

A Sandbox to Learn and Develop Soar Agents

Daniel Lugo and Douglas D. Hodson

Department of Electrical and Computer Engineering

Air Force Institute of Technology

Wright-Patterson Air Force Base, Ohio 45433

Daniel.Lugo@afit.edu.com, Douglas.Hodson@afit.edu

Abstract— Military personnel leverage simulations (and simulators) as cost-effective tools to train and become proficient at various tasks (e.g., flying an aircraft and/or performing a mission, among others). These training simulations often need to represent humans within the simulated world in a realistic manner. ‘Realistic’ implies creating simulated humans that exhibit behaviors, which mimic real-world decision making, and actions. Typically, to create the decision-making logic, techniques developed from the domain of artificial intelligence are used. Although there are several approaches to developing intelligent agents: we focus on leveraging the open source Soar architecture to define agent behavior. This research interfaced another open-source software product that facilitates the creation of 3D virtual worlds (called the AI sandbox) to the Soar package. Because the world created by the sandbox is rich in features, easily configurable using a simple scripting system, and visually engaging, it is ideal as a learning platform to develop Soar agents. In summary, this research developed a platform (or learning environment) to learn how to develop Soar-based agents in a friendly, engaging, visual 3D environment.

Index Terms — Military Training Simulations, Soar, Lua, Intelligent Agents

I. INTRODUCTION

Because of the critical nature of military missions to the country, it is important to be able to practice what we learn, test what we use, and evaluate what we need [1]. Armed forces must train to develop and maintain the proficiency necessary to effectually carry out the duties with which they are entrusted by their nations [2]. Unfortunately, the very nature of military missions makes it hard to set up real-world training, as mimicking a military mission in the real world requires many people and resources [3]. Simulations can greatly help the military perform this task. They also mitigate the cost of operating weapons systems and the risk of equipment damage, injuries, or loss of life [4].

Simulations, especially the ones used in training, need to provide the humans being trained a sense of realism so that they can be effective in preparing trainees for real-world situations. One way to make training simulations more realistic is to use human operators that perform different roles in the simulation. Human operators help create a more diverse and realistic environment for training [3]; however using human operators might not be the most viable solution because skilled workers is limited and costly. An alternative is to simulate the human operators that perform the various roles in the simulated

environment [3]. This is accomplished by creating artificial autonomous intelligent agents to perform the same tasks the human operators would in the virtual world. If these intelligent agents are able to produce realistic behavior similar to a real human operator, then the cost of having humans involved in training can be mitigated and we solve the issue of not having enough skilled operators to make training successful.

One way to create an intelligent agent is to use a cognitive architecture. A cognitive architecture defines an underlying infrastructure for an intelligent system; it provides us a definition of the computational structures required to store, retrieve, and process knowledge [5], [6]. For this research, we decided to use the Soar cognitive architecture. Soar is a well-defined mature framework. Moreover, it has been integrated with several systems including Joint Semi-Automated Forces (JSAF), which is a U.S. government owned and developed simulation system that is widely used in training and experimentation [7]. Soar is open source and available for Windows, iOS, Linux and Android operating systems.

We are interested in using Soar to create intelligent agents, due to the architecture’s ability to create agents that can learn and resolve various problems, called impasses. But agent development is challenging, as Soar uses its own language to define agents that is very different from other widely used programming languages like Java, C, C++, Python, C#, etc. Another issue with Soar is that the lack of example agents that have the tactical goals used in military simulations. In order to learn to create intelligent agents using Soar, we interfaced it to a virtual, interactive visual environment to test agent behaviors. Having such an environment helps in learning how to program Soar agents because the user can see what the agents are doing within a visually engaging situation.

During our research, we encountered several environments used to learn the basics of Soar agent definitions. These environments consist often consisted of simple two-dimensional grid worlds where the agents make decisions. They are useful for introducing users to Soar agent definition but are limited to just simple behaviors. We wanted a rich, complex and dynamic 3D environment for Soar agents to interact with; specifically, an environment where agents are capable of performing actions similar to military tactics would be beneficial. For example, moving at different speeds, following a leader, engaging an enemy, traveling to waypoints, and fleeing from danger are all actions that would make an environment more desirable.

Intelligent agents need to gather information from the world. Training in virtual environments provides lower cost than training in the real world. Virtual environments benefit from having a visual representation because it aids in understanding why the agent performs certain tasks. Since we can see the agent moving, acting upon the environment and responding to what other agents do, we are able to better understand how our agent sees the world and makes decisions. Another aspect of virtual environments that is beneficial in testing intelligent agents is the ability to be interactive. The environment can react and respond to what the agent does; this will give us the capability to create more complex agents that continually look at the world to decide what to do next. The ideal environment to learn to create intelligent agents should have various options for the agents to perform. These options can include how the agent move, perception of the environment, communication with other agents, etc. The environment will dictate what the agent is able to do in the simulation.

While researching intelligent agents, we found a sandbox environment used to learn artificial intelligence game programming using a combination of C++ and Lua scripts. We selected it as our environment to learn to program Soar agents because it has several of the characteristics mentioned above. This effort extended this sandbox environment to use Soar agents.

II. SOAR COGNITIVE ARCHITECTURE

Soar is a cognitive architecture, created by John Laird, Allen Newell, and Paul Rosenbloom at Carnegie Mellon University. It is considered an architecture for constructing general intelligent systems. Soar has been in use since 1983, and has evolved through many different versions [8]. As opposed to many artificial intelligence systems that have been designed to excel at performing a single task, Soar was designed with a more general purpose vision in mind [8]. Humans are able to perform many different tasks in an ever-changing environment; we gather information and learn constantly. Because of this, creating an intelligent agent that has similar abilities to humans is very different than designing a system that is able to solve a certain type of problem very well. This general approach to agent development covers a diverse set of domains and it is one of the reasons that Soar was selected for our purpose.

A. Capabilities of Soar

Figure 1 presents a structural view of the Soar cognitive architecture. Soar receives input from the external environment using what is called the perception module. Changes to perception module are processed and sent to the working memory, which represents short-term memory. Working memory also initiates retrieval of long-term memories [6]. The different types of long-term memories are independent of each other and they have their own learning mechanisms. Procedural long-term knowledge is composed of production rules that are organized by operators. Operators can describe simple actions or more abstract behaviors. Episodic long-term memory holds the history of the previous states, while semantic long-term memory contains previously known facts [8]. With the input from the environment, and the knowledge that exist on working

and long-term memories Soar can then make decisions. These decisions invoked through the action module. In summary, Soar can perceive information, use and create knowledge, learn from experience, and make decisions that in turn create behaviors to perform actions. With these fundamental capabilities, Soar supports planning, decision-making and problem solving.

B. Soar Operators and State

The design of Soar is based on the hypothesis that all deliberate goal-oriented behavior can be cast as the selection and application of operators to a state [9]. When Soar is running, it is trying to match operators to the situation (current state) until the goal state is reached. This is why initially Soar was an acronym meaning, State, Operator and Result; today this acronym is no longer used. The state is represented in working memory as a connected graph structure where the state is the root [10].

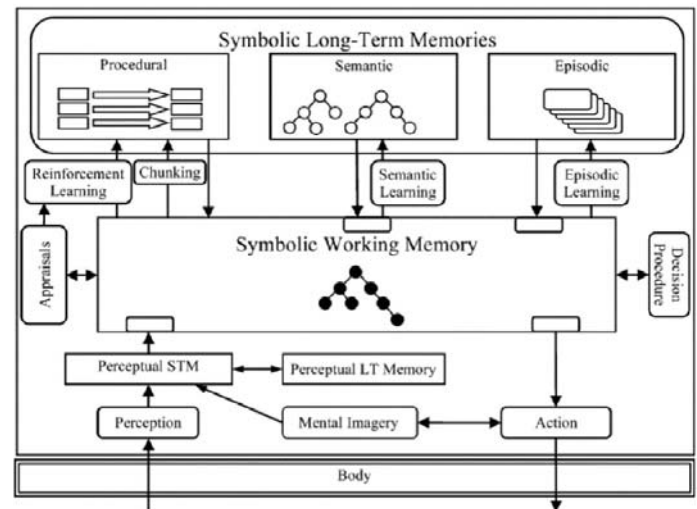


Figure 1. Soar Structure [6]

Operators are defined by rules which consist of conditions and actions. Each operator must have at least two rules, a proposal rule and an application rule [10]. The proposal rule indicates when an operator should be checked and the application rule let us know what it does or how it changes the state. Operators are the ones that perform actions in the environment or in the agent's internal working memory. The internal working memory is composed of working memory elements or WMEs for short. Once operators are selected, additional rules can match and apply the operator. This gives Soar a flexible representation of operators that makes it possible to have conditional, disjoint sets of preconditions and actions [6].

C. Integrating Soar with External Environments

For Soar to interact with external environments, it uses a mechanism that receives inputs and send outputs. This mechanism is defined by the Soar Markup Language (SML) interface. Input and output are performed by functions executed in the input and output phases of the Soar processing cycle. The

structures for manipulating input and output in Soar are linked to a predefined attribute of the top-level state, called the io attribute [9]. These structures are called the input and output links and they either create Working Memory Elements (WMEs) or respond to WMEs that appear in the output structure. SML provides us the interface to so that input and output data structures can be sent back and forth from the Soar Kernel to the external tool or environment. SML is based around sending and receiving commands packaged as XML packets [11].

The SML Application Programming Interface (API) is based on C++ with wrappers available for both Java and Python using the Simplified Wrapper and Interface Generator (SWIG) package. This gives the users some options to develop the client that communicates with Soar. In Figure 2 we have an example communication block diagram using SML between Soar and an external environment. In this example, the environment creates an XML object that has information for Soar. That object is then converted into an XML stream (a sequence of characters) and it is sent in to Soar, where it is converted back into an XML object and used by the Soar Kernel. The communication will work the same way from Soar to the environment. This is only one way of communication between Soar and an external tool or environment, if Soar and the tool or environment exist in a single process, they will communicate back and forth using XML elements without converting them to streams. If Soar and the external tool or environment each run on a different machine they will communicate via sockets to send and receive an XML stream [12].

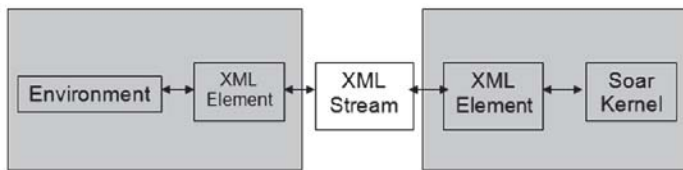


Figure 2. Communication diagram between external environment and Soar

The SML API is only one part of integrating Soar with an external environment. Code needs to be written on both sides of the communication, the environment and the Soar agent. On the environment side (the simulation), the incoming XML object needs to be translated in something useful for the environment to process and in the Soar agent rules need to be created to work with the input from the environment.

D. Soar Debugger

The Soar Debugger is a Java based tool created using SML API to aid in the development and debugging of Soar agents. It has the capability of stepping through the Soar processing cycle and display information about working memory, the operators that are selected, the rules that have been matched, and the status of the input and output links. The Soar Debugger can run agents locally or remotely, via a connected external Soar kernel. The user has the ability to add artificial breakpoints between Soar processing cycle phases. By doing so, the user can view the working memory after the decision or before the input phases.

III. VIRTUAL LEARNING ENVIRONMENT

A solution to the lack of a rich learning environment for Soar was developed that supported all the graphics and physics needed while leaving the decision making to Soar agents. This environment (called a sandbox) was implemented in C++ and uses open source libraries. The initial codebase is packaged with a book named Learning Game AI Programming with Lua – this book is focused on teaching readers how to programing agents using the Lua scripting language [13]. This book uses the sandbox environment to teach AI programing with Lua in a similar way that we would like to use it with Soar making this sandbox environment a natural choice to integrate Soar.

Some characteristics that make this sandbox environment an acceptable solution to learn Soar agent development are the ability to have multiple agents working in a team, having a three dimensional world, the agents have many movement options, and agents have the ability to shoot a projectile. These are important characteristics if we are to use Soar in military training applications.

A. Sandbox Design

The sandbox code is designed as a framework via integration of several open source libraries. The software code for the framework itself is located in one library, which provides the classes as points of extension – the interface to Soar being one.

From a developer's perspective, this framework library is the core of the sandbox and contains all the important classes of interest. The architecture defines an update, initialize, and cleanup phases for the simulation. The classes used for initialization and update were identified and modified to extend the sandbox functionality. Figure 3 shows a class diagram of some important classes and their relationship to each other for a running sandbox-based application.

Following are a list of classes and their purpose:

1. BaseApplication – responsible of configuring the application window, process input commands and interface with the Ogre3D library. It implements the Ogre::FrameListener interface for updating and drawing in the sandbox. It also implements Ogre::WindowEventListener to handle window events. BaseApplication also interfaces with the Object-Oriented Input System (OIS) library, which is responsible for all the keyboard and mouse handling events.

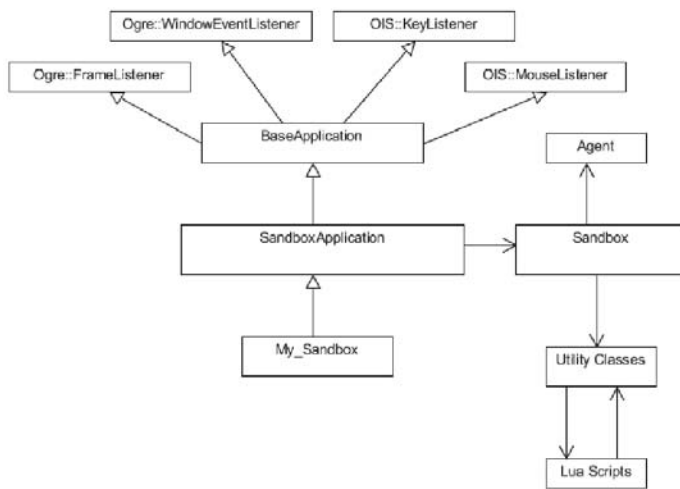


Figure 3. Class diagram of the important classes in the sandbox

2. **SandboxApplication** – this is a child class of **BaseApplication**. When the user creates an application they will be creating an instance of this class. This class implements the **Cleanup**, **Draw**, **Initialize** and **Update** functions inherited from the parent class. A function of interest defined by this class is **CreateSandbox**, which creates an instance of the sandbox and connects the sandbox with the Lua script that has the sandbox information.
3. **Sandbox** – this class handles the calls to the Lua script and has the sandbox data. It utilizes the **Ogre::SceneNode** to place the sandbox in the world.
4. **Agent** – this class encapsulates the agent data and handles calls to the agent Lua scripts.
5. The following utility classes are implementing the utility pattern, which separates the need for the agent and sandbox classes to know how to interact with the Lua VM. All the data manipulation interacting with the Lua VM, is performed by the utility classes [13].
 - a. **AgentUtilities** – handles all actions performed on AI agents from Lua
 - b. **SandboxUtilities** – handles actions performed on the sandbox from Lua
 - c. **LuaScriptBindings** – describes the C++ functions exposed to the Lua scripts.

IV. INITIAL TESTS AND CODE CHANGES

Before starting to modify the AI Sandbox code to be able to use Soar to control the agents, several tests were performed to verify that Soar was compatible with the development environment used by the sandbox. Once it was proven that Soar could communicate with the environment we made the code changes needed to the AI Sandbox.

A. Initial SML Testing

This section details the tests completed to verify Soar and SML in the development environment. The initial test was designed to create an instance of the Soar kernel and to populate

a WME in the input link. To set up the development environment the header file "sml_Client.h" was included and the Soar shared library was linked. The instance of the Soar kernel was created utilizing the command `Kernel::CreateKernelInNewThread()`. After testing that the kernel was created successfully the input link was created and information was committed to Soar's working memory. To verify if this was successful, we utilized the Soar debugger. By connecting the Soar debugger to the kernel (created in the SML application), we were able to inspect the contents of the input link. It was verified that the WME that was created in the initial test existed in the Soar kernel in the input link. Figure 4 shows a screenshot of the input link from the Soar debugger.

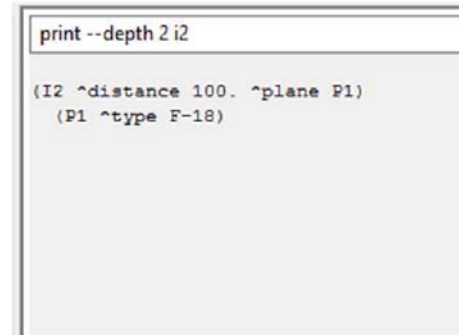


Figure 4. SML test, screenshot of input-link on Soar debugger

The second test was designed to check the communication between Soar and the SML client while utilizing a Soar agent's decision making capability. In this test, the agent needed to decide if it wanted to move or stop. The SML client sent the information about the world to the Soar agent. The world consisted of a long hallway with a wall at the end. This was represented in the code as an array. The array was populated with the string "empty" representing empty cells and "wall" to represent the wall at the end of the hall. The Soar agent receives from the SML client what is in front of it, and it decides if it should move forwards or stop. While the agent makes the decisions we can use the Soar debugger to verify that the Soar kernel was receiving input and producing output decisions. The test also used a command prompt to display debugging output from the C++ SML client. The test was successful and the agent was able to move to the end of the hall and stop when it encountered a wall in front. Figure 5 is a screenshot of the input link and Figure 6 shows the output link, which is the communication from Soar to the client.

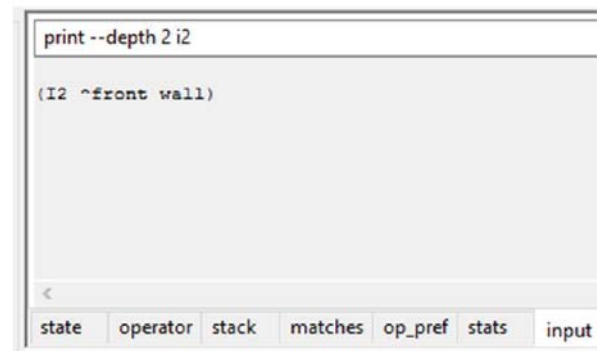


Figure 5. Input-link, simple environment test

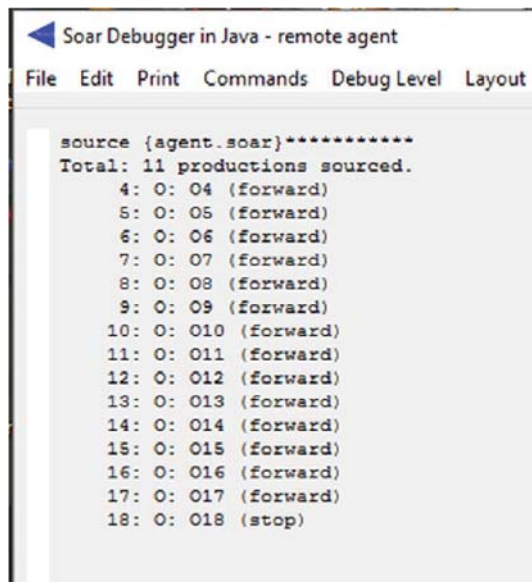
```

source {agent.soar}*****
Total: 11 productions sourced.
 6: O: 06 (forward)
 7: O: 07 (forward)
 8: O: 08 (forward)
 9: O: 09 (forward)
10: O: 010 (forward)
11: O: 011 (forward)
12: O: 012 (forward)
13: O: 013 (forward)
14: O: 014 (forward)
15: O: 015 (forward)
16: O: 016 (forward)
17: O: 017 (forward)
18: O: 018 (forward)
19: O: 019 (forward)
20: O: 020 (forward)
21: O: 021 (stop)
22: O: 022 (stop)
23: O: 023 (stop)

```

Figure 6. Soar agent output

Once the major elements of the SML API were tested the next step was to investigate other ways in which the SML client could manage the input and output link. The SML API has the capability of utilizing an event model framework in which the client can register for Soar events. There are many events you can register for and with each version of Soar that list grows [11]. To test the SML client ability to register and handle Soar events a test was conducted. The concept of the world as a long hallway was utilized here. The C++ client code was modified by registering for the Soar run event and the event handlers were created. A benefit of utilizing events was the ability to also control the simulation with the Soar debugger. Figure 7 shows the output from the agent as seen on the Soar debugger. Once these initial tests were complete it was time to study the sandbox environment and work on extending that application.



```

Soar Debugger in Java - remote agent
File Edit Print Commands Debug Level Layout

source {agent.soar}*****
Total: 11 productions sourced.
 4: O: 04 (forward)
 5: O: 05 (forward)
 6: O: 06 (forward)
 7: O: 07 (forward)
 8: O: 08 (forward)
 9: O: 09 (forward)
10: O: 010 (forward)
11: O: 011 (forward)
12: O: 012 (forward)
13: O: 013 (forward)
14: O: 014 (forward)
15: O: 015 (forward)
16: O: 016 (forward)
17: O: 017 (forward)
18: O: 018 (stop)

```

Figure 7. Soar agent output

B. Code Changes to the AI Sandbox

To have Soar agents controlling the entities in the sandbox, changes had to be made in the Agent class. The Agent class became a parent class for a Soar agent class. A Lua agent class had to be created to make use of the Lua script functionality. This gave us the ability to have both Lua and Soar agents in the same environment and kept the current agents created for the sandbox working. We added the capability of having Soar agents without affecting how the sandbox works.

A parent class named BaseAgent was created. It has all the base functionality that the Agent class had. Child classes named LuaAgent and SoarAgent were also created so that the communication with the Lua scripts and Soar kernel will be handled by dedicated classes. From researching the code and looking at where is the Agent class was used, several classes were identified as the ones that had to be modified. Here is a list of the code files that were changed:

- Agent – This class was kept in the project but functionality was replaced with BaseAgent, it includes some virtual methods for the SoarAgent class and LuaAgent class to implement.
- Sandbox – The Sandbox class keeps track of the agents by maintaining a list. Code was added to be able to keep lists for BaseAgent, LuaAgent and SoarAgent class instances.
- AgentGroup – This class handles agent groups, the methods that contain instances of Agent class were changed to use the BaseAgent class.
- AgentPath – Derived from OpenSteer, the methods that contain instances of Agent class were modified to use the BaseAgent class.
- AgentUtilities – Multiple methods to initialize, update and perform cleanup. This class had methods created for the SoarAgent class and LuaAgent class. This is the intermediary between Lua, Soar and the sandbox.
- SandboxUtilities – This class has methods to create Lua and Soar agents in the sandbox. New methods were written to perform these tasks.
- LuaScriptBindings – This class has the list of methods that can be called from the Lua script. Current methods involving the Agent class were changed to use LuaAgent and SoarAgent classes.
- SandboxObject – The Agent class is friend class to SandboxObject. The new implementation uses BaseAgent as a friend class to SandboxObject.
- NavigationUtilities – The default configuration for the agent movement is set up here. Code was modified to use BaseAgent class.

The new implementation of the Agent class is called BaseAgent and it has a child class named LuaAgent. The LuaAgent class works with the Lua scripts and the sandbox. The SoarAgent class is a child of LuaAgent and inherits all the functionality that helps with the communication of the Lua scripts and the sandbox. The SoarAgent class serves as the parent class for Soar agent classes. Figure 8 shows the diagram

of the new implementation of the agent classes. Users that want to create Soar agents will need to inherit from SoarAgent class. Users can also have Lua agents and Soar agents in the sandbox at the same time. This design uses the Lua scripts to set up the sandbox and handle the physics and animation of the agents. The Soar agent code will be the decision maker part of the agent. Each Soar agent has a Lua script associated with it.

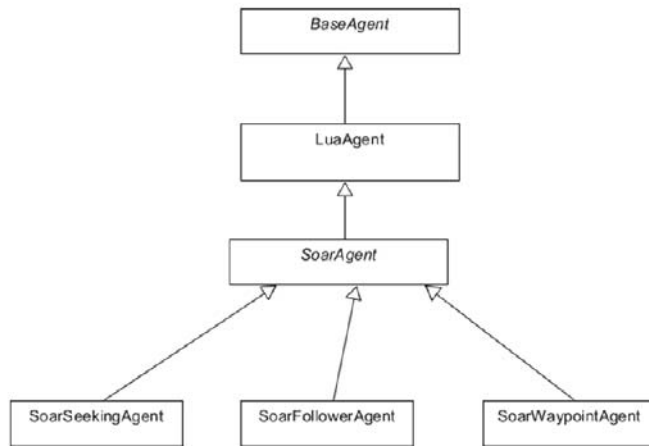


Figure 8. Diagram of Agent classes

V. SOAR AGENT TESTS

After the completion of the code changes necessary to be able to use the new agent classes, the next step was to create agents in the sandbox controlled by Soar. First, we create the sandbox project for the agents to exist on. The project my_sandbox was created for this purpose. This sandbox project includes the MySandbox class and the Lua scripts necessary to set up the sandbox. These Lua scripts generate the floor, light the sky, and initialize the agents in the sandbox. The Soar agents created for the tests are based on Lua agents available in the sandbox project code [13]. Creating Soar agents that mimic existing agents was a way to test the new functionality without having to design agents from the ground up.

A. Seeking Agent

The Soar seeking agent moves to a target point in the environment and once it reaches that point it randomly selects another point in the environment to set as its target. To create a Soar seeking agent a SoarSeekingAgent class was created. This includes a header file and a source file. This new class is a child of SoarAgent class and becomes part of the demo framework project. The SoarSeekingAgent class creates an instance of the Soar kernel; it also sends the environment information to the Soar agent and receives the output from Soar. Since the Lua scripts are responsible for the initialization of the sandbox, code needed to be added to the LuaScriptBindings and LuaScriptUtilities classes. This code makes it possible for the Lua script to create a SoarSeekingAgent instance. After the C++ code is complete the agent needs a Soar file with the decision-making code which runs on the Soar kernel, and a Lua script that handles the agent representation and movement. The Soar agent code will receive the distance parameter calculated by the

SoarSeekingAgent code and decide whether to continue heading towards the target point or choose a new target point. Having all the components ready the test was run. The Soar Debugger was used to view the input and output on the Soar kernel side. Figure 9 shows a screenshot of the output from the Soar agent from the debugger, line 1711 shows the output “newtarget” which was used by the SoarSeekingAgent code to initiate the new target process. Figure 10 is a screenshot of the sandbox with the agent and the target point.

```

1704: O: 01704 (sametarget)
1705: O: 01705 (sametarget)
1706: O: 01706 (sametarget)
1707: O: 01707 (sametarget)
1708: O: 01708 (sametarget)
1709: O: 01709 (sametarget)
1710: O: 01710 (sametarget)
1711: O: 01711 (newtarget)
1712: O: 01712 (sametarget)
1713: O: 01713 (sametarget)
1714: O: 01714 (sametarget)
1715: O: 01715 (sametarget)
  
```

Figure 9. Soar Seeking Agent output

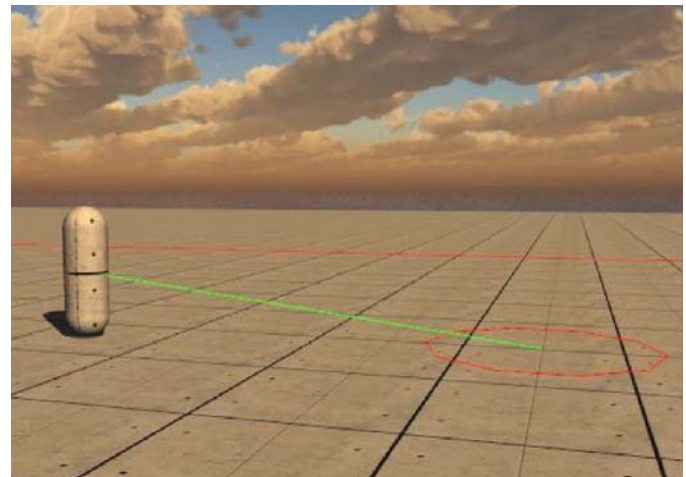


Figure 10. Screenshot of seeking agent

B. Follower Agent

The follower agent is similar to the seeking agent in because it needs to travel to a target point. In the case of the follower agent the target is another agent. This agent can be used to follow a leader or to pursue an enemy. The SoarFollowerAgent class was created as a child class of SoarAgent. This class adds a member named pLeader. The pLeader is a pointer to a BaseAgent element. Since Soar agents and Lua agents inherit from BaseAgent, this gives us the capability to have any type of agent as the leader. The utility classes and binding classes used by Lua scripts to call C++ functions were changed to add the capability of Lua to initialize follower agents. After updating the utility classes the MySandbox class has to be updated to initialize the different types of agents. For this test a Soar seeking agent was utilized as the leader.

Each frame rendered on the sandbox environment calls the update method of the agents. The update code in SoarFollowerAgent gets the current position of the leader from the pointer and compares it with the agent's position. This distance between the follower and the leader is then sent to the Soar kernel and it is used by the Soar agent to decide to stop following or to continue to follow. Figure 11 shows the output from the Soar agent as seen from the Soar Debugger. Once the follower agent gets to a predefined distance from the leader, the follower agent stops and waits until that distance increases to continue following. Figure 12 shows a Soar seeking agent with one Soar follower agent. By creating more than one follower agent we can have a group of follower agents. When performing this test we can see how the group steering behaves by keeping the agents within the group a minimum distance from each other. Figure 13 shows a Soar seeking agents and a group of Soar follower agents.

```

432: O: 0432 (follow)
433: O: 0433 (follow)
434: O: 0434 (follow)
435: O: 0435 (stop)
436: O: 0436 (stop)
437: O: 0437 (stop)
438: O: 0438 (stop)
439: O: 0439 (stop)
440: O: 0440 (stop)
441: O: 0441 (stop)
442: O: 0442 (stop)
443: O: 0443 (stop)
444: O: 0444 (stop)
445: O: 0445 (stop)
446: O: 0446 (stop)
447: O: 0447 (follow)
448: O: 0448 (follow)
449: O: 0449 (follow)
450: O: 0450 (follow)

```

Figure 11. Soar Follower Agent output

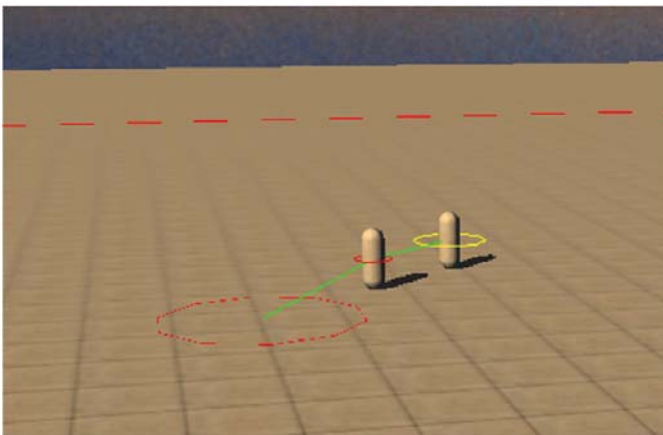


Figure 12. Leader agent with Soar Follower Agent

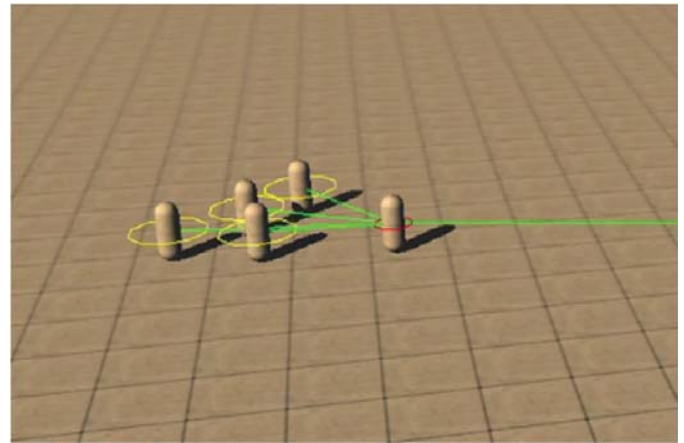


Figure 13. Group of Soar Follower Agents with leader

C. Waypoint Agent

The purpose of the waypoint agent is to go through a list of predetermined points. Waypoints are used to aid in navigation. This test replicates a multi-waypoint mission where an agent needs to travel a certain path. Similar to the previous Soar agents a new class had to be created, named SoarWaypointAgent. The Lua utility classes were updated to be able to initialize these types of agents from the Lua script. The Lua script in the my_sandbox project was updated so that a Soar waypoint agent is initialized. The Lua script agent file that acts as the helper for the Soar agent will have the list of points. This gives us the ability to change the amount of waypoints and modify the list without having to compile the project. The SoarWaypointAgent gets the position of the current target and the position of the agent and calculates the distance between them. This distance is sent to the Soar agent's input link as a working memory element. The Soar agent goes through its decision cycle and outputs "samewaypoint" to keep it current heading or "nextwaypoint" if the distance from the agent and its target reaches a certain value. The SoarWaypointAgent code will take the output from the Soar agent and communicate it to the Lua script. The Lua script uses this to update the agent's target to the new waypoint if needed. This agent was tested by itself in the sandbox and later it was tested by initializing the 3 different types of Soar agents that had been developed so far. The screen shot of the test with all agents can be seen in Figure 14.

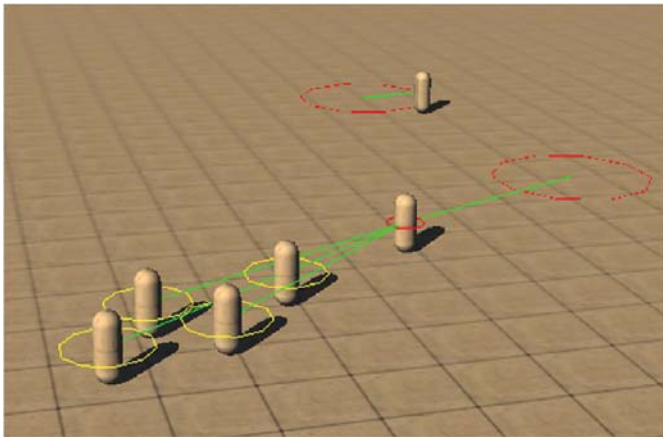


Figure 14. Soar agents in the sandbox, Waypoint Agent (top), Follower Agents (group of 4) and Seeking Agent (center right)

D. Soar and Lua Agents

Two tests were used to verify the ability to have Lua agents and Soar agents working in the same sandbox environment. The first test added a Lua agent as a follower agent in a group. This group consisted of the Lua agent and three Soar follower agents.

They were all following a Soar seeking agent. To create this test the sandbox Lua script was modified to initialize the Soar seeking agent and then to create the different types of follower agents. To be able to identify the Lua agent in the group the AgentUtilities Lua script was modified so that the Lua agent was visually recognizable. Figure 15 shows the result of the test. The Lua agent behaved as expected as part of the follower group. The second test added a Lua agent as the leader of the group. A script named Seeking-LuaAgent was created based on the agent in the Lua book [13]. Figure 16 shows a screenshot of the test where the Lua agent is the leader.

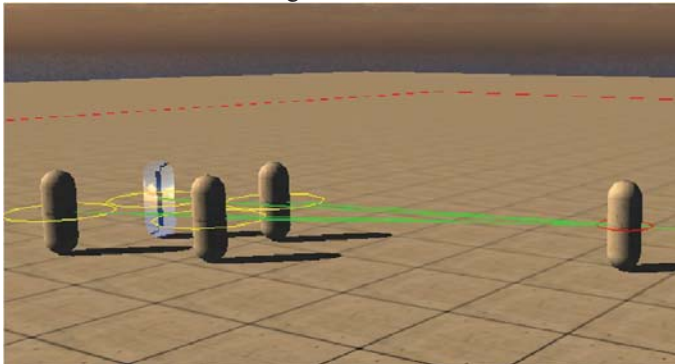


Figure 15. Screenshot Lua follower agent in group

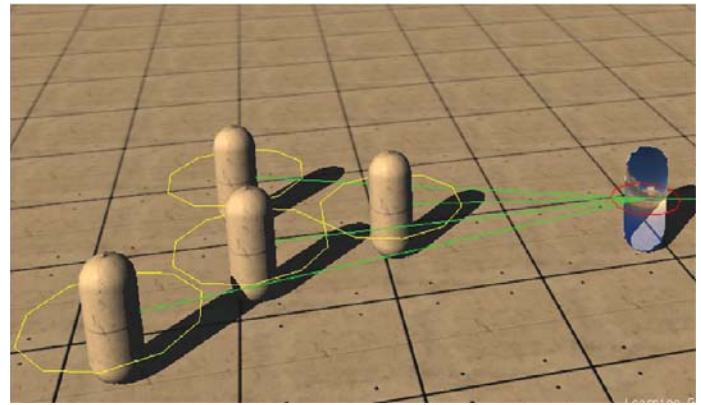


Figure 16. Screenshot Lua agent as leader

VI. CONCLUSION AND SIGNIFICANCE

This research successfully extended the sandbox, an open-source software product, and developed a learning environment for creating Soar agents. The results of the tests performed with the Soar agents show that it is possible to extend the sandbox environment and give it the capability of using Soar and Lua to control intelligent agents. The Soar cognitive architecture requires effort to learn and time to practice in order to become proficient. If your goal is to create simple intelligent agents or you have a limited time to develop your system, Soar may not be the right tool to use. However, if your system needs to integrate learning with problem solving, have parallel reasoning, or capable of complex intelligent agents Soar could be an appropriate tool.

By extending the sandbox environment and enabling it to use the Soar cognitive architecture to create intelligent agents, the Soar community and interested parties now have a customizable environment that is rich in features to learn and practice the Soar agent development process. This sandbox being similar to military simulations gives users chances to not only learn to create Soar agents, but also determine if Soar is the right choice for their modelling needs. This research also presents a blueprint on how to create Soar agents using the sandbox environment so that future research and more advanced agents can be developed.

The next step in the process of extending the environment would be to update the visualization of the current Soar agents. This will give the ability of the Soar agents to have the appearance of human beings. Additionally, the sandbox environment has features that were not able to be used by the agents that were tested. These features, like having agents in different teams and communicate with each other, would be beneficial in creating advanced agents that perform simulated military engagements. It is recommended to utilize the sandbox environment to build Soar agents that could later be used as intelligent entities in military simulations.

REFERENCES

- [1] Office of the Deputy Under Secretary of Defense for Acquisition and Technology, Systems and Software Engineering, Developmental Test and Evaluation (ODUSD(A&T)SSE/DTE).

- Modeling and Simulation Guidance for the Acquisition Workforce, Version 1.0. Washington, DC, 2008.
- [2] J. E. Summers. *Simulation-based Military Training: An Engineering Approach to Better Addressing Competing Environmental, Fiscal, and Security Concerns*. Washington Academy of Sciences, 2012.
 - [3] A. Heuvelink et al. "Intelligent Agent Supported Training in Virtual Simulations". TNO Defense, Security, and Safety. Soesterberg, Netherlands, 2009.
 - [4] Army and Marine Corps Training. *Better Performance and Cost Data Needed to More Fully Assess Simulation-Based Efforts*(GAO-13-698). United States Government Accountability Office. 2013.
 - [5] P. Langley et al. "Cognitive architectures: Research issues and challenges", *Cognitive Systems Research.*, 2008.
 - [6] J. E. Laird, *The Soar Cognitive Architecture*, The MIT Press, 2012.
 - [7] J. E. Laird et al. *Integrating Intelligent Computer Generated Forces in Distributed Simulations: TacAir-Soar in STOW-97*[Online].Available:
<http://ai.eecs.umich.edu/people/laird/papers/siw.html>
 - [8] J. E. Laird, "The Soar User's Manual Version 9.5.0". The Regents of the University of Michigan, 2015.
 - [9] J. E. Laird. "The Soar 9 Tutorial" (Updated for Soar 9.5.0), 2015.
 - [10] J. E. Laird et al. (2016, Jun. 7). *Soar Basics* [Online]. Available: <https://web.eecs.umich.edu/~soar/tutorial16/Tutorial-2016-SW-basic.pdf>
 - [11] "SML Quick Start Guide - Soar Cognitive Architecture", Soar.eecs.umich.edu, 2017. [Online]. Available: <http://soar.eecs.umich.edu/articles/articles/soar-markup-language-sml/78-sml-quick-start-guide>.
 - [12] D. Pearson. *XML Interface to Soar (SML) Software Specification*. ThreePen Software LLC., 2005.
 - [13] D. Young. *Learning Game AI Programming with Lua*. Packt Publishing Ltd., 2014.