SORTS: A Human-Level Approach to Real-Time Strategy AI

Sam Wintermute, Joseph Xu, and John E. Laird

University of Michigan
2260 Hayward St.
Ann Arbor, MI 48109-2121
{swinterm, jzxu, laird}@umich.edu

Abstract

We developed knowledge-rich agents to play real-time strategy games by interfacing the ORTS game engine to the Soar cognitive architecture. The middleware we developed supports grouping, attention, coordinated path finding, and FSM control of low-level unit behaviors. The middleware attempts to provide information humans use to reason about RTS games, and facilitates creating agent behaviors in Soar. Agents implemented with this system won two out of three categories in the AIIDE 2006 ORTS competition.

Introduction

The goal of our research is to understand and create human-level intelligent systems. Our strategy for achieving that goal is to develop AI systems in a variety of complex environments that make differing demands on the underlying cognitive architecture. Computer games provide rich and varied environments in which we can pursue that goal [Laird & van Lent 2001].

A variety of agents have been developed in Soar [Lehman et al. 1998] for first-person shooter (FPS) games including Descent 3, Quake 2 [Laird 2001], Unreal Tournament [Magerko et al. 2004], and Quake 3. These agents controlled a single embodied entity, emphasizing tactics over strategy, and explored the capabilities required for human-level behavior from a first-person perspective.

Real-Time Strategy (RTS) games make very different demands on the AI than FPS games, both in terms of the reasoning strategies and knowledge that must be encoded to win, but also in terms of basic perceptual and cognitive capabilities. RTS games are distinguished by the following characteristics:

1. A dynamic, real-time environment. In an RTS, a player must respond quickly to environmental changes.
2. Regularities at multiple levels of abstraction. Just as militaries organize soldiers and armaments into squads, platoons, battalions, and regiments, and strategize over these units of varying granularity, RTS games exhibit salient strategic patterns at many different levels.
3. Multiple, simultaneous, and interacting goals. RTS games require players to manage their army's resources and production capabilities simultaneously with engaging in battles or defending bases.
4. Knowledge richness. Players control a wide variety of combative and support units that have distinctive performance characteristics.
5. Large amounts of perceptual data. Each player can control hundreds of units at once, and data about each unit is simultaneously available to the player.
6. The dominance of spatial reasoning. A player must reason about space to explore the map, defend its home base, and organize its troops during an attack.

To explore the interplay of these requirements and intelligent systems, we developed an RTS agent in Soar to play ORTS [Buro & Furtak 2003]. This involved interfacing Soar to the ORTS game engine, developing middleware to support appropriate abstraction of perception and action, and developing agents in Soar. Our system is called SORTS, for Soar/ORTS. We entered our agents in the AIIDE 2006 ORTS competition, winning two of the three categories.

System Description

The organization of SORTS is shown in Figure 1. ORTS, the game engine, is on the right, and Soar, our AI engine, is on the left. In the upper middle is perception, which includes grouping and attention processing controlled by Soar. Motor commands initiated by Soar control finite-state machines that perform primitive actions in ORTS.

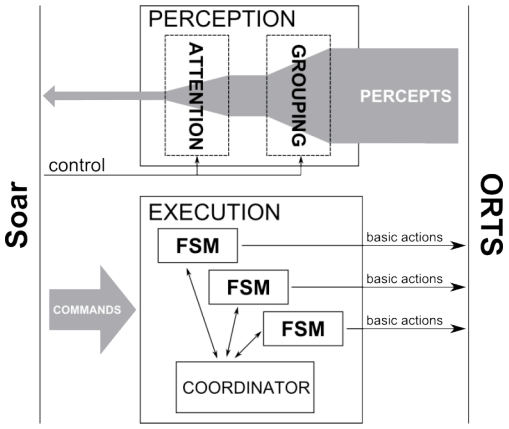

Figure 1. Overview of the SORTS architecture.

ORTS

ORTS is an open source RTS game engine being developed at University of Alberta. ORTS is designed from the ground up for use in AI research. Among the advantages of using ORTS are an extensive, low-level C++ API, a server-client approach that allows for secure competitions over the internet, and easily modifiable game mechanics specified by scripts, allowing the ORTS engine to emulate many commercial RTS games.

The design of ORTS presents some challenges for AI development. First, it takes a minimalist approach to the game engine and only simulates game physics, leaving functionality such as complex path finding and default unit behaviors to the user program. Second, the ORTS API gives low-level and thus voluminous perceptual data to the AI: after each tick of the game clock (typically 8 times per second), the state of every changed game object is sent.

Soar

Our RTS AI is implemented in Soar, an AI architecture that encodes procedural long-term knowledge as production rules and represents the current situation in a declarative working memory, which includes perceptual and internally derived data. Soar does not select a single rule to fire, but instead fires all matching rules in parallel. Soar organizes behavior in terms of decision cycles where it first elaborates the current situation (using rules), noticing patterns in the input and deriving task-relevant structures such as "the worker sent to explore has finished". Additional rules then test the situation and propose alternative actions (called operators). Some operators involve motor actions in the environment (such as building a new structure, or assigning a unit to attack an enemy), while others modify internal data structures, such as storing the fact that a worker has been commanded to build a barracks. If an operator cannot be directly executed, it becomes a goal, which is recursively decomposed into simpler operators, leading to a stack of goals. Soar can select and apply only a single operator at a time (although a single operator can initiate multiple actions) and can have only a single stack of goals, restricting Soar to following a "single train of thought." This has a significant impact on our approach to implementing an RTS AI in Soar.

Soar has two other characteristics that influenced our design. First, moving large amounts of data into Soar from perception is computationally expensive. Soar does not have built-in capabilities for visual abstraction or filtering. Second, Soar is primarily a symbolic reasoning system, and is not designed to process complex numeric calculations, especially vector and matrix operations – not unlike human post-perception capabilities.

Key Issues Addressed by the Interface

Our previous experience with modeling human military pilots [Jones et al. 1999] taught us that achieving human-level performance starts with the interface between the environment and the AI system and the middleware that supports that interface (the center of Figure 1). The interface must allow the agent to receive the same types of information experienced by a human, not as pixels, but in terms of the abstractions that humans have post-perception. For example, humans can sense groups of units, and must focus attention on a subset of the perceptual stream.

Similarly, our AI should control units under the same constraints as a human player who can issue only one command to a unit or group of units at a time. Thus, the middleware must provide low-level control of units (path planning, low-level combat), while the Soar agent provides higher-level control (go to this location, fight this enemy).

Perceptual System

The purpose of the perceptual system is to take game state data received from the ORTS server and create appropriate structures in Soar's working memory. Game state information is provided by ORTS for each individual object, each game frame. In a typical RTS game, there can be hundreds of objects changing each game frame, and those objects will each have numerous properties that could be updated. To avoid an avalanche of perceptual data and to provide Soar with information similar to what a human uses, our middleware supports two operations on the game state information: grouping, which summarizes the information about individual objects; and attention, which excludes unnecessary information. Both of these decrease the amount of incoming data, with grouping providing a key abstraction for tactical reasoning. These capabilities should eventually be addressed by the perceptual system of the cognitive architecture, but is beyond the scope of what is implemented within Soar.

Grouping. The ability of humans to see sets of similar objects as unitary wholes, called Gestalt grouping [Kubovy et. al. 1998], has been well studied by psychologists. The principles of Gestalt grouping specify that if objects are spatially close, and have common features such as shape, color, and motion, they can be perceived as a group. The observer has some top-down control and can choose to see individuals or groups. We model this in our system, enabling it to perceive units and objects grouped by type, owner, and proximity. By default, groups are formed based on all three – the grouping rule associate units of the same type and owner that are within a specified grouping radius, which the agent can change by issuing a command to the middleware. By adjusting the grouping radius to 0, the agent will perceive every unit individually. Figure 2 shows an example of object grouping – there are seven workers, five minerals, and a building, which result in three worker groups, two mineral groups, and a building group.

For each group, the properties of the individual units, such as health and weapon damage, are summarized and attributed to the group. This is the information sent to Soar. Information about individuals is sent only if there is a single individual in a group.

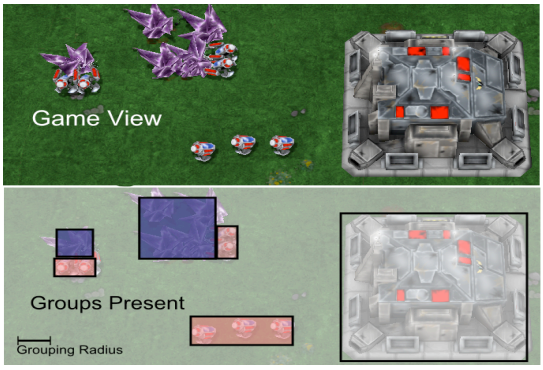

Figure 2. Grouping of objects by type and proximity.

An agent can decrease the amount of perceptual information by increasing the grouping radius, while conversely it can increase the level of detail by decreasing the grouping radius. Hence, perceptual information is never completely unavailable, it is only slower to obtain. Grouping also provides a mechanism for adjusting the level of reasoning. If an agent wants to micromanage, it can set the grouping radius to 0 and reason about individual units, whereas if it wants to assess the overall distribution of forces on the map, it can set a high grouping radius where certain patterns are easier to recognize than if the agent is perceiving individual units. Another benefit is that the agent can use the same rules to reason about situations that differ only in level of detail. For example, a set of rules that can recognize an opponent's flanking maneuver can be applied at the level of single units or multiple armies simply by modifying the grouping radius.

Attention. Even with grouping of objects, the amount of information can be excessive. Since meaningful tactical events in an RTS game tend to occur in restricted spatial areas, it is worthwhile to concentrate perception on a small area while limiting information about the rest of the scene.

The human visual system provides some inspiration for this process. A "zoom lens" metaphor is often used to describe human visual attention [Erikson & St. James 1986, Hill 1999] – some small area of the field of vision is attended to, providing detailed information, while surrounding areas present less information. The area of attention can shift in a guided manner – a human can easily jump to a single red object in a sea of black, for example. Feature Integration Theory (FIT) is a common model for describing this kind of "pop-out" effect [Anderson 2004]. The basic concept of FIT is that unattended objects are available only as features (like colors or shapes) but the features are not integrated together into individual objects. In the example of a red object in a field of black, there is information that something red exists, but the remaining features of the object (it is a rectangle, for example) can be integrated only when attention selects it. Attention can select the red object without search if it is the only red object, but if there are many red objects, any particular one must be found by searching all the red objects, focusing attention on each individually.

To achieve a similar affect, we overlay a resizable rectangular viewfield on the game map, so that all visual information outside this field is ignored. The attentional "zoom lens" is implemented as a moveable point, or focus, in the viewfield, around which a fixed number of the closest groups are perceived in full detail. These are the *attended* groups. The viewfield is evenly divided into nine sectors in a grid layout. The features of unattended groups are then coalesced into a *feature map* within each of the grid sectors, so that there is a count of the number groups with each features in each sector. Example features include enemy units, worker units, or units with low health. The feature maps allow Soar to switch its focus quickly to an unattended group that has a specific feature of interest. This scheme, in addition to supporting fast searches, limits the size of the perceptual input, since unattended groups are only reflected in the feature counts.

Figure 3 illustrates the attention system. The four groups closest to the focus point are attended and all information about them is presented to Soar. The feature information about groups in each of the sectors outside attention (in this case, their friendliness) is summarized. The agent can change the focus by selecting one of the unattended objects by a feature in a sector, for example, attending to the enemy group in the upper-left sector.

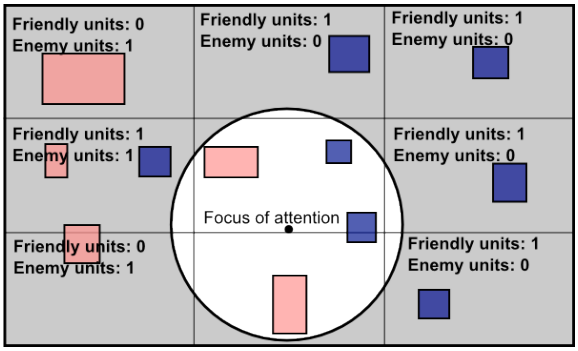

Figure 3. Filtering of objects through attention.

There could be objects outside of these nine sectors, if the agent has restricted its viewfield to a region smaller than the entire world. Since the feature maps have a fixed number of sectors, restricting the field of vision serves to increase their resolution. A situation where this is useful is the identification of an enemy unit in an unusual place. If the entire map is always in the field of view, a small area of it being attended to at once, the data in the feature maps will be very low resolution. If the enemy is in many places, it is likely at least one enemy unit will be somewhere in each sector. If one of those sectors also contains a region the agent is trying to control, there is no way of quickly knowing if an enemy is inside or outside the region without attending to it. However, if the agent restricts its field of view to the region in question, an enemy present in the feature map will "pop out" and must be an invader.

Execution System

ORTS is a minimalist RTS game engine. The built-in unit commands include moving in straight lines and executing behaviors such as mining or attacking, but not much else. For example, if a unit is ordered to attack a target that is out of its firing range, the unit will not automatically move within range of the target before executing the attack.

Most commercial RTS games do not require human players to command units at such a low level. Instead, they provide higher-level commands such as attack-move and harvest looping. Units also typically have default behaviors such as attacking nearby targets or running away when unarmed and under attack. While humans are capable of commanding units to carry out each of these operations manually, requiring them to do so would be overwhelming. In order to provide Soar with a human-level interface, we created middleware to support default unit behaviors via finite state machines and global coordinators.

Finite state machines. The execution system accepts commands for groups from Soar and translates them into atomic actions which are sent to the ORTS server. For non-atomic Soar commands, the execution system assigns a finite state machine (FSM). The FSM provides the control for the detailed execution of the individual units that make up the group. FSMs persist until either the specified behaviors are completed or the command is cancelled. After each game frame, every active FSM is updated. FSMs are given access to all percepts and can be arbitrarily complex. However, in adhering to the policy of emulating commercial RTS games, we have implemented only common high-level commands and default behaviors such as attack-move, attack nearby targets, and harvest loop. FSMs are not tied to the Soar decision cycle in any way. Once an FSM has been assigned to a unit, the FSM does not wait for any other output from Soar. This allows units to act according to their instructions even when Soar is attending to other tasks.

Global coordinators. In commercial RTS games, units not only exhibit a certain level of autonomy in executing their own commands, but often can cooperate in jointly carrying out a behavior assigned to a group. For example, when a group of marines are ordered to attack an enemy force in Starcraft, the computer will automatically distribute the rifle fire of the group evenly over the enemy units. This kind of behavior is not possible with FSMs unless each FSM knows about the presence and state of the other FSMs they are cooperating with. To achieve this kind of group behavior, we created global coordinators that direct what each FSM in a group should do, rather than having them negotiate a multi-agent policy without executive control.

We have implemented two coordinators: one for managing resource mining and one for managing attacks. The mine manager attempts to optimize the assignment of worker units to resource patches in order to maximize the rate of resource income. Simple learning is used here—the manager keeps track of the actual performance of each worker, and reassigns workers from poorly performing routes to potentially better routes. The attack manager attempts to implement the strategy of focusing the entire group's fire on one enemy unit at a time, in order of decreasing threat. The attack manager also handles the movement and positioning of units during the attack.

We also extended the native ORTS pathfinder by adding heuristics for cooperative pathfinding. All commands involving movement use the pathfinder. With these capabilities, the middleware takes much more processing time than Soar. Activities such as pathfinding and attack coordination are computationally expensive, and the middleware must manage each individual unit.

Multi-tasking

RTS games are typified by having multiple, interacting tasks. These tasks are hierarchical in nature, such as a frontal attack decomposing into producing sufficient units and mounting the attack, tasks which can be further decomposed themselves. When a player must handle multiple tasks, it is useful to switch to other tasks when the current task is waiting on some event. There are also situations in which the player must respond to some unexpected event and interrupt the current task for another more urgent task, such as defending his home base.

In SORTS, tasks are the actions that can be unified under a single goal. Tasks map well onto Soar's subgoals:

1. Soar's subgoaling mechanism is hierarchical, so we can easily define a hierarchy of tasks.
2. Since Soar can only select operators on the lowest subgoal, while the subgoal is present Soar can only send commands to ORTS concerning that subgoal.
3. A subgoal is retracted if a more important subgoal's conditions are met, allowing for task preemption. However, subgoals of equal importance will not interleave.
4. Soar signals when there is no activity for a current task, making it easy to implement task switching to another task that is ready for execution.

As an example, consider a Soar agent that has two tasks, building up its base and launching an enemy attack. The agent will first send its troops toward the enemy base, and will have to wait for them to get there. Since it has nothing to do for a while, the agent will switch to the building task and perhaps construct another building. However, as soon as its troops engage the enemy, the attack task will have higher priority than the building task and the agent will preempt the building task. The agent will not switch back to the building task until the attack task is finished.

Note that this behavior is more comparable to human task-switching behavior than existing AIs. Human players tend to stick to a single task until it is finished or there is a clear hiatus because task switching incurs a high overhead. However, most existing RTS AIs do not suffer from this overhead and thus interleave task execution as much as possible. This is one of the main artificial advantages

existing AIs have over human players, and also one of the main complaints human players have concerning AIs. Our system is naturally constrained by Soar to not take unfair advantage of this discrepancy.

Implemented Agents

We developed agents for the RTS game AI competition at AIIDE 2006. This competition consisted of three separate categories of games: 1. a mining competition, 2. a tank battle, and 3. a complete (but limited) RTS game. We wrote three agents, one for each game, but kept the middleware (and Soar) constant over all three games. The complexities in creating agents for all games meant that there was limited time to test and debug all behaviors. Thus, for this first competition, software bugs played a significant role in the results.

The process of writing all three agents took about two weeks, much of that time dedicated to fixing bugs in the middleware. There was minimal sharing of Soar code between the three agents, as the games presented very different tactical situations. Note that we are not making any claims about how natural it is to program RTS agents in Soar. Instead we found that some aspects of RTS AIs were better suited for production systems such as Soar, while other aspects were easier implemented in sequential programming languages.

Game 1

In this game, the task is to gather as many mineral resources as possible using multiple workers in a fixed amount of time. This is difficult because gathering resources involves many units moving in a small area. Planning paths for many units is difficult, and alternate control strategies must be used if the paths cannot be made collision-free. Even if a perfect cooperative pathfinding system were available, it is difficult to derive the optimal assignment of workers to resources – there are difficult tradeoffs between workers waiting for one another, sharing the same resource location, and sending them to alternate locations that may require more travel time.

In SORTS, resource gathering is handled by a mining coordinator and FSMs in the middleware. Soar only assigns units to the task. Thus, this game mainly tested the middleware components. The pathfinder has simple heuristics to assist cooperative path planning, but does not guarantee collision-free paths. To remedy this, and to avoid dynamic obstacles in the game (sheep), the movement FSM incorporates reactive rules to avoid local collisions. Route assignment was well-handled by the simple learning mechanism in the mining coordinator. Overall, these systems worked well and we won the competition. The second and third place entries gathered 78% as much as our agent and the fourth place gathered 38%.

Game 2

In this game, each player starts with 5 bases and 50 tanks. The goal is to destroy the opponent bases while preserving one's own bases. We used a variety of heuristic strategies that included gathering tanks together and attacking the enemy with a large group, attacking tanks that are firing at one's own bases first, and retreating and regrouping when one's forces become scattered.

Even though Soar could have handled viewing all individual units at once, the ability to group tanks significantly simplified the reasoning processes, especially when evaluating force distributions. However, the inability to perform all but the simplest spatial reasoning made it difficult to implement sophisticated tactics. Unfortunately, a few bugs in perception processing resulted in our entry freezing in many cases, so we lost this competition.

Game 3

The "real RTS" in game 3 started each player with a control center building and a few worker units, requiring the player to build up an army by gathering resources and constructing buildings. Scoring is through a formula incorporating resource and unit production weighted with gains and losses in battle. A successful agent in this game must include many of the capabilities we have discussed – coordinating the behavior of many units both on a small scale and towards a common plan, while remaining responsive to outside factors.

A Soar agent was programmed using task switching and subgoaling as discussed above. This game, while complicated, is still much smaller than the scale of game our system is designed for, which would include many types of units and multiple opponents. The agent we used did not reflect the full range of our system's abilities.

The agent followed a simple overall plan: build a barracks (which can produce marine units) with one worker, and mine minerals with the rest. Produce a few more workers, sending them to mine, in order to generate new resources as fast as possible. Spend the remaining resources on producing marines. Once ten marines are available, send a few out to locate the enemy. As soon as the enemy is sighted, send marines in groups of five to attack it.

The agent did not rigidly follow its plan. If the base was attacked, the mining goal was overridden, and miners were diverted to defend the base, unless sufficient marines were available. If an exploring marine came across an enemy unit, it might be destroyed without discovering the enemy base, necessitating a repeat of the exploration process.

A typical execution sequence might see the agent executing the exploration subtask, adjusting the grouping radius to see individuals, and sending out a marine to explore. Then, there being no more relevant operators in the exploration subtask, the task switching system might change the current task to miner assignment. Upon executing this task, the agent might adjust the grouping

radius to see large groups, and look for a group of unoccupied workers. This entire group could then be quickly assigned to mine minerals, resulting in no more relevant operators in the miner assignment task, triggering another task switch, etc.

For game 3, the only other entry was from University of Alberta. Our agent won 60% of 400 games. The Alberta agent focused on resource collection and defense. As this was the most comprehensive game in the first ORTS competition, bugs had a significant impact on both agents' behavior. In code as complex as these agents, bugs do not always lead to crashes and they are not encountered on every run. Our bug was in our perceptual system (since fixed) and affected ~40% of the games. In ~50% of the games, one or both agents had buggy behavior. When our agent's behavior was bug-free, we found the enemy and attacked. Either we would destroy them (if they encountered a bug) or there would be a battle (which we won 60%). If our agent encountered a bug, we would not find the enemy and the game became a production race, which we won about half of the time.

Discussion

As a Soar research environment, SORTS has shown itself to be very useful. Much of the work described in this paper addresses general problems for cognitive architectures, and is useful outside of the RTS domain.

We are working on extensions to further investigate intelligent systems in domains like ORTS. First, we have experimented with using Soar's reinforcement learning mechanism [Nason & Laird, 2004] with SORTS. While we successfully demonstrated learning of simple policies on restricted aspects of the game, our results suggest that it will be difficult to use reinforcement learning more generally for RTS games until a method is developed for extracting useful features to learn over. This is a general problem in applying reinforcement learning to complex problems and is not specific to Soar.

The RTS domain is particularly suitable for spatial reasoning research, due to the large number of dynamic objects in the world, along with the diagrammatic view. In SORTS, we originally used simple problem specific methods in the middleware. These methods will not scale up to the capabilities required by complete RTS AI agents. In response, we have developed a comprehensive, general spatial reasoning system for Soar, which we will use in future versions of SORTS.

Conclusions

The design of the interface between Soar and ORTS has been driven by two forces: a commitment to humanlike game play and reasoning, and a need to resolve conflicting practical constraints presented by the two systems. Fortunately, these forces were not at odds. Humans face the same kinds of interface problems and the mechanisms they employ solve these problems well. Taking advantage of these human-inspired mechanisms has resulted in a system that is different from conventional approaches to RTS AI, but is still very competent at playing the game. This points the way for more human-like behavior in RTS AI, which has the potential of greatly enhancing the single-player experience so that playing against the computer is more and more like playing against a human opponent.

In conclusion, RTS games are a useful arena for AI research. We have encountered challenging research problems such as grouping, attention, hierarchical control, and spatial reasoning, while integrating them within a cognitive architecture. Future research includes integrating learning and spatial reasoning, in addition to developing knowledge-rich agents for complete games.

References

Anderson, J. R. Cognitive Psychology and its Implications. Worth Publishers, New York, 2004.

Buro, M., Furtak, T. RTS Games as Test-Bed for Real-Time Research, Invited Paper at the Workshop on Game AI, JCIS 2003.

Erikson, C.W., St. James, J.D. Visual Attention Within and Around the Field of Focal Attention: A Zoom Lens Model. Perception & Psychophysics, 40, 225-240, 1986.

Hill, R. Modeling Perceptual Attention in Virtual Humans. Proc. of the 8th Conference on Computer Generated Forces and Behavioral Representation, 1999.

Jones, R. M., Laird, J. E., Nielsen P. E., Coulter, K., Kenny, P., Koss, F. Automated Intelligent Pilots for Combat Flight Simulation, AI Magazine, 20(1), 27-42, 1999.

Kubovy, M., Holcombe, A. O., Wagemans, J., On the Lawfulness of Grouping by Proximity. Cognitive Psychology, 35, 71-98, 1998.

Laird, J. E., It Knows What You're Going To Do: Adding Anticipation to a Quakebot. Agents 2001, Montreal, CA, 385-392, 2001.

Laird, J. E., van Lent, M. Interactive Computer Games: Human-level AI's Killer Application. AI Magazine, 22(2), 15-25, 2001.

Lehman, J. F., Laird, J. E., Rosenbloom, P. S., A Gentle Introduction to Soar, an Architecture for Human Cognition, in Invitation to Cognitive Science, Vol. 4, S. Sternberg, D. Scarborough, eds., MIT Press, 1998.

Magerko, B., Laird, J. E., Assanie, M., Kerfoot, A., Stokes, D. AI Characters, Directors for Interactive Computer Games, Innovative Applications of Artificial Intelligence, 2004.

Nason, S., Laird, J. E., Soar-RL: Integrating Reinforcement Learning with Soar, Cognitive Systems, 6(1), 51-59, 2004.