. The required properties list specifies those properties that must be present for a resource to be bound to the variable. If there is more than one resource with all the required properties, the resource matching the largest number of preferred properties will be bound. If two or more resources have all the required properties and the same number of preferred properties, then one resource will be chosen arbitrarily.

This representation allows the creation of agents with flexible behaviour. An agent can prefer a particular item, but substitute any item with the necessary properties for the task at hand. For instance, in a game such as Unreal Tournament, an agent might prefer rocket launchers, as specified by the resource variable:

```
(?Weapon
  ([TYPE Weapon] [CARRIED True])
  ([SUBTYPE RocketLauncher]))
```

If the available resources are those listed above, the agent won't be able to use its preferred weapon, the rocket launcher, because it has a CARRIED value of false. Instead, the resource variable will bind to one of the other weapons (chosen arbitrarily). However, the agent will choose a rocket launcher if it is carrying one.

In addition, we define special notation to allow us to prefer particular values or ranges of values for properties. A range with soft boundaries represents a preference for a value within the range, but a value outside the boundary is also acceptable. A range with firm boundaries specifies that the value must lie within the range. We also allow a utility ordering over the range. For example, we can specify that a property value should be 'between 5 and 10 but closer to 10', 'as small as possible', or 'ideally between 30 and 40'. In a game world, this notation can be used to select nearby items (smallest distance), the weapon with the most ammunition, or spare items to sell (largest number). It could also be used to select locations such as the closest hiding place.

Details of the range notation are beyond the scope of this paper (see (Gordon & Logan 2004)). However, as an example, the following resource variable requests a weapon, preferably a machine gun, as close to the agent as possible and no further than ten units away.

```
(?ClosestWeapon
  ([TYPE Weapon]
   [DISTANCE #| 0 <- 10 |#])
  ([SUBTYPE MachineGun]))
```

Finally, here is an example of a TRP using resources. This TRP causes an agent in a first-person shooter type game to shoot at nearby enemies, turning to face them if necessary. The rules are written using an if ... then ... format, with the word "null" representing the null action.

```
if  not (?enemy
          ([TYPE Enemy] [VISIBLE True]
           [NEARBY True])
          ())
then
   null

if (?enemy
      ([TYPE Enemy] [VISIBLE True]
       [NEARBY True])
      ())
and (?weapon
      ([TYPE Weapon] [CARRIED True])
      ([SUBTYPE MachineGun]))
then
   shoot(?enemy)

if (?enemy
      ([TYPE Enemy] [VISIBLE False]
       [NEARBY TRUE])
      ())
and (?weapon
      ([TYPE Weapon] [CARRIED True])
      ([SUBTYPE MachineGun]))
then
   turn_towards(?enemy)
```

Overview of GRUE

The GRUE architecture contains four main components: a working memory, a set of goal generators, a program selector and the arbitrator (see Figure 1).

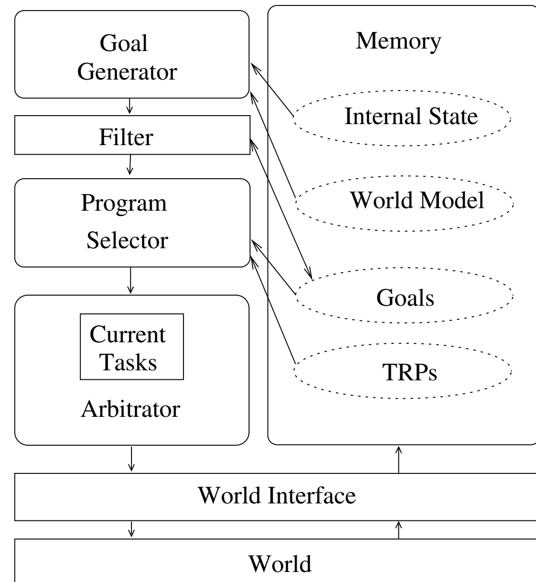

Figure 1: The GRUE Architecture

The system runs in cycles. At each cycle:

- the world interface takes information from the environment, in this case a game engine, does any necessary preprocessing, e.g., computing the distance between game objects and the agent, and stores the results as resources in working memory;
- goal generators are triggered by the presence of particular information in working memory, creating new goals and/or adjusting the priority level of existing goals as the agent's situation changes;
- the filter removes low priority goals;
- the program selector creates a task for each goal by matching the goal against the success condition of a TRP—a task contains the TRP, the priority of the goal, and a flag indicating whether the associated goal is a maintenance goal or an achievement goal; and
- the arbitrator decides which task(s) to run during the current cycle.

The cycle then repeats.

Goal arbitration is a key feature of GRUE and we describe the goal arbitration algorithm in more detail. The arbitrator allocates resources to each task, allows each program to propose one or more actions, checks the list of proposed actions for conflicts and finally executes a consistent subset of the actions. The arbitrator attempts to execute as many high-priority tasks as possible. To do so, it allocates resources to the tasks, starting with the highest priority task. The main criteria used for binding resource variables is that a lower

priority task may never take resources from a higher priority task. Therefore, we allow the highest priority task to bind its resources first. When a GRUE TRP is run, the resource variables are matched against the resources stored in working memory. A resource variable can only be bound to a resource which has all of the properties listed in the required properties list. In the case of a tie, the resource variable will bind to the resource with the largest number of preferred properties. If two or more resources have all the required properties and the same number of preferred properties, then one resource will be chosen arbitrarily. If a resource variable is bound during one execution cycle it will be bound to the same resource at the next cycle unless the resource is no longer available, e.g., if it has been bound by a higher priority task or the game object represented by the resource is no longer available.

A task which is capable of firing a rule in the TRP associated with the task, i.e., where the resource variables in the conditions of a rule can be bound to resources, is said to be runnable. Once the variables have been bound, each runnable task proposes an action. If the actions conflict, for example, if two tasks propose moving in opposite directions, the action proposed by the lower priority task is discarded. GRUE chooses the largest set of actions that can be run concurrently during each cycle.

Evaluation

We have tested our architecture in Tileworld, a commonly used testbed for agent architectures (Pollack & Ringuette 1990). Tileworld consists of an environment containing tiles, holes and obstacles, and an agent whose goal is to score as many points as possible by pushing tiles to fill in the holes. The environment is dynamic: tiles, holes and obstacles appear and disappear at rates controlled by the experimenter. While Tileworld is not a game, it shares the dynamic features of many computer games. We have used Tileworld to allow us to verify that GRUE works as expected before implementing an agent for a more complicated commercial game environment.

GRUE is designed such that all of the main features can be enabled or disabled. We conducted three tests to evaluate the effectiveness of preferred properties, divisible properties and the GRUE architecture as a whole. In each case we compared a 'standard' GRUE agent with all the features enabled with an agent which had the relevant feature(s) disabled, and/or simple transformations applied to the goal generators and TRPs. The agents were compared in two Tileworld environments, 'hard' and 'easy. The hard environmental case is a sparse, rapidly changing environment, while the easy case is a dense, slowly changing environment. The data presented is an average of 50 scores in each of the two environments.

To test the advantages of preferred properties, we investigated two methods of eliminating preferred properties from the programs. The first method simply deletes all preferred properties from each resource variable appearing in a rule condition. The second method moves all preferred properties into the required properties list.

As can be seen in Figure 2, the standard GRUE agent outperforms both the agent with preferred properties deleted and the agent with 'preferred' properties required in both hard and easy environments.

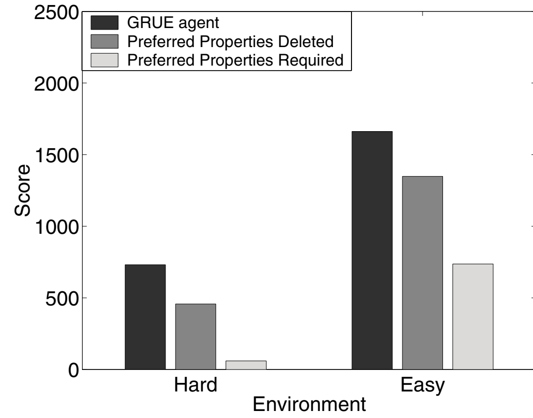

Figure 2: Performance of a GRUE agent compared to agents without preferred properties

Next we compared the standard agent against an agent without divisible properties or value ranges (see Figure 3). As can be seen, the standard agent performs better than the agent that does not use divisible properties or ranges. The difference is particularly marked in the easy environment, where the score of the standard agent is over 2.5 times greater than that of the test agent.

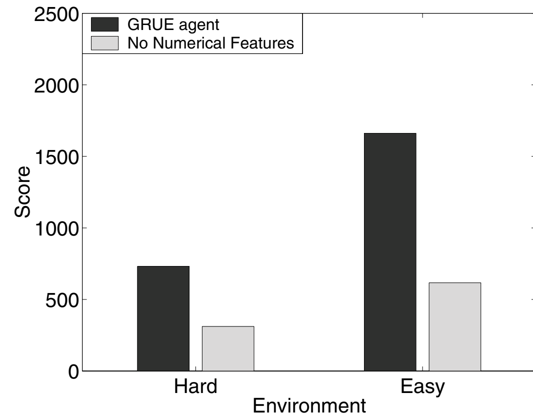

Figure 3: Performance of a GRUE agent and an agent without numerical ranges or divisible properties

Finally, we evaluated the effectiveness of the GRUE architecture as a whole. From Figure 4 we can see that the GRUE agent (with the goal generators, filter and arbitrator enabled) outperforms a similar agent using a static set of goals with constant priorities set by the programmer.

Although preliminary, our data supports our hypothesis that the features provided in GRUE are useful for agents in dynamic environments. An agent with all of the features enabled performs better (on average) than agents with one or more features disabled. In the Tileworld agent, our agent shows the largest decrease in performance when the numer-

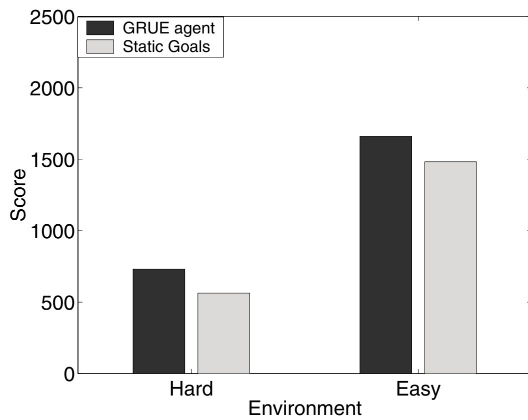

Figure 4: Performance of a GRUE agent and an agent with static goals

ical features are disabled. We can also see that if preferred properties are not used, the agent will perform better if it is designed to require only those properties that are absolutely necessary rather than requiring all properties that might be useful.

We have also implemented a GRUE agent for Unreal Tournament. We are currently improving the set of TRPs for the Unreal Tournament agent and collecting data in this environment.

Related Work

Computer games and other simulated environments have many attractions for AI researchers and there has been a considerable amount of work in this area.

Our approach builds on that of (Nilsson 1994; Benson & Nilsson 1995). GRUE differs from (Benson & Nilsson 1995; Benson 1996) in that the GRUE arbitrator does not use rewards or time estimates. Instead, goals in GRUE are given a priority value by the goal generator. While it would be possible to use reward/time ratios to compute goal priorities, we believe that simple priority values determined by the current context result in more believable behaviour. A disadvantage of using a reward/time ratio with fixed rewards is that unless the reward values are carefully chosen (or context dependent), the programmer cannot force the agent to work exclusively on a high priority goal. If a less important goal can be completed in a sufficiently short time, the arbitration scheme proposed by Benson and Nilsson will always take the time to complete it. By contrast, GRUE's arbitrator will pursue multiple goals simultaneously where possible and otherwise focus on the highest priority goal.

The architecture which is probably most similar to ours is Wright's MINDER1 (Wright 1997). MINDER1 uses a library of teleo-reactive programs and generates and manages motives, which seem to be the functional equivalent of goals. Each motive contains a condition to be made true, an insistence value which represents the importance of the goal, and a flag indicating whether or not the motive has passed through an attention filter. The filter uses a simple threshold function, allowing motives through when their insistence is above the threshold. MINDER1 is based on a three-layer architecture developed as part of the Cognition & Affect project by Sloman and Beaudoin (Wright, Sloman, & Beaudoin 1996). As such, it includes both a management layer and a meta-management layer. The management layer can suspend tasks, schedule tasks, and expand a motive into a plan. The meta-management layer is responsible for monitoring the management layer and making adjustments as necessary. However, MINDER1 can only execute one plan at a time. GRUE's ability to work toward several goals at once, choosing actions based on the relative importance of the goals and the availability of resources gives GRUE additional flexibility.

The use of commercial computer games as a platform for AI research goes as far back as 1987, with the creation of an agent for the game Pengo (Agre & Chapman 1987). Agre & Chapman's approach is to use simple rules, which, when combined with a complex environment, allow the appearance of intelligent behaviour in their agent. These rules are quite similar to TRPs in that they allow the agent to automatically take advantage of new opportunities and react to obstacles. Pengi's rules match against agents and objects in terms of their use rather than using specific identifiers. This is rather similar to our own approach of representing resources in terms of properties. However, in contrast to GRUE, Pengi does not represent or reason about goals, and has no state.

Much work on game agents has been done using the Soar architecture (e.g., (Laird & Duchi 2000; Laird 2000; van Lent *et al.* 1999)). The Soar system is designed to reason about goals, but it is limited to a single goal hierarchy. Several approaches have been taken to the use of multiple goal hierarchies in Soar, however they require either representing some goals implicitly or forcing unrelated goals into a single hierarchy (Jones *et al.* 1994). In contrast, GRUE has been designed to process parallel goals.

DePristo and Zubek (DePristo & Zubek 2001) describe a hybrid architecture used for an agent in a MUD (Multi-User Dungeon). This type of environment is essentially a role-playing game and typically involves tasks like killing monsters and buying equipment such as weapons and armor. The architecture included a deliberative truth maintenance and reasoning component along with a reactive layer capable of handling urgent situations without input from the deliberative layer. The system was capable of surviving in the MUD environment, but several problems were encountered. The system had difficulty representing quantities like amounts of gold, and there were some problems handling goals. In contrast to DePristo and Zubek's architecture, GRUE has been designed to process goals and we have designed our data structures specifically to handle the types of information and tasks that are commonly encountered in game environments.

Bryson's work (Bryson 2001) is closely related to teleo-reactive programs. Bryson describes an approach to building behaviour-based agents called Behavior-Oriented Design. She discusses both a development process and a modular architecture which includes Basic Reactive Plans as one of the core components. Basic Reactive Plans are essentially

identical to teleo-reactive programs. She has used BOD systems for a number of applications, including a robot control system, modelling primate behaviour, and characters in an interactive virtual world. One significant difference between Bryson's system and ours is that her agents are controlled by drive collections. A drive collection is just a Basic Reactive Plan which contains a list of tasks the agent might want to do. Using this representation, tasks always have the same relative importance. GRUE's ability to change the relative priorities of goals at any time allows greater flexibility in decision making.

Conclusions

We have argued that game environments can be effectively handled by an agent architecture which uses an explicit representation of goals. Taking inspiration from robotics, we have proposed an approach based on teleo-reactive programs. TRPs allow goal-directed behaviour and handle changes in the environment gracefully. Existing teleo-reactive architectures (e.g., (Benson & Nilsson 1995; Benson 1996)) have a number of drawbacks from a game point of view. We have therefore developed a new architecture, GRUE, which overcomes these limitations. GRUE agents are capable of generating their own goals and can execute multiple actions per cycle to achieve several goals simultaneously. A key idea in GRUE is the notion of resources which represent game objects and information. Resource properties allow the flexible allocation of game objects to tasks and the sharing of resources between tasks depending on task priority and the agent's preferences. In addition, GRUE provides support for handling numerical property values which are common in game worlds. We have evaluated the effectiveness of our architecture, and shown that, in a dynamic environment, a GRUE agent performs better than agents without preferred properties, agents without special facilities for handling numerical values, and agents with a static set of constant priority goals.

We are currently developing a GRUE agent for Unreal Tournament. However, we believe the GRUE architecture can be applied to a wide range of game types. GRUE also promotes code reuse both within and across games. Teleo-reactive programs and their associated goal generators can be reused in similar games or for multiple characters in the same game. By careful use of preferred properties we can generate different behaviour from the same basic program.

Acknowledgements

This work is supported by Sony Computer Entertainment Europe.

References

Agre, P. E., and Chapman, D. 1987. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence, AAAI-87*, 268–272.

Benson, S., and Nilsson, N. 1995. Reacting, planning and learning in an autonomous agent. In Furukawa, K.; Michie, D.; and Muggleton, S., eds., *Machine Intelligence*, volume 14. The Clarendon Press.

Benson, S. 1996. *Learning Action Models for Reactive Autonomous Agents*. Ph.D. Dissertation, Stanford University.

Bryson, J. J. 2001. *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents*. Ph.D. Dissertation, MIT, Department of EECS, Cambridge, MA. AI Technical Report 2001-003.

DePristo, M., and Zubek, R. 2001. being-in-the-world. In *Proceedings of the 2001 AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*.

Gordon, E., and Logan, B. 2004. Managing goals and real-world objects in dynamic environments. In Davis, D., ed., *Visions of Mind: Architectures for Cognition and Affect*. Idea Group, Inc. Forthcoming.

Jones, R.; Laird, J.; Tambe, M.; and Rosenbloom, P. 1994. Generating behavior in response to interacting goals. In *Proceedings of the 4th Conference on Computer Generated Forces and Behavioral Representation*.

Laird, J., and Duchi, J. 2000. Creating human-like sythetic characters with multiple skill levels: A case study using the Soar Quakebot. In *AAAI Fall Symposium Series: Simulating Human Agents*.

Laird, J. 2000. It knows what you're going to do: Adding anticipation to a quakebot. In *AAAI 2000 Spring Symposium Series: Artificial Intelligence and Interactive Entertainment, March 2000, Technical Report SS-00-02*.

Nilsson, N. 1994. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research* 1:139–158.

O'Brien, J. 2002. A flexible goal-based planning architecture. In Rabin, S., ed., *AI Game Programming Wisdom*. Charles River Media. 375–383.

Pollack, M., and Ringuette, M. 1990. Introducing the Tileworld: Experimentally evaluating agent architectures. In Dietterich, T., and Swartout, W., eds., *Proceedings of the Eighth National Conference on Artificial Intelligence*, 183–189. Menlo Park, CA: AAAI Press.

van Lent, M.; Laird, J.; Buckman, J.; Hartford, J.; Houchard, S.; Steinkraus, K.; and Tedrake, R. 1999. Intelligent agents in computer games. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, 929–930.

Wright, I.; Sloman, A.; and Beaudoin, L. 1996. Towards a design-based analysis of emotional episodes. *Philosophy Psychiatry and Psychology* 3(2):101–137.

Wright, I. 1997. *Emotional Agents*. Ph.D. Dissertation, University of Birmingham.