# 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.

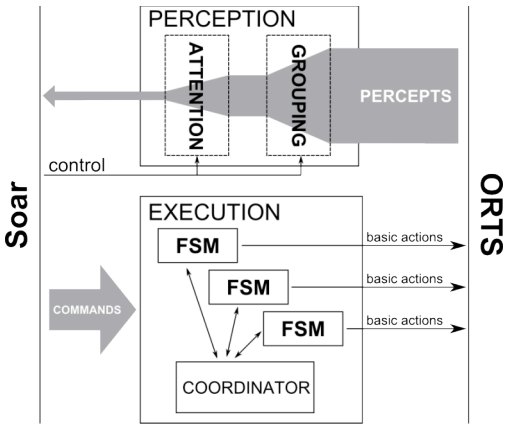


Figure 1. Overview of the SORTS architecture.

## 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.