Soar-SC: A Platform for AI Research in StarCraft: Brood War

Alex Turner

Abstract

Real-Time Strategy (RTS) games are a challenge for artificial intelligences (AI). They are a challenge because of the complexity of spatial reasoning, processing perceptual information, and interpreting imperfect and inaccurate information. RTS games have scripted AIs to deal with these challenges. A scripted AI relies on predetermined behavior to perform actions and decisions in an environment. In this paper, we describe a new platform, Soar-SC, which can be used to create non-scripted AIs for StarCraft. Soar-SC does not cheat and is limited by almost exactly the same constraints as a human player would be. Soar-SC will facilitate research into these challenges using the Soar cognitive architecture.

Introduction

Real-Time Strategy games have been popular among human players for many years. In most RTS games, players can play against other human players or against built-in AIs. Early examples of RTS games include Bungie's Myth, Cavedog Entertainment's Total Annihilation, and Blizzard's StarCraft. StarCraft is an example of a RTS game that has continued to be popular since its release in 1998. In late 1998, Blizzard released an expansion pack called "Brood War" which expanded StarCraft by including more units such as the Terran Medic and maps like Ice maps to play on. StarCraft: Brood War has continuously updated by Blizzard Entertainment well after its release; the last patch to it was in 2009, over a decade after its release.

In 2008, a project called Brood War API (BWAPI) was started ("BWAPI" 2012). API stands for Application Programming Interface. It is a category of tools that programmers use to help develop applications on computers. BWAPI allows developers to program artificial intelligences for StarCraft: Brood War using the programming language C++. An example of a project using BWAPI is the Berkeley Overmind Project at the University of California ("The Berkeley Overmind Project"). The Overmind AI plays the Zerg race of StarCraft: Brood War and uses swarm behavior to play. Swarm behavior in StarCraft's case is when a player, human or AI, use lots of units to overwhelm and destroy an opponent even though the opponent's units may be more powerful individually. BWAPI limits the developer to the same constraints a human player would have. These constraints are that they must play only as one race for the entire duration of the game and they may not try to execute more than 21 actions per unit per second on the “fastest” speed setting. However, BWAPI does not provide a perfect imitation of what a human player would see. In BWAPI, an AI can see the entire terrain of the map even if part of the terrain should be hidden under the cover of limited visibility, commonly known in RTS games as “Fog of War.”

RTS games provide different challenges for AIs in the areas of spatial reasoning challenges, perceptual information, and imperfect and inaccurate information (Buro 2003). Each of these challenges has real world implications as well and are not confined solely to StarCraft: Brood War. An example of a real-world application with challenges in spatial reasoning, perceptual information, and imperfect and inaccurate information is the DARPA Grand Challenge, a prize competition for developing autonomous, driverless, cars. Each of these cars has to use some form of localized radar to navigate. Most of these cars use LIDAR, a system that sends back 3D data representing the world around the car. Besides having to deal with navigating the world, they also need to deal with a dynamic environment, for example people moving and interacting around the cars. The amount information leads to perceptual information challenges. In addition to this, the data the cars get back is imperfect in that it may have gaps in it and is not a perfect model of the world around the cars. There is also inaccurate information because there is a delay in receiving data from the sensors on the cars. Both this delay and the fact that the data the cars get back may be inaccurate or misleading are examples of inaccurate information challenges. StarCraft is an environment that features similar challenges while being a virtual, safe environment where damage can be reset at the developer’s will. Thus, the development of AIs playing StarCraft have implications beyond just being more fun for a human to play against. The methods they use to solve these challenges can be applied to the real world.

In StarCraft, there are built-in scripted AIs. There are four different difficulties for each AI script. There are three different AI behavior scripts, one for each playable race.

On easy, medium, and hard difficulties, the AI uses different scripts for determining the order of buildings and units to build. They also are scripted to use different attacks. In order to respond to situations, which they are not designed for, they all include a generalized script for responding, which is neither efficient nor effective. The fourth AI difficulty, “insane,” is even more scripted. Insane is scripted to rush the opponent at specific intervals and is designed to be the hardest challenge for a player other than another human. The insane AI also cheats. The insane AI gains seven resources for every five that a player would gain. This means that for every five workers a player builds, the insane AI will be able to build seven. In contrast to the StarCraft scripted, potentially cheating, AIs, Soar-SC is designed to be a “smart” AI, one that is not scripted and can react efficiently and effectively to unseen and unknown challenges and strategies. The purpose of Soar-SC is to be a platform for future research into these challenges. Future research will allow for advancements in AIs confined to virtual worlds as well as AIs that interact with the real world.

Platform Design

BWAPI

BWAPI is a Dynamically Loaded Library (DLL). BWAPI works by overriding specific functions in StarCraft and then performs its actions within those overridden functions. BWAPI’s interface is pure C++. BWAPI is injected into the StarCraft process using a DLL injector. In our case, we use Chaos Launcher, the most common injector available for injecting DLLs into StarCraft. Soar-SC is also a DLL, which BWAPI loads into the StarCraft process when StarCraft starts a match. BWAPI allows an AI to be notified when events occur. Soar-SC uses the onFrame event, which is called whenever StarCraft tries to render a frame. Soar-SC also uses events for determining when a unit is created or destroyed, becomes visible or disappears, and when the game starts.

Soar

The Soar Cognitive Architecture is a general architecture, sometimes known as a framework, for developing AIs that exhibit intelligent behavior. Soar is designed to model the functionality of the human mind, not the details of the internal workings of the human brain. It has been used extensively in designing AIs (Jones et. all 1999). Soar also has various memories in which a Soar Agent can store knowledge to use. A Soar agent is the collection of Soar code that is interpreted by Soar to perform interactions with its memory and the world in which it interacts. Soar agents are written in a specialized programming language for Soar commonly referred to as Soar code. Soar also has specialized memories for Soar agents, two of which are used by Soar-SC. Soar-SC uses both Working Memory and Procedural Memory. Soar-SC also uses Soar's Spatial Visual System (SVS), an extension to Soar for specialized processing of spatial and visual information (Lathrop, Wintermute, and Laird 2011; Wintermute 2010; Wintermute 2009), currently under development by the Soar Group at the University of Michigan and not yet included in the main Soar package.

Working Memory is where a Soar agent's internal representation of the current situation including its beliefs gained from perception, the results of internal reasoning, and knowledge retrieved from long term memory. The data is represented as a symbolic, labeled graph of working memory elements (WME). Each WME is represented as a node on the graph of Working Memory. A WME node is an object that can contain information. An example of a WME is the amount of money the agent can spend on buildings and other units. Other WMEs are Procedural Memory is where behaviors and actions in Soar are represented as production rules. Rules in Soar work by matching conditions. Decisions in Soar are goal directed. Goals in Soar are known as operators that are represented in working memory.

Operators are specialized objects in working memory. Candidate operators are operators that are proposed by rules. They are only proposed when the conditions specified in the **proposal rules** match. A **proposal rule** is a rule that proposes a candidate operator based on conditions found in the current state. All candidate operators are also proposed simultaneously as a consequence of all rules firing simultaneously. This is unlike most programming languages and modern computers where instructions are sequential. Once operators have been proposed, a selection mechanism makes a decision as to which operators to apply based on selection knowledge. Selection knowledge is knowledge created from preference rules, rules that prioritize one operator over another. An example of a Soar-SC preference rule would be a rule that preferences building a worker over building a building. An operator is applied by an **apply rule**. An **apply rule** is a rule that tests for an operator being selected as the current action of the current state. This **apply rule** then performs an action that may modify WMEs or perform an action in the world.

Soar also has impasses. Impasses occur when the knowledge to select an operator is missing. An example of this would be when no operators can be proposed based on the current Working Memory state. Impasses can also occur when implementation knowledge is missing. An example of this would be when an operator was proposed but requires additional information not found in working memory to perform an action. An impasse is a condition in a state. When an impasse occurs, a substate is created to resolve the impasse. Each substate has a link to the state that created it. This is called the superstate. A substate may resolve the impasse in multiple ways. In an impasse where there is no candidate operator, the AI would resolve it by modifying working memory in the superstate to create conditions in which an operator could be proposed. A substate can also create further substates through impasses. An example of a substate would be when the AI wants to move a worker to the closest fog of war tile. In this case, the AI may propose the operator "move-worker" but it does not know which tile is the closest. When the "move-worker" operator is selected, a substate will be created because the **apply rule** of the operator cannot fire, execute, because it is missing which tile to go to. In the substate, the AI may have a series of operators determine which tile is the closest by using SVS. Once the query to SVS returns a result, an operator will match and fire to solve the impasse. This operator will create a WME on the **move-worker** operator, which created the impasse, the closest tile. To place means to add a node onto another node. Once the "move-worker" operator has which tile is the closest, the substate will be destroyed and then the "move-worker" operator's **apply rule** will fire and perform the action of moving the worker to the closest tile.

Soar uses a decision cycle for making decisions and actions in an environment. Soar's decision cycle consists of five phases. The first phase is called the **input phase** and is where all input to Soar is given. The second phase is the **proposal phase** where all operators are proposed. The third phase is the **decision phase** where an operator is selected. The fourth phase is the **apply phase** where the operator is fired. The fifth and last phase is the **output phase** where Soar outputs to the environment, or middleware, in Soar-SC's case, the action it wants to perform by placing objects on the output link. The middleware is the C++ portion of Soar-SC that handles communications between BWAPI and Soar.

Each state in Soar contains an input link and an output link. An input link is an identifier in working memory (WM) created by Soar. An identifier is a node that can contain other identifiers or WMEs. Each input link and output link is unique in that there is no duplication of data on each input link unless the AI creates it. The input link is where Soar-SC places information for the agent to use. An example is the units the Soar agent controls. An output link is an identifier in WM created by Soar. A unique output link exists for every state and substate. The output link is where the Soar agent places all output generated. For instance, if the agent selects an action to move a unit, it will place a "move" WME on the output link containing the Unique Identifier (ID) of the unit and where it wants to move it.

Soar-SC

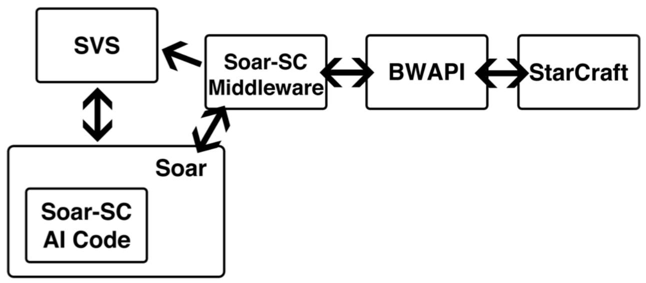

Figure 1: Diagram of Soar-SC

Soar-SC communicates with Soar through its C++ Soar Markup Language (SML) interface. Neither BWAPI nor Soar SML are multi-threaded or thread-safe. Soar-SC however, is heavily multi-threaded. Soar-SC has a terrain mapping thread, two events threads, and a soar agent thread. Soar-SC uses a series of mutexes, objects that prevent simultaneous access to shared data in memory, to manage and interact with all the different threads.

When Soar-SC is loaded into the StarCraft process, it spawns several threads. When Soar-SC spawns a thread, it creates a new thread for a specific purpose. The first two threads spawned are the two events threads. One of the events threads handles input to Soar. Every time an object is added to SVS, or input is given to Soar, an event class is created. This event class goes into the queue of events for Soar. When Soar is in its input phase, all the events in the queue are executed. New threads are only created for events that do not terminate until the agent is done executing or a "stop" command is issued.

The other event thread handles output events from the Soar agent. In the output phase of the Soar agent, Soar-SC creates an event around the output. A BWAPI Event class is created to handle the event. This new object is put into a queue of BWAPI Events. When BWAPI calls the onFrame event, Soar-SC tries to execute events in the BWAPI Event queue. Soar-SC cannot execute more than one event per unit per latency frame. A latency frame is a frame in which all events have been executed successfully or not and new events can occur on units. StarCraft has 15 latency frames per second as a default however, that can be increased and decreased. Soar-SC uses what most professional StarCraft players use, which is the "fastest" setting, 21 latency frames per second.

After the event threads are spawned, the terrain thread is spawned. In StarCraft, there are walk-tiles and build-tiles. A build-tile is what is used to measure the size of the map. A 64 by 64 map has 4096 build tiles on it. These build tiles are used for building sizes; all buildings are a multiple of build tiles. A walk-tile is one 16th as big as a build tile. A build tile is 32x32 pixels and a walk tile is an 8x8 pixel tile. There are 16 walk tiles per build tile. BWAPI allows the AI or platform to poll for whether a walk-tile is "walkable" or not. "Walkable" means that a unit can be placed there. The terrain can be represented as walk-tiles however this is very inefficient. Soar-SC generates the terrain map and then puts it into SVS so the Soar agent portion of Soar-SC can use it. Soar-SC only gives SVS vertices and locations. SVS cannot handle large numbers of cubes though, at least on modest hardware. Originally, Soar-SC created a cube for every walk-tile and then put the cube into SVS. This would have led to 65536 cubes being put into SVS if every tile were not walkable. Soar-SC has been tested on a MacBook Pro running Windows 7 natively. This laptop has a 2.8GHz Core 2 Duo made by Intel. On this hardware, it took upwards of one minute to load everything into SVS if all nonwalkable tiles were placed in SVS. To compensate for this, Soar-SC uses a simplistic algorithm for generating the largest rectilinear polygons that can represent the terrain. Soar-SC places every walk tile in a two dimensional Boolean array. The outer dimension of the array is the y coordinate, the inner dimension of the array is the x coordinate. Each (y, x) point is mapped to a Boolean, which represents whether a tile is walkable or not. Soar-SC starts at (0,0) and loops across the x and then increments y by one, loops across x, and so on. Every time it finds a non-walkable tile it checks to see if it is contained in an existing rectangle. If the tile is not, it tries to generate the largest possible rectangle. Soar-SC does this by incrementing along the y-axis until it finds a walkable tile or the edge of the map. Once it has found this, it records this value. Then for each row between the initial walk-tile point and the walk-tile point plus the y distance, it increments along them and records the distance to the nearest unwalkable tile. It then does this for the columns along the x-axis. It then takes the minimum distance to an unwalkable tile along the x-axis and the minimum distance to an unwalkable tile along the y-axis and generates a rectangle from that. This allows the terrain to be represented by a manageable number of polygons.

After Soar-SC has generated the terrain for SVS, it spawns the Soar agent thread. This thread is where the Soar agent's run function is called. Soar runs in this thread until "stop" is called on the agent at which point the thread will exit. The debugger when trying to debug the AI calls "Stop". "Stop" is also called at the end of the game to terminate the AI.

Agent Design

StarCraft

The agent plays as the Terran race in StarCraft. This is the most strategically balanced race. StarCraft also has Zerg and Protoss. The Zerg race is geared towards swarm behavior while Protoss is geared towards individual units. Terran is balanced between these two. Terran is sometimes referred as the “human” race in StarCraft. Currently the agent can only use two ground units, marines, and Space Construction Vehicles (SCV). Marines in StarCraft are the equivalent of modern naval-based marines in space. SCVs are the workers of the Terran race, they gather resources, build buildings, repair vehicles, and can attack an enemy if necessary. Soar-SC, however, does have knowledge of other units besides these. Currently on the input link, there is a node that has all building types and units types as WMEs. The soar agent uses these WMEs for building buildings, building units, and moving units around including attacking.

Limited Visibility

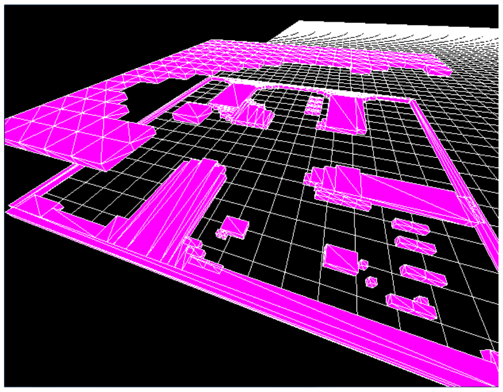
Figure 2: Soar-SC’s representation of StarCraft. The tiles in the top layer are the Fog of War layer after exploring the map a while.

In RTS games, there is limited visibility that is sometimes referred to as “Fog of War.” It is a covering put over areas of the map that have not been explored, or areas that do not have player units currently near them. In Soar this is represented in SVS as a series of 4x4 build tiles above the terrain. As the agent explores the map, these tiles are removed from SVS. The agent uses these tiles to determine how much of the map is explored and how much is left to explore.

Scouting

Soar-SC can scout a map. It scouts a map to determine where the enemy’s starting location is and to determine where the enemy’s buildings and units are. It does this by using scouting operators. To start scouting a map it first designates a unit as a scout by setting a scouting flag on it to 1 or true. Once the unit is

Figure 3: Soar-SC starting to scout the map

designated as a scout, then a scouting operator will be proposed. This operator will not be able to fire because it does not have a location. A substate will be created to figure out which fog of war tile is the closest.

When the substate is proposed, it will propose and fire an operator to query SVS for which is the closest fog of war tile. Once SVS returns a result, then a preference rule will fire. This preference rule will place a location WME on the scouting operator.

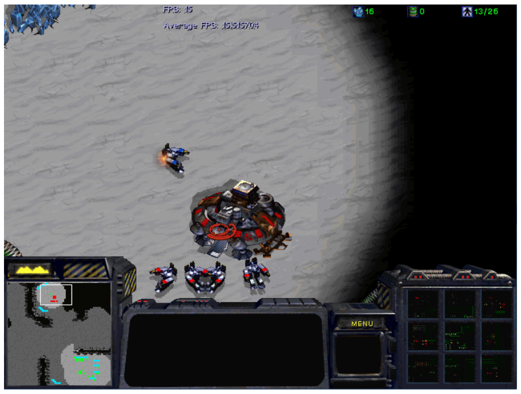
Figure 4: Soar-SC scouting the enemy base

Once the scouting operator has a location, an **apply rule** will fire. This **apply rule** will try to move the scout to the closest fog of war tile. It will place a “move” WME on the output link which contains a location and which unit to move. After the scout has moved to the fog of war tile, this process will repeat. The scout also has time outs. If the scout is unable to reach a certain area it will time out. A time out in this case is not that a certain time has elapsed and the scout has not reached the location yet, rather it is that the scout is idle and is not at the location of the fog of war tile. If this is the case, then the agent will mark the fog of war tile as timed out and try to find another tile to explore. It will continue to do this until there are no more fog of war tiles that are not marked as timed out or there are no more fog of war tiles.

Building Buildings

Soar-SC can currently build buildings. It can only build barracks and supply depots though. Barracks are where the agent builds marines, and supply depots are what allows it to increase the maximum number of units it can build.

To build a building, the Soar agent first proposes an operator to build a building. If the Soar agent wants to build a Barracks for instance, it will try to propose a build-barracks operator. Soar then checks for whether the **proposal rule** matches against the current state. The current state is everything in WM. In the case of a **proposal rule** to build a barracks, it checks whether there is a “Terran Barracks” type within the “types” Identifier on the input link. It will also check for a “Terran SCV” type on the “types”

Identifier as well. It then checks whether there is a unit, which has the “Terran SCV” type ID. It also checks to make sure there is not a SCV currently building a barracks. If those conditions match then the proposed operator will become a candidate operator. If the operator is selected then an **apply rule** for the operator will be fired.

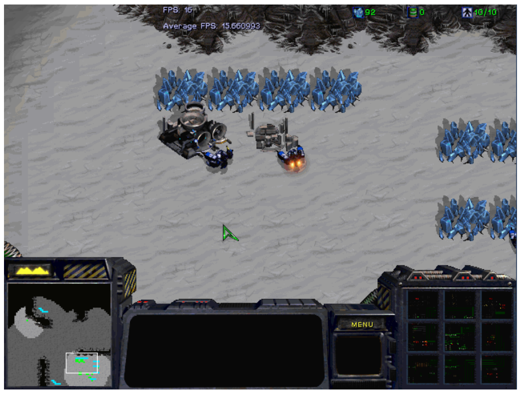
Figure 5: Soar-SC building two supply depots

In the case of buildings, the **apply rule** is a “apply*build-building.” This rule will only fire if the proposed operator has building type ID, a worker ID, and a location in x and y build coordinates. Soar-SC can build any type of building at any location, however, the barracks locations and supply depots locations are currently hard coded. Once the “apply*build-building” rule is fired, it puts on the output link a “build-building” WME. This WME contains the location along with the type of building to build and which worker to use to build it. The rule also decrements the agent’s internal budget of materials to prevent it from trying to build multiple buildings without the necessary resources. Each action is not guaranteed to be executed before the next decision cycle because of StarCraft’s latency frames.

Building Units

Soar-SC can also build units. It can currently build two types of units, marines and SCVs. SCVs are built by the Command Center while a barracks builds marines. StarCraft automatically creates a Command Center for the player when the game starts. There can be multiple Command Centers but Soar-SC currently only uses one. Soar-SC also will try to build as many units as possible currently.

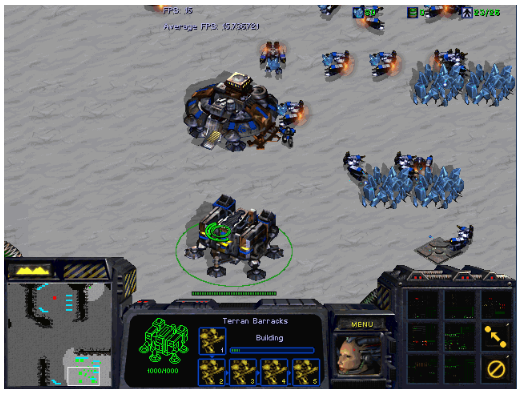

Figure 6: Soar-SC building marines

Building units are very similar to building buildings. To build a unit, a **proposal rule** is fired for a specific unit. If the agent wants to build a SCV, then a "propose*build-scv" rule is fired. This rule checks whether there is a command center and a "Terran SCV" type on the "types" node on the input link. If there is and there are enough resources, then a candidate operator will be proposed. This operator contains which unit type to build and where to build it.

If the operator is selected, then an "apply*build-unit" rule will be fired if some conditions match. These conditions are whether the operator has a location on it and there is a unit type ID on it. If these conditions match, then a "build-unit" WME will be created on the output link. This build-unit WME has the type of unit to build along with which building to be designated to create it.

Gathering Resources

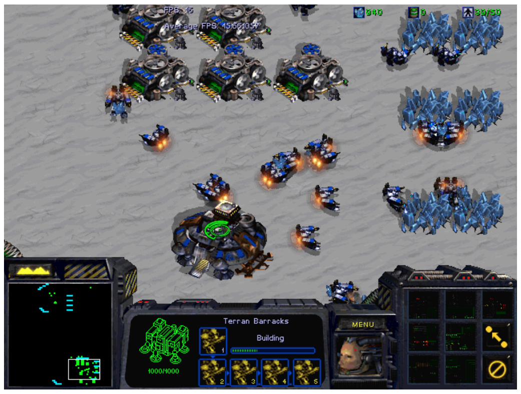

Figure 7: Soar-SC gathering resources

When StarCraft starts the game, there is one Command Center and four SCVs. Soar-SC will try to gather resources with the idle SCVs. To do this a gather resources operator is proposed and fired if there is an idle SCV and a mineral patch. This operator however will not be able to fire. When it is proposed, it is proposed without a mineral patch to gather from. To figure out which mineral patch to send the SCV to, a substate is created.

In the substate, operators are proposed and fired to determine which mineral patch is the closest. An operator is created to query SVS for which is the closest mineral patch. Once SVS returns a result, then a preference rule fires to place a WME on the gather minerals operator, which contains which mineral patch to go to.

When the operator has the "mineral-patch" WME on it, then the **apply rule** can fire. Since it can fire, the substate will be removed and the "apply*gather-minerals" will fire. It creates a move WME on the output link to move the SCV to the mineral patch.

The SCV will continue to gather resources until it is interrupted by an outside event or there are no more mineral patches within its sight range.

Attacking the Enemy

Soar-SC can also attack and defeat the default StarCraft AI on medium. The default AI on easy will build barracks, build marines, build SCVs, gather resources, and build more advanced buildings given enough time. The medium AI is similar to the easy AI but will be more efficient about where it places buildings and be faster about building units and buildings. Currently when the Soar agent sees an enemy, it is added to the input link. If there is an Enemy on the input link and there is a marine, then a **proposal rule** will be fired. This **proposal rule** will create an attack operator. The operator has a location and a unit on it. The location is which enemy to attack. The unit is the marine to use to attack the enemy. If the operator is selected, an attack WME will be created on the output link. This WME will contain the location and which unit to send. Soar-SC will then send the unit to attack the enemy. While it is attacking, no other operators will operate on the unit to make it do something else, except one, the counter attack operator.

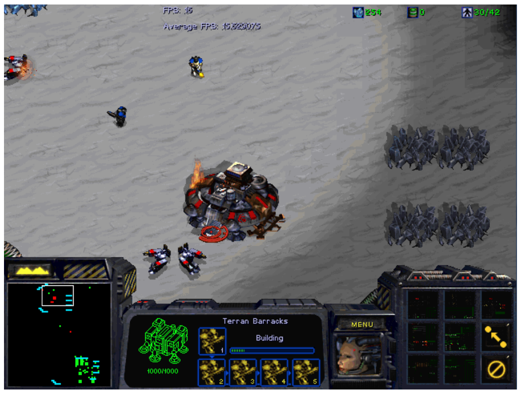

Figure 8: Soar-SC attacking an opponent

Whenever a marine becomes the target of an enemy unit, a **proposal rule** will fire proposing a specialized attack operator. This attack operator has for its location the unit that is currently targeting the marine. It also has which unit to attack with. This unit is the unit, which is the target of the location unit. The operator will then be selected, as it is always the “best” choice, highest priority, operator out of all the operators in Soar-SC. When it is selected, the apply attack rule will fire and stop the unit from doing whatever it is doing and attack the unit targeting it.

Future Work

Soar-SC was created from the ground up to play StarCraft. Soar-SC uses BWAPI, Soar, and the SVS extension to Soar. At present, Soar-SC can defeat the built-in StarCraft AI on medium difficulty and sometimes on the hard difficulty. In StarCraft: Brood War there are several built in AIs. However, the AIs are scripted AIs. On the hardest difficulty, “insane,” the AI starts to cheat. The AI gains 7 resources for every 5 a normal player gains. It also uses scripted rushes; it rushes at predetermined pointers without much in regards to strategy. Soar-SC, however, is designed to not cheat and use strategy to win, and Soar-SC is designed to be a “smart” AI. Soar-SC gathers resources, builds buildings, builds units, scouts the map, and attacks the enemy without cheating in any way.

Soar-SC is intended as a platform for research. An example of a future research area is using virtual objects in conjunction with SVS. Currently, all the building positions are hard coded, however, using virtual objects would change this. Virtual objects are objects that the AI creates temporarily to reason about where to place buildings. Soar-SC could use these objects to determine where is the best place to place a new barracks or supply depot. Virtual objects could also allow the expansion of new types of buildings. Using virtual objects, it would be simpler to add the ability to use new types of buildings, which in turn would allow Soar-SC to use new types of units.

Another example of future research is learning. Soar has two different types of learning, reinforcement learning and chunking. Reinforcement learning is learning through positive and negative feedback. An example of this would be a mouse learning to pick one of two doors to go through, in one it would receive a shock and the other it would receive a treat. The shock is the negative feedback and the reward is the treat. Reinforcement learning in StarCraft would be a challenging but potentially worthwhile thing to do. It would be difficult because in the environment, there is never the same situation twice and reinforcement learning tweaks existing rules. To tweak them in a general, effective, way will be a challenge to implement. Chunking is a mechanism Soar has to prevent having to compute the same result twice. When Soar goes into a substate and generates a result, chunking can be used to prevent it from computing the same result again. Chunking could potentially speed up Soar-SC reactions over time and combined with reinforcement learning could benefit Soar-SC greatly.

A third area for future research is strategy and tactics. Soar-SC currently has minimal strategy. Soar-SC can be used for strategy and tactics research though. Future research in strategies and tactics using Soar-SC could involve determining different strategies to counter other strategies. An example of this would be determining how to counter the Overmind AI's swarm tactics. Another example would be determining how to create rules that would allow Soar to adapt to different strategies as the game goes on, and adapt to new circumstances it has not encountered before in an effective way.

Soar-SC is designed as a platform for research into the challenges associated with RTS games. As mentioned earlier in this paper, AIs face challenges in RTS games. These challenges are the complexity of spatial reasoning, processing perceptual information, and interpreting imperfect and inaccurate information. Implementing virtual objects would help solve the spatial reasoning and processing perceptual information challenges. Using virtual objects to build would require understanding the terrain and already placed objects in the StarCraft environment. Solving the challenge of virtual objects would help solve the challenge complexity of spatial reasoning by determining an efficient way and effective way to manage placing new buildings in the environment. Virtual objects would work by trying to place a building in an area and determining whether it is possible to actually build there by determining whether it collides with other objects or not. However, virtual objects can be used even more, optimality of the location. This could include factors such as whether workers gathering resources frequently cross where the building would be placed and whether it is a good location for the type of building. For example, if the AI wanted to build a barracks it would not want to build in an area that has little free space around it because it would prevent it from easily building marines.

On top of managing the spatial reasoning challenge, there is an additional component to virtual objects: strategy. Building new buildings using virtual objects could be further compounded on and improved to incorporate strategy. Strategy could be used for creating choke points to force enemy attacks to funnel through an area to increase the effectiveness of Soar-SC's counter attack. Another area of strategy that could be applied is organizing the more vital buildings in locations, which are more heavily protected. StarCraft also has the equivalent of Missile Turrets that can shoot down enemy air units and function as detectors of cloaked enemy units. Cloaked units are units that are invisible unless there is a unit that is a detector near them. A detector is a type of unit, which allows a player to see enemy units that are cloaked. Virtual objects could be used to determine which is the optimal place to place the Missile Turrets to protect the base. Each of these examples would help solve the spatial reasoning challenge by creating algorithms and strategies to do each of these examples effectively and efficiently. Soar-SC has the potential to aide in solving numerous complex challenges that RTS games have. Solving these challenges will allow for advancements in other areas of artificial intelligence.

References

The Berkeley Overmind Project. (n.d.). In Berkeley Overmind. Retrieved October 17, 2012, from http://overmind.cs.berkeley.edu.

Buro, M. (2003), Real-time strategy games: A new AI research challenge. In Proceedings of the International Joint Conference on AI, 485–486.

BWAPI. (October 15, 2012). In Google Code. Retrieved October 17, 2012, from http://code.google.com/p/bwapi.

Jones, R. M., Laird, J. E., Nielsen, P. E., Coulter, K. J., Kenny, P., & Koss, F. V. (1999). Automated intelligent pilots for combat flight simulation. AI Magazine, 20(1), 27-41.

Lathrop, S. D., Wintermute, S., Laird, J. E. (2011), Exploring the Functional Advantages of Spatial and Visual Cognition from an Architectural Perspective. Topics in Cognitive Science, 3, 796-818.

Wintermute, S. (2010). *Abstraction, Imagery, and Control in Cognitive Architecture*. PhD Thesis, University of Michigan, Ann Arbor.

Wintermute, S. (2009). *An Overview of Spatial Processing in Soar/SVS* (Report No. CCA-TR-2009-01). Ann Arbor, MI: Center for Cognitive Architecture, University of Michigan