

# Construct User Guide



Product Name: Voisus

Construct User Guide

© Copyright ASTi 2023

Restricted rights: copy and use of this document are subject to terms provided in ASTi's Software License Agreement ([www.asti-usa.com/license.html](http://www.asti-usa.com/license.html)).

ASTi  
500A Huntmar Park Drive  
Herndon, Virginia 20170 USA

## **Red Hat Enterprise Linux (RHEL) Subscriptions**

ASTi is an official Red Hat Embedded Partner. ASTi-provided products based on RHEL include Red Hat software integrated with ASTi's installation. ASTi includes a Red Hat subscription with every purchase of our Software and Information Assurance (SW/IA) maintenance products. Systems with active maintenance receive Red Hat software updates and support directly from ASTi.

## **Export Restriction**

Countries other than the United States may restrict the import, use, or export of software that contains encryption technology. By installing this software, you agree that you shall be solely responsible for compliance with any such import, use, or export restrictions. For full details on Red Hat export restrictions, go to the following:

[www.redhat.com/en/about/export-control-product-matrix](http://www.redhat.com/en/about/export-control-product-matrix)



## Revision history

Date	Revision	Version	Comments
7/26/2017	B	0	Converted Construct content to PDF format; edited content for grammar, style, and accuracy.
1/12/2018	B	1	Fixed broken hyperlinks to Construct API.
3/8/2018	B	2	(5.33) Fixed variable display errors throughout document and modified formatting styles.
6/13/2018	B	3	(5.34) Removed VBS2 reference in "Add an interaction." Made changes to syntax and grammar per the style guide.
9/28/2018	B	4	(7.0.1) Removed <b>Add Voices</b> reference in "Add an entity." Updated screenshot.
9/18/2020	C	0	(7.8.0) Updated Voibus web interface screenshots.
8/10/2021	C	1	(7.11.1) Updated screenshots of all table header styles.
3/8/2023	C	2	(8.0.0) Added the Red Hat Enterprise Linux subscription and export statements to the front matter.



# Contents

<b>1.0 Introduction</b>	<b>1</b>
1.1 Speech licenses	2
<b>2.0 Scenario Management</b>	<b>4</b>
2.1 Add a scenario	4
<b>3.0 Comm Plan</b>	<b>6</b>
<b>4.0 Entities</b>	<b>7</b>
4.1 Add an entity	8
4.2 Monitor entities	12
4.3 Entity action	13
<b>5.0 Interactions</b>	<b>14</b>
5.1 Add an interaction	15
<b>6.0 Sounds</b>	<b>17</b>
6.1 Add a sound	18
<b>7.0 Language models</b>	<b>21</b>
7.1 Enable speech recognition	22
7.2 Grammar syntax	24
<b>8.0 Radio Effects</b>	<b>26</b>
<b>9.0 Construct Settings</b>	<b>27</b>
<b>10.0 AIML</b>	<b>29</b>
10.1 AIML tags	31
10.2 AIML Construct integration	31
10.3 AIML responses and meaning values	32
10.4 Create and map AIML definition to entity	33
<b>11.0 Behaviors</b>	<b>35</b>
11.1 Create a new behavior	35
11.2 Map new behavior to entity	36



11.3 Behavior execution .....	37
11.4 Behavior node types .....	38
11.4.1 Composite nodes .....	39
11.4.2 Repeat nodes .....	39
11.4.3 Action nodes .....	40
11.4.4 Utility nodes .....	41
11.5 Behavior expressions .....	41

# 1.0 Introduction

Construct is a constructive simulation toolkit that adds synthetic communications to training systems such as flight simulators, Serious Games, and command and control environments. Construct adds voice and radio capabilities to simulated entities and game avatars so that trainees can interact with them verbally. Entities are augmented with a simulated radio that transmits and receives on a specified communications net. Construct also enables face-to-face communication in 3D game environments between avatars and human players. Construct's resources are set up using the Voisus web interface or via programming with its HTTP Application Programming Interface (API).

Construct contains the following features:

- Simulated radios (Distributed Interactive Simulation (DIS) / high-level architecture (HLA))
- Text-to-speech (TTS)
- Automated speech recognition (ASR)
- Sound file playback
- Face-to-face communications in 3D environments
- HTTP API for realtime control and status reporting
- Web interface for remote viewing and configuration
- Integrated with Voisus scenarios and Comm Plans
- DIS entity attach
- Entity behavior modeling tools
- Natural language processing
- Radio effects and background sound layering

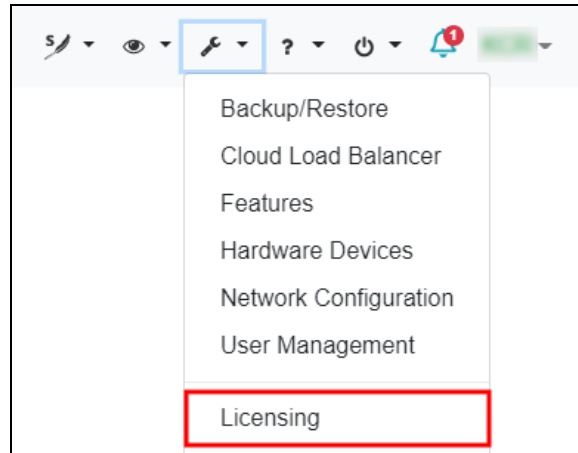
Each application may use a different combination of these features. Applications needing only simple background radio chatter, for example, may use the Voisus web interface to quickly script and then trigger radio transmissions on demand. Higher fidelity and more automated training systems may call for dozens of intelligent voice agents that listen, think, and respond to voice messages from human players using a combination of ASR, automated behaviors, and TTS.

Construct is used both standalone and as part of large, integrated training systems. When integrated into a larger training system, Construct's HTTP API and the DIS network are the two mechanisms for inter-operation. For example, a simulation host computer can issue HTTP requests to Construct in order to trigger TTS transmissions from simulation entities, which are then heard by all human trainees with in-tune radios in the exercise.

## 1.1 Speech licenses

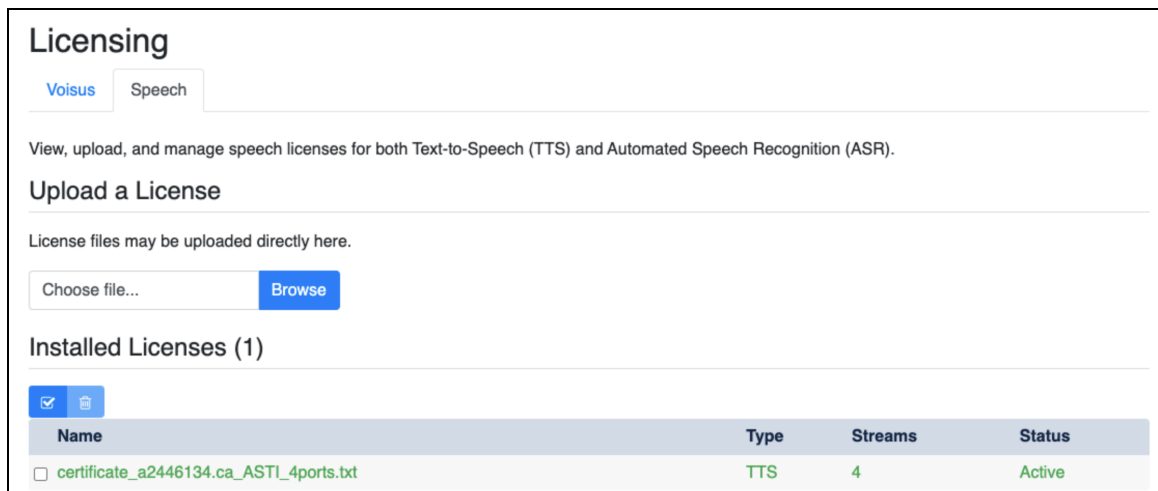
If your application requires text-to-speech (TTS) or automatic speech recognition (ASR), one or more speech licenses must be installed on the Voisus server. To confirm the expected number of speech streams are enabled, follow these steps:

1. From the top right, go to **Manage** (  ) > **Licensing**.



*Figure 1: Licensing navigation*

2. Go to **Speech**.
3. Under **Upload a License**, select **Browse**, and find the speech license file on your local system.



*Figure 2: Licensing*

The number of TTS streams determines the number of speech events that can be processed simultaneously. For example, if three entities need to talk at the same time, then you need three TTS streams.

Licenses are included in system backups via **Backup & Restore**.

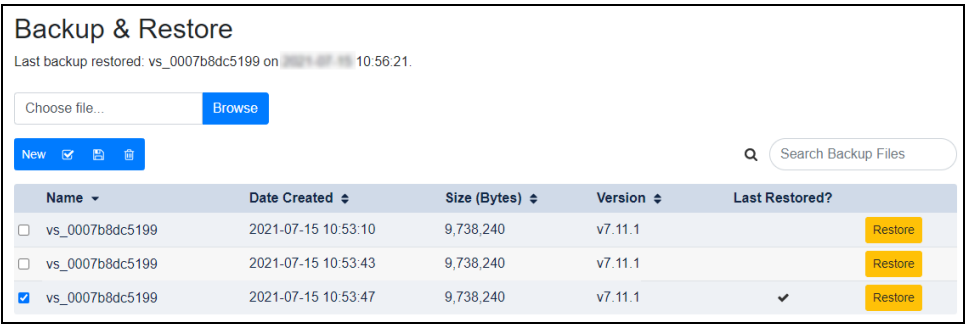



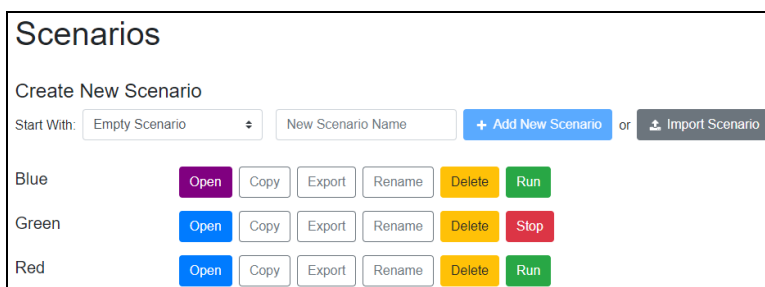
Figure 3: Backup & Restore

## 2.0 Scenario Management

A scenario contains all the information needed for a specific training task or simulation, such as entities, interactions, behaviors, sounds, and a Comm Plan. You can add these resources to a scenario via the Voisus web interface or the Hypertext Transfer Protocol (HTTP) Application Programming Interface (API). This section describes the management of scenarios in the Voisus web interface.

Access Construct's scenario management by selecting **Construct** (  ) on the home page and then selecting **Scenarios**.

On **Scenarios**, add, copy, delete, rename, and run scenarios as needed.



*Figure 4: Scenarios*

Without a scenario running, the Voisus server is almost completely idle. Running a scenario brings to life the sounds, speech, and communications stored within. Scenarios can be edited on the fly and changes take effect immediately. When the Voisus server reboots, the scenario running at shutdown also restarts.

Voisus servers may hold an unlimited number of scenarios and can run one scenario at a time. These scenarios can be created, viewed, modified, and run by all system users logged on to that particular server.

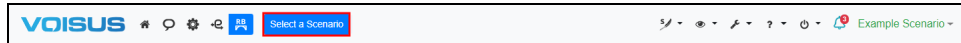
### 2.1 Add a scenario

Before Voisus clients can communicate with each other, you must build and run a new scenario in the Voisus web interface. To build and run a scenario, follow these steps:

1. Open a web browser on a computer or tablet sharing a network with the Voisus server.
2. In the address bar, enter the Voisus server's IP address.
3. Log into the Voisus web interface using the following default credentials:

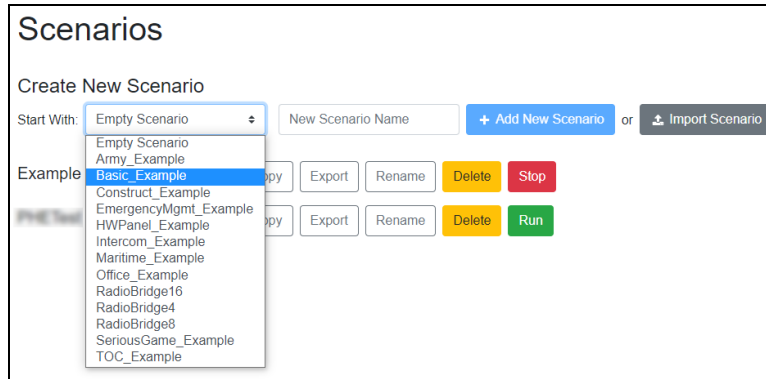
Username	Password
admin	astirules

4. From the top navigation bar, go to **Select a Scenario**.



*Figure 5: Select a Scenario*

5. Under **Create New Scenario**, select **Start with**, and choose a scenario template.



*Figure 6: New scenarios*

The example scenarios come with a Comm Plan, roles, and radios that you can modify as needed.

6. In **New Scenario Name**, enter a unique name for the scenario.
7. Select **+ Add New Scenario**.
8. Select **Run** to run the scenario and **Open** to view/edit its resources. You can dynamically edit scenarios while they are running.

## 3.0 Comm Plan

The Comm Plan is a collection of virtual nets that represent communications channels or frequencies for radios or intercoms. Entities may have a radio with a net. The net determines which entities and human players can communicate.

- Nets define the frequency, modulation type, bandwidth, and crypto settings for the radios.
- At a minimum, nets require a valid frequency and waveform to work properly.
- Only entities and Voisus clients with matching net settings are able to communicate.
- If using speech recognition, ensure the transmitting radio is using a high quality waveform, otherwise recognition accuracy will be reduced. Use PCM encoding and a 16 KHz sampling rate for best results. In some situations, this may require changing the radio settings on a remote system.



The screenshot shows the 'Comm Plan' interface with tabs for 'Nets', 'Netgroups', 'Waveforms', 'Cryptos', 'Freqhops', and 'Satcoms'. The 'Nets' tab is active, displaying a table with columns: Name, Description, Frequency (Hz), Waveform, Crypto, Freqhop, Satcom, and Netgroups. Two nets are listed: AM\_Net1 and AM\_Net2, both with AM waveforms and frequencies of 200,000,000 Hz and 202,000,000 Hz respectively. Both have Crypto, Freqhop, and Satcom settings set to 'Off'. The Netgroups column shows 'AM Group + 1 more' for both.

Name	Description	Frequency (Hz)	Waveform	Crypto	Freqhop	Satcom	Netgroups
AM_Net1	AM Radio Net1	200,000,000	AM	Off	Off	Off	AM Group + 1 more
AM_Net2	AM Radio Net2	202,000,000	AM	Off	Off	Off	AM Group + 1 more

*Figure 7: Construct Comm Plan*

## 4.0 Entities

Communications modeling in Construct centers on simulation entities that represent aircraft, ground vehicles, or other agents in the environment. Each entity is given a voice and radio to enable communication with each other and with human players. Hundreds of entities can be modeled by a single Construct instance to support large simulation events with a small footprint. Construct entities may be created, modified, and deleted on demand using both the web interface and the Hypertext Transfer Protocol (HTTP) Application Programming Interface (API).

The Voisus web interface is the most commonly used approach during initial setup and for simple applications. Using a web browser, an administrator or scenario designer can quickly create a handful of entity definitions with customized voice and radio settings. These entities definitions are stored in the scenario, and the corresponding run-time entity instances are created when the scenario is installed and destroyed when the scenario is uninstalled. Configuration changes, such as changing an entity's communication net, take effect immediately while the scenario is running.

The HTTP API, on the other hand, supports low-level, high-performance, and dynamic control of entities. Any client of the API is able to program entity definitions in the scenario and interact directly with the run-time entity instances. The HTTP API is also the primary means to integrate Construct with external simulator systems. For more information about the HTTP API, go to [Construct API](#).

Construct entities follow standard radio protocol to avoid ‘stepping on’ radio transmissions from other entities or human players on the network. When an entity plans to speak, it will wait for the communication net to be idle before transmitting.

Although entities have many attributes, many are optional, and only a few are needed for an entity to talk on a radio. At a minimum, the entity will need a domain, net, and text-to-speech (TTS) voice set in order for it to communicate. The domain and net determine which Distributed Interactive Simulation (DIS) exercise and radio frequency the entity will communicate on, respectively. Entity attributes are editable both on the **Entity** web page and using the HTTP API.



## 4.1 Add an entity

To add an entity, follow these steps:

1. From the top-left navigation bar, go to **Construct** (⚙️) > **Entities**.

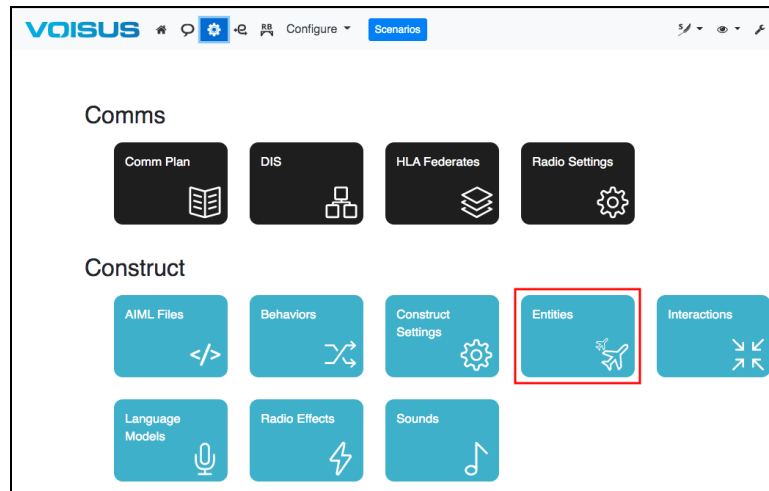


Figure 8: Entities navigation

2. To create a new entity, under **Entities**, select **Create Entity** (+).
3. On **General**, in **Name**, enter a name.
4. To set the entity's Distributed Interactive Simulation (DIS) domain, under **Comms**, select **Domain**, and choose a domain. To create a domain, select **Edit Domains** (⚙️).
5. To set the entity's virtual net, select **Net**, and choose a net. This list displays all nets available in the scenario's Comm Plan. To add a net, select **Edit Commplan** (⚙️).

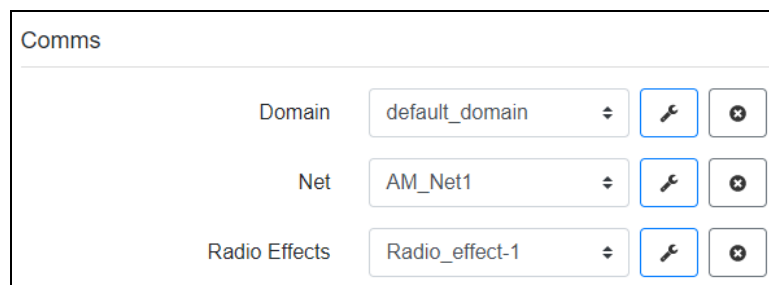


Figure 9: Entity Comms

6. (Optional) To select an audio effect, select **Radio Effects**, and choose an effect. When applied, these effects degrade the audio quality for increased realism.

7. *(Optional)* If the entity uses propagation for radio ranging, under **Position**, in **Marking**, enter a marking string. These coordinates associate the entity with a DIS entity on the network. For Virtual Battlespace (VBS), enter the Uniform Resource Name (URN) marking field of the object you wish to attach to.

The screenshot shows a configuration panel titled "Position". Inside, there is a "Marking" label followed by a text input field containing "marking". Below this, there are three rows for coordinates: "X" with a value of "1", "Y" with a value of "2", and "Z" with a value of "3". Each coordinate value is in a light green input field.

*Figure 10: Entity Position*

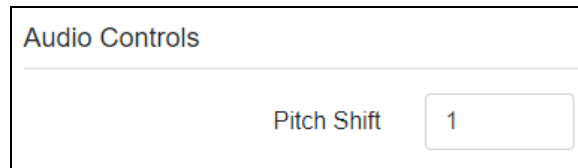
8. To edit synthetic speech and voices, go to **Voice**.
9. Under **Text-to-Speech**, select **Voice**, and choose the entity's voice.
10. To adjust the TTS voice's pace, select **Rate**, and then choose a pace. Values range from 1–9, where 1 is slowest, and 9 is fastest.
11. To adjust the TTS voice's volume level, select **Volume**, and choose a volume. Values range from 1–9, where 1 is the quietest, and 9 is the loudest.

The screenshot shows a configuration panel titled "9 Line - Red Horse 11". It has three tabs: "General", "Voice", and "Advanced". The "Voice" tab is selected. Under the "Text-to-Speech" section, there are three settings: "Voice" is a dropdown menu showing "Kendra" with a plus icon to its right; "Rate" is a dropdown menu showing "5"; and "Volume" is a dropdown menu showing "6".

*Figure 11: Entity Text-to-Speech*




12. To raise or lower the voice's pitch, under **Audio Controls**, in **Pitch Shift**, specify a value. You can adjust the values by 0.1 at a time. The adjustment range is 0.8–1.2. The default value of 1.0 means no pitch shifting occurs.

This setting expands the number of distinct voices available. Different entities can use the same pitch-shifted TTS voice to appear as though it is two different voices.

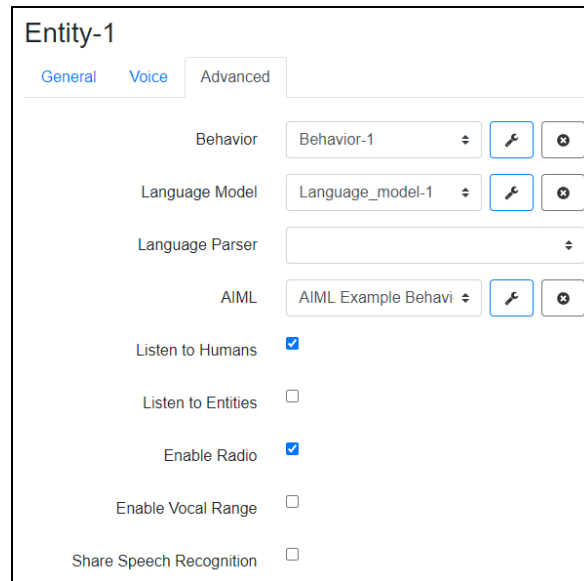


The screenshot shows a window titled "Audio Controls". Inside the window, there is a label "Pitch Shift" followed by a numeric input field containing the value "1".

*Figure 12: Entity Audio Controls*

13. To assign an intelligent behavior to this entity, go to the **Advanced** tab, select the **Behavior** list, and choose a behavior. To edit or add behaviors, select **Edit Behaviors** ()
  14. To assign a speech recognition language model to this entity, select the **Language Model** list, and choose a model. To edit or add models, select **Edit Language Model** ()
  15. To assign a language parser to the entity, select the **Language Parser** list, and choose a parser. Language parsers extract call signs, waypoints, and other variables from recognized speech. ASTi creates language parsers and installs them as plug-ins.
  16. To assign an Artificial Intelligence Markup Language (AIML) definition, select the **AIML** list, and choose a definition. To edit or add definitions, select **Edit AIML** ()
- These definitions allow you to create automated, reactive behaviors and simple natural language-processing solutions. Much like language models, AIML definitions allow entities to automatically respond to verbal commands.
17. If the entity should actively listen and recognize audio from other human trainees, select **Listen to Humans**.
  18. If the entity should actively listen and recognize audio from other entities, select **Listen to Entities**.
  19. To enable radio commands, select **Enable Radio**. For the radio to operate, ensure a net and domain are set in Steps 4 and 5.

20. To enable face-to-face communications, select **Enable Vocal Range**. This setting is useful when the entity has an avatar in a 3D game environment.

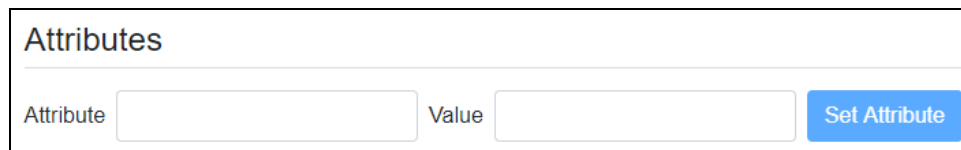


The image shows the 'Entity-1' settings window with the 'Advanced' tab selected. The settings are as follows:

Setting	Value	Icon
Behavior	Behavior-1	⚙️
Language Model	Language_model-1	⚙️
Language Parser		⌵
AIML	AIML Example Behavi	⚙️
Listen to Humans	<input checked="" type="checkbox"/>	
Listen to Entities	<input type="checkbox"/>	
Enable Radio	<input checked="" type="checkbox"/>	
Enable Vocal Range	<input type="checkbox"/>	
Share Speech Recognition	<input type="checkbox"/>	

*Figure 13: Entity Advanced settings*

21. You can add JavaScript Object Notation (JSON)-formatted attributes to the entity for behaviors and speech text with parameters. In **Attribute**, enter an attribute value (e.g., "altitude"). Enter a value in **Value**, and select **Set Attribute**.

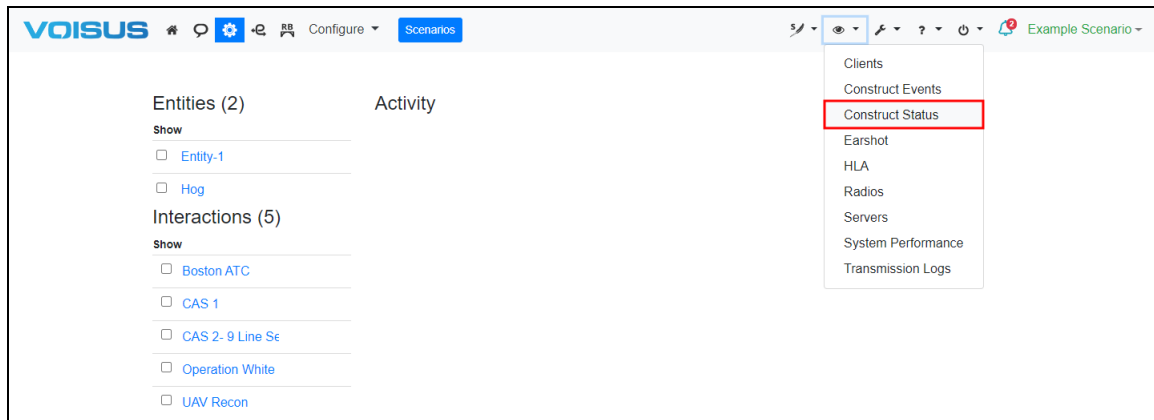


The image shows the 'Attributes' section of the interface. It contains two input fields labeled 'Attribute' and 'Value', and a blue button labeled 'Set Attribute'.

*Figure 14: Entity Attributes*

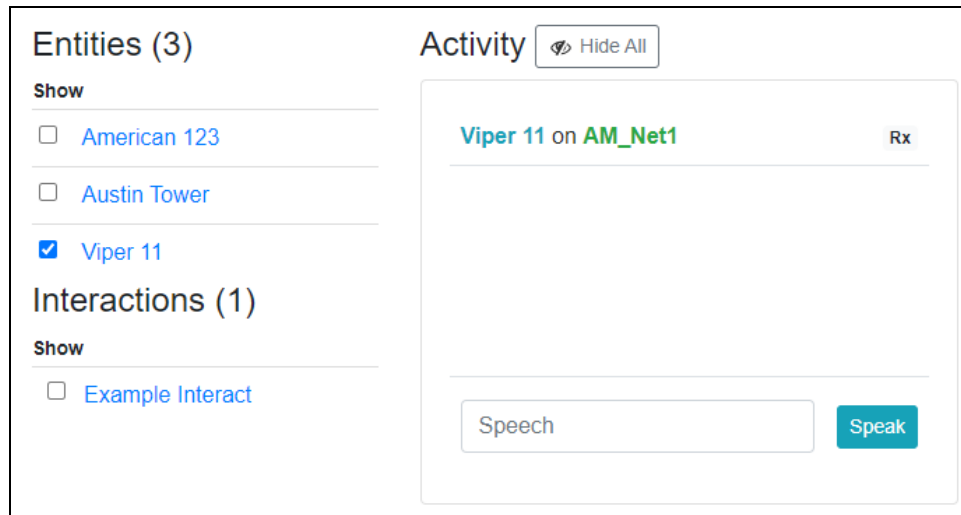
## 4.2 Monitor entities

For a realtime view of the running entities, go to **Construct Status**.



*Figure 15: Construct Status*

Here you can monitor all entities running on the server, including entities created in the Voibus web interface and those created using the run-time entity Application Programming Interface (API). The status page also features a text field to input and trigger speech from an entity on the fly. Each time an entity receives or transmits, the activity will be displayed on the web page. This run-time status information is also accessible via the Hypertext Transfer Protocol (HTTP) API.



*Figure 16: Monitor entities*

## 4.3 Entity action

Entities are spurred into action in one of several ways:

- Interactions script a fixed sequence of radio calls between one or more entities. For more information about interactions, go to Section 5.0, "Interactions" on the next page.
- Behaviors define entity decision making and voice responses. for more information about behaviors, go to Section 11.0, "Behaviors" on page 35.
- The Hypertext Transfer Protocol (HTTP) Application Programming Interface (API) allows you to use a programming language of your choice to remotely trigger entity behaviors and speech. For more information about the HTTP API, go to [Construct API](#).

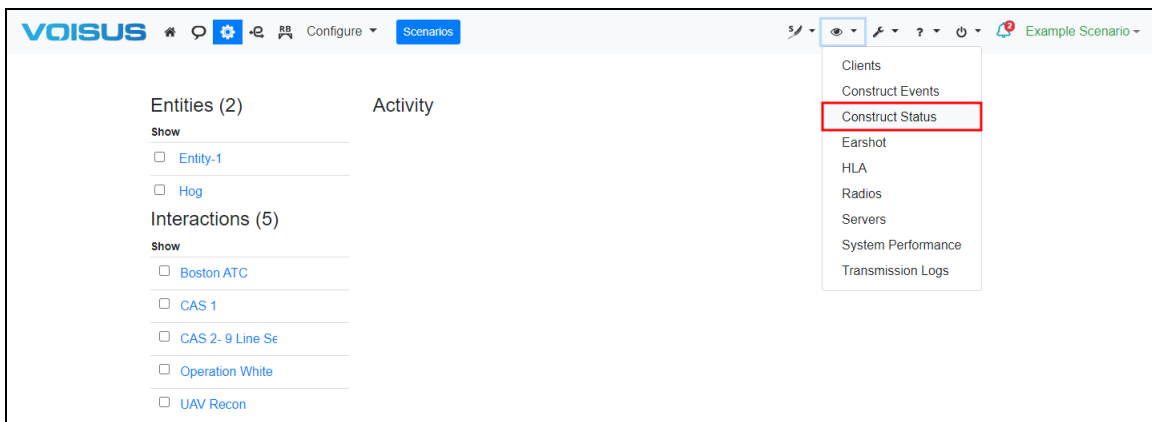
If you are unsure of which route to take, start with interactions and go from there. Construct also works with Discovery Machine Behavior Modeling Console and other third party software using the HTTP API.

## 5.0 Interactions

Interactions prompt entities to take action and speak. The interaction can be a one-way broadcast, or it can describe a back-and-forth conversation between multiple entities. Interactions are useful when creating an ongoing backdrop of radio chatter on a radio frequency. Alternatively, trigger interactions at a specified start time or based on other conditions, like a character's position in a 3D game environment. Each interaction lists a fixed sequence of speech events that occur, separated by a specified amount of time.

To create a short exchange in Construct, create two entities and one interaction. The interaction should contain two actions, one for each radio transmission. Interactions are relatively simple and limited in possibilities. If you wish to recreate more sophisticated reactions and decision making, use behaviors, Artificial Intelligence Markup Language, or the Hypertext Transfer Protocol (HTTP) Application Programming Interface (API).

As with entities, interactions are shown on **Construct Status** along with any activity resulting from the running interactions.



*Figure 17: Construct Status page*

## 5.1 Add an interaction

To add an interaction, follow these steps:

1. From the top-left navigation bar, go to **Construct** (⚙️) > **Interactions**.

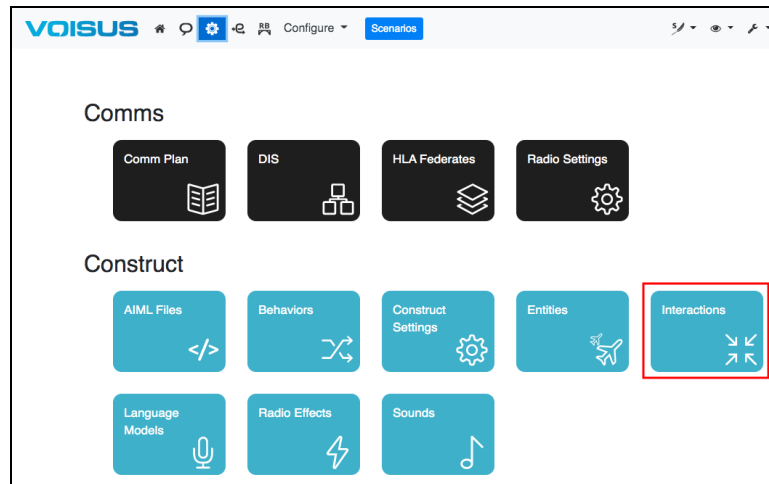


Figure 18: Interactions navigation

2. Under **Interactions**, select **Create Interaction** (+).
3. On **General**, in **Name**, enter a name for the interaction.
4. To enable the interaction, select **Enabled**.
5. Under **Start Time**, select **Type**, and choose one of the following:
  - **Relative**: time is specified relative to the start of the scenario.
  - **UTC**: time is based on the system clock; a relative time of 0 means start executing immediately.
  - **VBS Trigger**: starts the interaction based on a Virtual Battlespace (VBS) scripting command.
6. In **Time**, enter a time offset in seconds. This value determines when the interaction starts execution. The meaning of this value depends on the **Type**.

 A screenshot of a 'Start Time' configuration dialog box. It has a title bar 'Start Time'. Inside, there are two fields. The first field is labeled 'Type' and has a dropdown menu currently showing 'Relative'. The second field is labeled 'Time' and has a text input box containing the number '5'.

Figure 19: Interaction Start Time

7. If the interaction should play on a loop, under **Loop**, select **Loop Forever**.



8. If **Loop Forever** is cleared, in **Play Count**, enter the number of times the interaction will loop.
9. In **Delay**, enter the delay time in seconds between loops.

**Loop**

Loop Forever

☐

Play Count

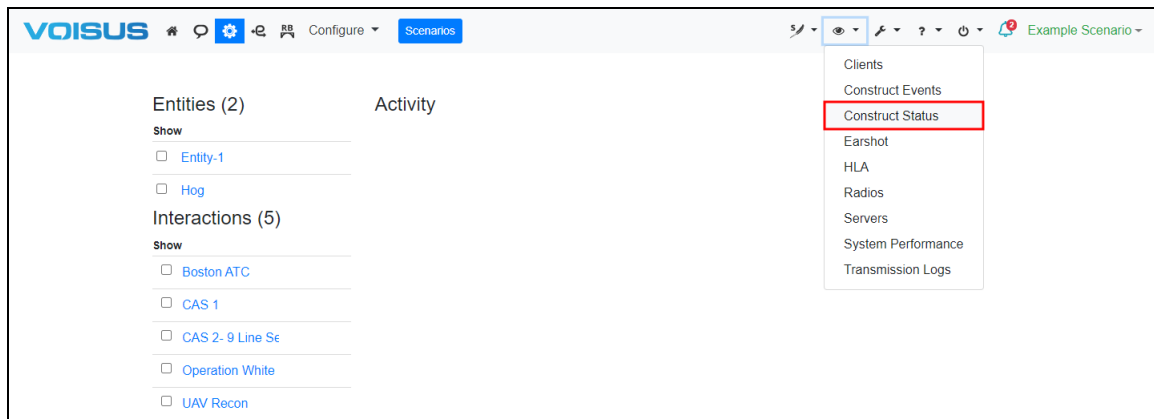
Delay

*Figure 20: Interaction Loop*

## 6.0 Sounds

Sounds define segments of a sound file that Construct uses. Sound settings including gain, an optional transcript, and loop parameters. Sounds reference sound files containing either recorded speech or effects like gunfire or cockpit noise. Once a sound is defined in a scenario, it can be referenced by multiple interactions and behaviors to be used for entity speech or sound effects.

If you have speech recordings you would like to replay on the network as radio chatter, create a sound for each unique radio transmission. If you are using an interaction to sequence the radio chatter, add an action for each radio transmission, selecting the corresponding sound in the action list. Running the interaction will now replay your recordings onto the Distributed Interactive Simulation (DIS) network. Sounds used in this way are an alternative to using text-to-speech (TTS) to synthesize speech. For sounds containing speech, it is useful to fill **Transcript** so that the corresponding text displays on **Construct Status** when the entity speaks.



*Figure 21: Construct Status*



**Note:** Sounds are distinct from the sound files (.wav) they reference. Sound files are uploaded and managed on **Sound Files**, which contains a library of default sound files. Sound files can be referenced by sounds in multiple scenarios if desired.

When a sound is created and its sound file is selected, the entire sound file will be used by default. The **Offset** and **Length** parameters identify a subsection of the sound file to use instead. This is useful if there is too much silence at the beginning or end of the file. Editing sound parameters takes effect immediately.

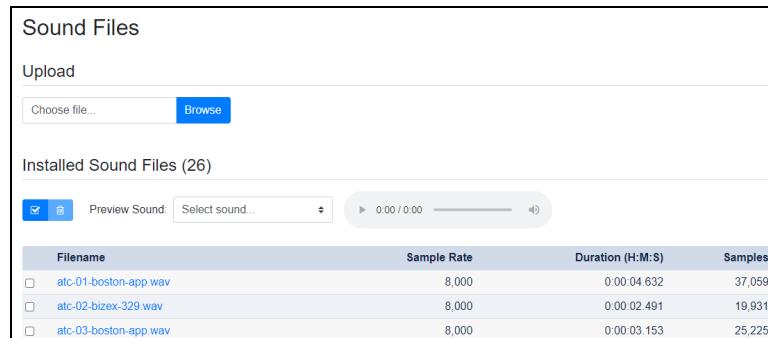


Figure 22: Sound Files page

## 6.1 Add a sound

To add a sound, follow these steps:

1. From the top-left navigation bar, go to **Construct** (⚙️) > **Sounds**.

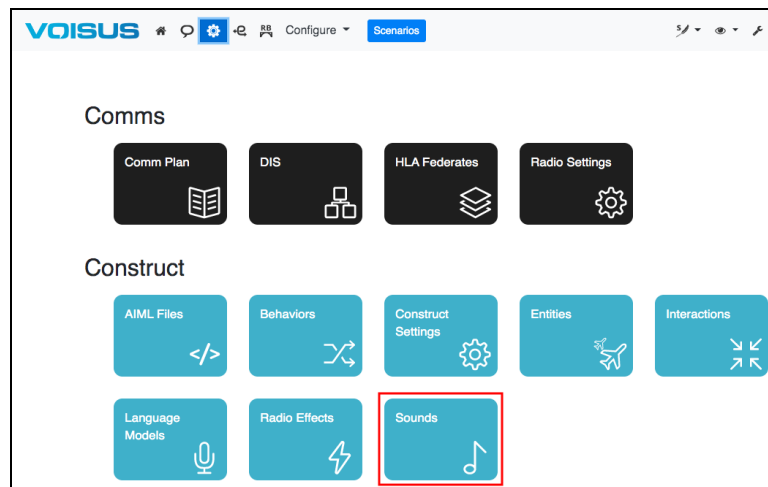


Figure 23: Sounds navigation

2. Under **Sounds**, select **Create Sound** (+).
3. In **Name**, enter a unique sound name.
4. In **Gain**, enter a value representing the gain applied to the sound during playback. A value of 1, which is the default, results in no volume change; a value of 0 silences the sound.

5. Under **Sound File**, select **File**, and choose a sound file. To upload new sound files, select **Edit Sound Files** (🔧).
6. In **Offset**, enter the location in the sound file to start playback. The default value of 0 means playback starts at the beginning, whereas a value of 16,000 means playback starts 16,000 samples into the file.
7. In **Length**, enter the length in samples of the section to play. The default value of 0 means play to the end of the file.
8. *(Optional)* In **Transcript**, enter a transcript for a sound containing speech. When a Construct entity speaks the sound, the associated transcript displays on the status page and in the Construct event history.

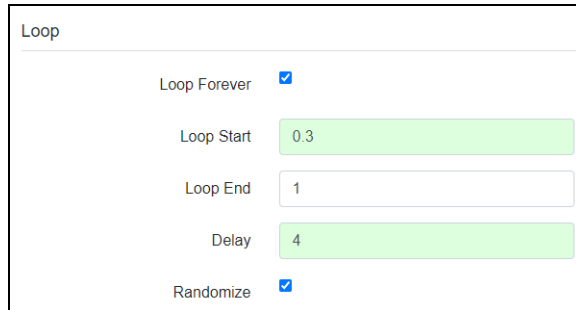
The screenshot shows a settings panel titled "Example Sound". It contains the following fields and controls:

- Name:** A text field with the value "Example Sound".
- Gain:** A text field with the value "1".
- Sound File:** A section header.
- File:** A dropdown menu showing "helo-cockpit-1.wav", with a wrench icon (Edit Sound Files) and a plus icon (Add File) to its right.
- File size:** A text field showing "27000 samples".
- Offset:** A text field with the value "3".
- Length:** A text field with the value "3".
- Sound size:** A text field showing "Sound size: 3 samples (0.00s)".
- Transcript:** A large text area with the placeholder text "Enter transcript here."

*Figure 24: Sound File settings*

9. If this sound should repeat indefinitely, under **Loop**, select **Loop Forever**.
10. In **Loop Start**, enter the starting position of the loop within the sound. Decimal values range from 0–1 inclusively.
11. In **Loop End**, enter the ending position of the loop within the sound. Decimal values range from 0–1 inclusively.
12. In **Delay**, enter the delay value in seconds between sound loops.

13. If playback should start at a random location in the sound, select **Randomize**. Use this setting to add variation to the sound environment.



Loop	
Loop Forever	<input checked="" type="checkbox"/>
Loop Start	<input type="text" value="0.3"/>
Loop End	<input type="text" value="1"/>
Delay	<input type="text" value="4"/>
Randomize	<input checked="" type="checkbox"/>

*Figure 25: Sound loops*

## 7.0 Language models

Language models define the phraseology to be recognized and transcribed by the Construct Automatic Speech Recognition (ASR) system. Construct supports two types of speech recognition language models:

- Statistical Language Models (SLMs)
- Grammars

In Construct, each entity has an optional language model selection, which should be filled in if that entity should listen and respond to human speech. The type of speech the entity is expected to encounter should determine the settings for its language model. For example, if the entity should listen and respond to conversational English, a general English SLM should be used.

The grammar approach describes a strict syntax for the speech to be recognized using Backus Naur Form (BNF). Grammars are best suited for small domains with very constrained phraseology. Conversely, the more complex and variable the phraseology, the more likely it is that the statistical approach is the better choice.

With the SLM approach, the model is trained on thousands of transcriptions from the application domain that help determine the most likely sequence of words for each new utterance. This approach is accepting of new word combinations that are more suitable for large vocabulary tasks. SLMs are highly accurate when the training data is a good match to the real data.

ASTi has a number of models available if the Construct ASR package is enabled. To obtain a model for your project, contact ASTi. To access speech recognition events from other systems on the network, go to [Construct API](#).

## 7.1 Enable speech recognition

To enable speech recognition, follow these steps:

1. From the top-left navigation menu, go to **Construct** (⚙️) > **Language Models**.

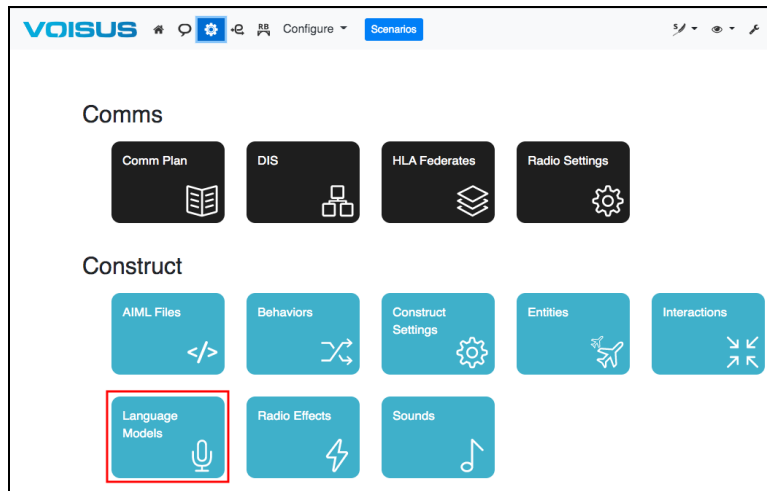


Figure 26: Language Models navigation

2. To add a new model, under **Language Models**, select **Create Language Model** (+).
3. In **Name**, enter a name for the language model.
4. Select **Speech Model**, and choose a grammar or Sound Language Model (SLM). ASTi recommends starting with **package-16K-EN-120210**, a general English SLM that recognizes conversational English spoken with an American accent.
5. If multiple entities share the language model, select **Shared**. This setting is recommended for SLMs due to their consumption of notable amounts of system random-access memory (RAM). Depending on the system's amount of RAM, multiple entities using non-shared language models could starve the system, causing audio breakup or other issues.

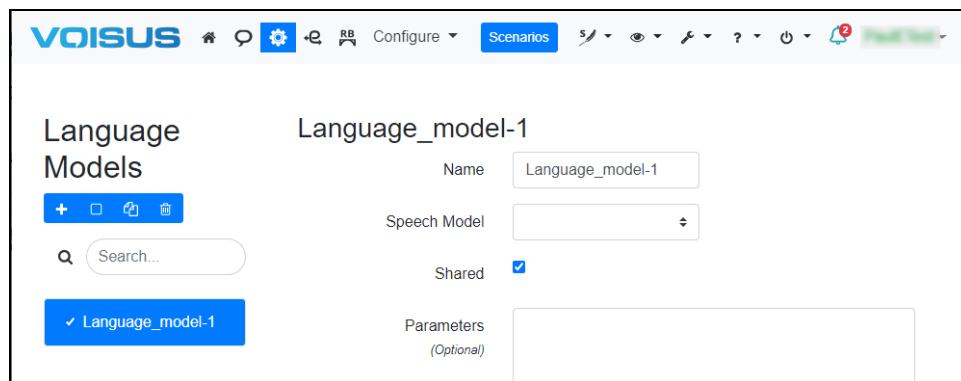
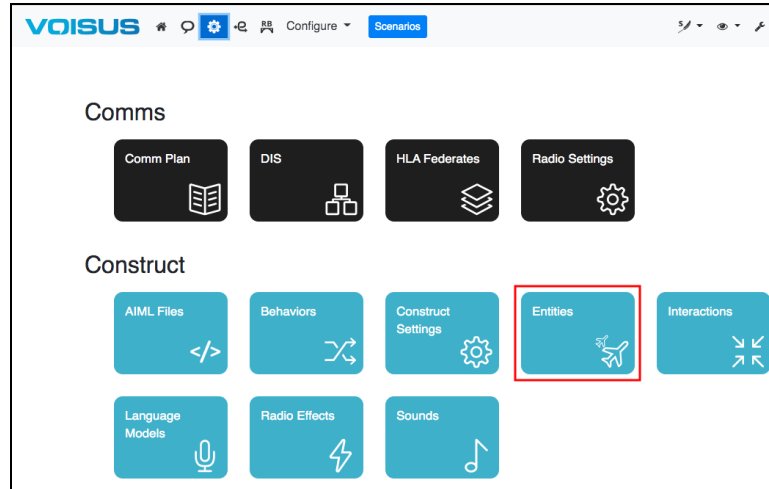


Figure 27: Language Models page

6. From the top-left navigation bar, go to **Construct** (⚙️) > **Entities**.



*Figure 28: Entities navigation*

7. On **Entities**, select or create the entity that will use speech recognition. For more information about entities, go to Section 4.1, "Add an entity" on page 8.
8. Go to **Advanced**.
9. Select **Language Model**, and then choose the language model you created in Steps 2–5.
10. Select **Listen to Entities**.



The entity automatically transcribes any received audio, producing a recognition event. The recognition text displays on **Construct Status** and **Construct Events**.

Construct Events						
Show: 100		Filter Events	Date: mm/dd/yyyy			
<input checked="" type="checkbox"/> Capture Events <input checked="" type="checkbox"/> Capture Trainee Audio <input checked="" type="checkbox"/> Capture Entity Audio						
Scenario: Construct						
	Time	Entity	Net	Event	Transcript	Meaning
☆	Aug 5, 4:26:22 PM	ATC - Bizex 329	ATC	TTS	<ATC-02-Bizex329>	Ⓜ Ⓛ
☆	Aug 5, 4:26:16 PM	ATC - Boston Approach	ATC	TTS	<ATC-01-Boston-App>	Ⓜ Ⓛ
☆	Aug 5, 4:26:16 PM	9 Line - Hog 21	CAS 2	TTS	<Hog 21 TX2>	Ⓜ Ⓛ
☆	Aug 5, 4:26:11 PM	9 Line - Red Horse 11	CAS 2	TTS	<Red Horse 11 TX2>	Ⓜ Ⓛ
☆	Aug 5, 4:26:07 PM	9 Line - Hog 21	CAS 2	TTS	<Hog 21 TX1>	Ⓜ Ⓛ
☆	Aug 5, 4:26:01 PM	9 Line - Red Horse 11	CAS 2	TTS	<Red Horse 11 TX1>	Ⓜ Ⓛ
☆	Aug 5, 4:25:59 PM	ATC - Cessna 01C	ATC	TTS	<ATC-04-Cessna01C>	Ⓜ Ⓛ

*Figure 29: Construct Events*

Entities do not listen to each other speak; they only listen to and perform speech recognition on audio transmitted by humans. An entity's response is determined by the entity's behavior, selection status, or the simulation host computer if it is listening via the Hypertext Transfer Protocol (HTTP) application program interface (API).

## 7.2 Grammar syntax

A grammar text area is displayed when an acoustic model is selected, allowing you to perform minor tasks for rote speech recognition. Edit the grammar in place on the web page, or copy and paste the grammar contents from a file on your computer. A simple grammar to recognize one or more digits is as follows:

```
<Digit> = (zero | one | two | three | four | five | six | seven |
eight | nine);
public <TopLevel> = <Digit>+;
```

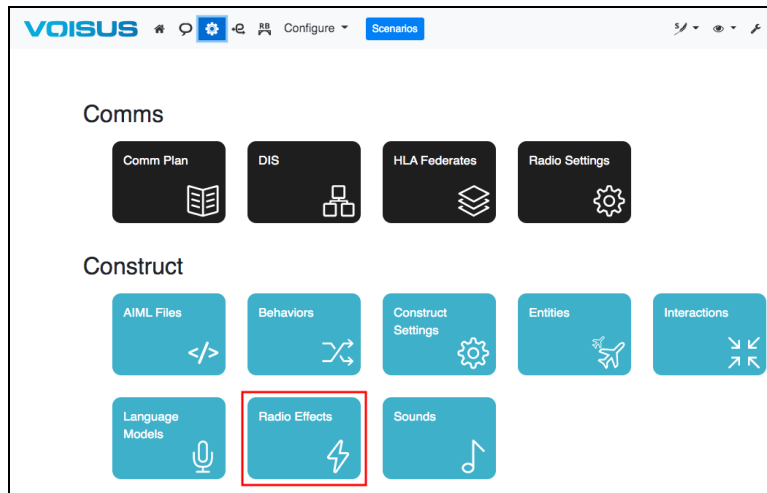
Table 1, "Grammar syntax meaning" below shows Construct grammar syntax and its meaning:

Grammar Syntax	Meaning
<Rule> = expression;	Define a new grammar rule or non-terminal
hi   hello	Choose between multiple options
( )	Parentheses create groups of items
+	Match the preceding expression one or more times
*	Match the preceding expression zero or more times
[]	Brackets make the expression within optional

*Table 1: Grammar syntax meaning*

## 8.0 Radio Effects

Radio effects make entity radio transmissions sound more realistic using added distortion, noise, and other filtering effects. You can also number radio effects to a scenario, but each entity can only use one effect at a time. A wide range of sounds is possible (e.g., "clean," "gritty," "noisy") all by adjusting the handful of settings in the radio effect definition. Multiple entities can share a single radio effect. Entities without radio effects are automatically assigned system default effects.

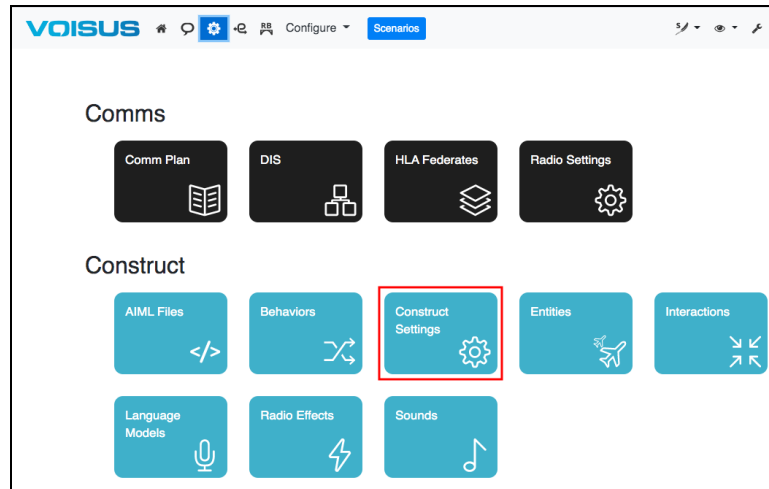


*Figure 30: Radio Effects navigation*

## 9.0 Construct Settings

**Construct Settings** allows editing miscellaneous Construct parameters. These settings affect all scenarios, not just the one you are editing. To edit Construct settings, follow these steps:

1. From the top-left navigation bar, go to **Construct** (⚙️) > **Settings**.



*Figure 31: Settings navigation*

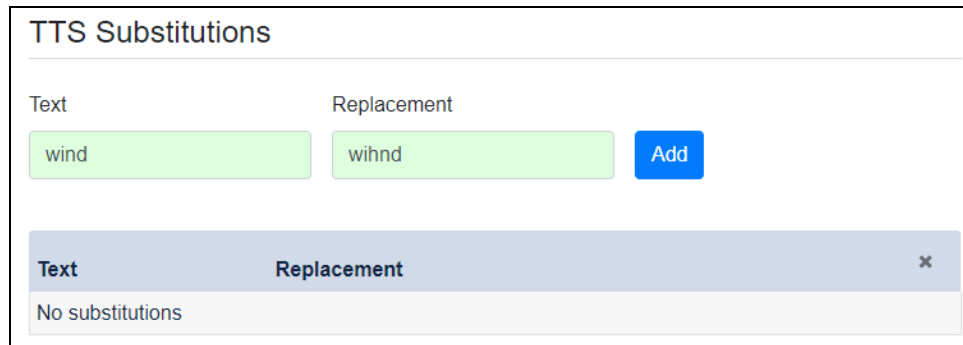
2. Set one of the following:
  - **Discovery Machine Port:** opens a port on this server to accept connections from a remote Discovery Machine artificial intelligence (AI) instance; supports external behavior modeling for Construct entities. A value of 0 disables this feature.
  - **VBS Server:** contains the IP address or host name of an associated Virtual Battlespace (VBS) server. Construct behaviors and interactions use this VBS server when specified. This features requires ASTi's JRPC-VBS Add-in.
  - **VBS Port:** the network Transmission Control Protocol (TCP) port opened by the ASTi JRPC-VBS Add-in.

Construct Settings	
Discovery Machine Port	<input type="text" value="0"/>
VBS Server	<input type="text"/>
VBS Port	<input type="text" value="65005"/>

*Figure 32: Construct port settings*

3. The **TTS Substitutions** area allows you perform text substitutions within entity transcripts. Use this feature to correct mispronunciations and transform abbreviations into the corresponding spoken form. These substitutions affect all entities and scenarios.

In **Text**, enter the expression you want to replace, and in **Replacement**, enter the corrected script. When finished, select **Add**.



**TTS Substitutions**

Text Replacement

wind wihnd Add

Text	Replacement	
No substitutions		

*Figure 33: TTS Substitutions*



**Note:** *Text* accepts Python-formatted regular expressions. For more information about Python syntax, go to [Regular expression operations](#).

## 10.0 AIML

Artificial Intelligence Markup Language (AIML) describes voice responses and simple natural language understanding for Construct entities. The XML-based AIML syntax features pattern matching, response templates, and the ability to store and retrieve variables. To activate certain response templates, Construct matches AIML input patterns against voice messages that the entity receives. Response templates modify entity state variables and generated automated voice response. Once you create an AIML definition, you can reuse it in any number of scenarios on the Voisus server.

AIML is a useful tool for creating simple reactive entities, but some applications may require more custom logic or more sophisticated natural language understanding. When an application outgrows AIML, you can use other behavior and natural language processing tools provided by ASTi or from another source. You can integrate these tools with the Construct Hypertext Transfer Protocol (HTTP) application program interface (API).

The AIML definition below matches phrases in the following formats:

- "Contact wolverine on blue four"
- "Radio check two one eight point five"
- "Radio check"
- Unknown phrases result in "say again" response

The following text shows an example of AIML code:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<aiml version="1.0">

  <category>
    <pattern>CONTACT * ON * </pattern>
    <template>
      Roger contacting <star index="1"/> on <star index="2"/>
      <think>
        <set name="Command">contact</set>
        <set name="Callsign"><star index="1"/></set>
        <set name="Frequency"><star index="2"/></set>
      </think>
    </template>
  </category>
```

```
<category>
<pattern>RADIO CHECK *</pattern>
<template>
<srai>RADIO CHECK</srai>
</template>
</category>

<category>
<pattern>RADIO CHECK</pattern>
<template>
Read you lima charlie
<think>
<set name="Command">radio_check</set>
</think>
</template>
</category>

<category>
<pattern>*</pattern>
<template>
Say again
<think>
<set name="Command">unknown</set>
</think>
</template>
</category>

</aiml>
```

## 10.1 AIML tags

Artificial Intelligence Markup Language (AIML) syntax includes the following tags:

Tag	Definition
<?xml>	Must be present as the first tag in an AIML definition.
<aiml>	AIML block delimiter; only one supported per file.
<category>	Knowledge unit containing one pattern and one response template.
<pattern>PATTERN</pattern>	Input pattern used to match received speech.
<template>	Template describing the response for an input pattern.
<star index="N"/>	Binds to the value of * for use in response templates.
<srai>PATTERN</srai>	Symbolic reduction operator for calling other categories.
<set name="VAR">VALUE</set>	Sets a variable to the specified value.
<get name="VAR"/>	Retrieves the value of a variable.
<think>	Hides the output of the computations within from the response.

**Table 2: AIML tag definitions**

## 10.2 AIML Construct integration

Construct adds functionality beyond the core Artificial Intelligence Markup Language (AIML) standard in order to more tightly integrate AIML with the state of the entity and its natural language understanding. In particular, special meaning is given to AIML variables depending on capitalization:

- Lowercase variables like *name* are synced between the entity blackboard and the AIML state. This enables setting entity attributes on the web page or in behaviors and using those variables in speech generated from AIML. Similarly, when AIML sets one of these variables, the variable is set in the entity blackboard.
- Uppercase variables like *Command* set in AIML are used as key-value pairs in the speech recognition result meaning value. The meaning is then included in the speech recognition event sent to HTTP application program interface (API) clients for use in artificial intelligence decision making.
- Variables beginning with an underscore (e.g., *\_state*) are considered private to AIML.
- The AIML response, if there is one, is immediately spoken by the entity.



## 10.3 AIML responses and meaning values

Received voice message:

```
contact wolverine on blue four
```

Matching AIML category:

```
<category>
<pattern>CONTACT * ON * </pattern>
<template>
Roger contacting <star index="1"/> on <star index="2"/>
<think>
<set name="Command">contact</set>
<set name="Callsign"><star index="1"/></set>
<set name="Frequency"><star index="2"/></set>
</think>
</template>
</category>
```

Resulting meaning value:

```
{
  "Command": "contact", "Callsign": "wolverine", "Frequency": "blue
four"
}
```

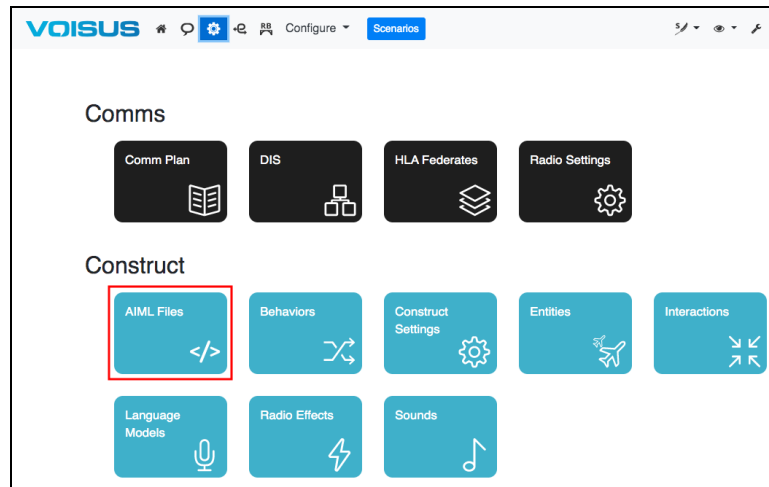
Entity voice response:

```
Roger contacting wolverine on blue four
```

## 10.4 Create and map AIML definition to entity

To create a new AIML definition and map it to an entity, follow these steps:

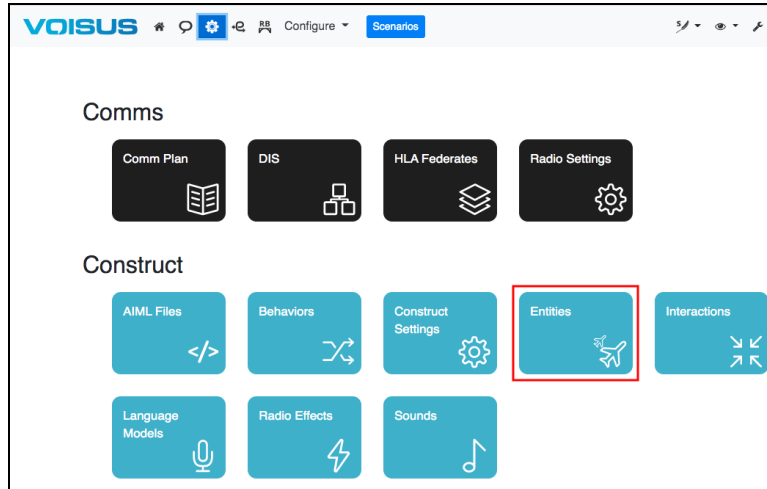
1. From the top-left navigation bar, go to **Construct** (⚙️) > **AIML**.



*Figure 34: AIML navigation*

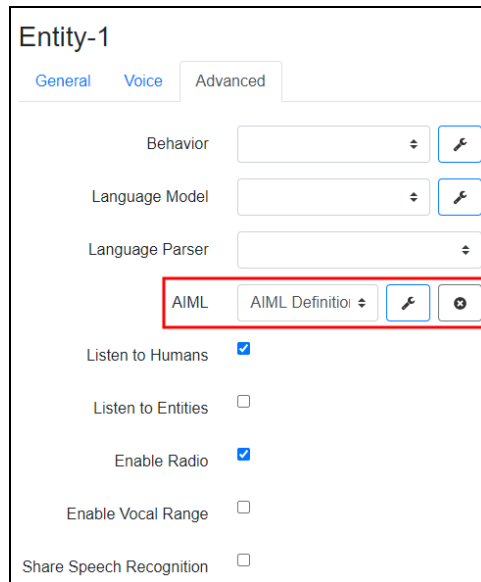
2. Under **AIML Files**, select **Create AIML File** (+).
3. On the right, in **Name**, enter a unique name for the definition.
4. In **XML Definition**, modify the definition as desired. If your AIML is large, it may be easier to edit the file on your computer, and copy and paste it into the Voisus web interface.
5. To save your changes, select outside **XML Definition**, and the definition automatically saves.

6. From the top-left navigation bar, go to **Construct** (⚙️) > **Entities**.



*Figure 35: Entities navigation*

7. Under **Entities**, choose an entity to associate with the definition.
8. On the right, go to **Advanced**.
9. Select **AIML**, and then select the definition you created in Steps 2–5.



*Figure 36: AIML definition mapping*

## 11.0 Behaviors

Behaviors enable Construct entities to listen, speak, and otherwise act autonomously in the simulation environment. These behaviors consist of a hierarchical, tree-like structure of nodes with the purpose of breaking complex high-level tasks down into smaller sub-tasks and eventually into individual actions for the entity to execute. Although behaviors can be built to automate many types of tasks, with Construct the focus is most commonly on reproducing the speech patterns and radio communication protocols of real world agents.

A few examples of actions available in Construct behaviors include:

- Listen for a radio transmission containing certain keywords (e.g., a call sign).
- Speak and transmit on a radio or face-to-face in a 3D environment.
- Execute a scripting command in attached game environments like Virtual Battlespace.
- Wait on a condition or event, then speak.

Depending on the application, behaviors are built to run for the duration of the scenario, or they can complete execution and exit after certain tasks are finished.

### 11.1 Create a new behavior

To create new behaviors, follow these steps:

1. From the top-left navigation bar, go to **Construct** (⚙️) > **Behaviors**.

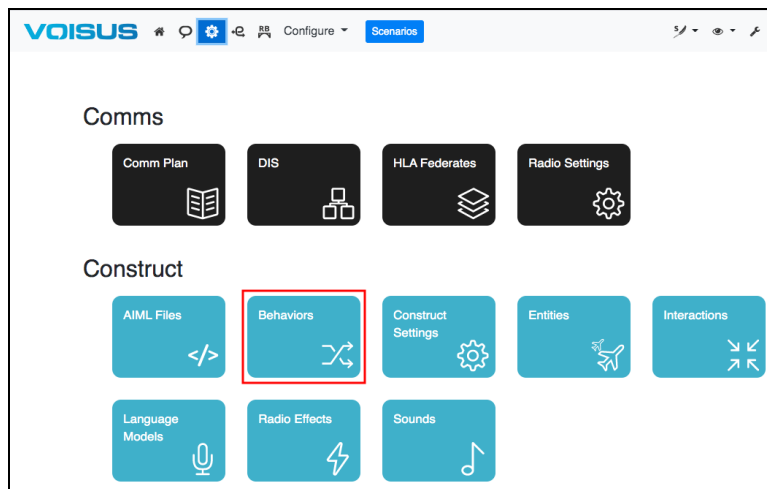
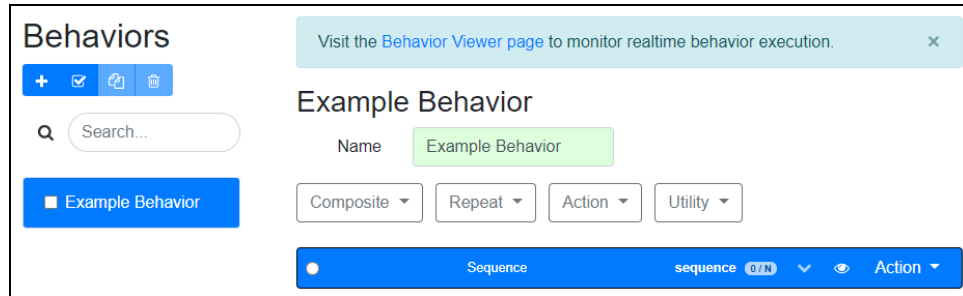


Figure 37: Behaviors navigation

2. Under **Behaviors**, select **Create Behavior** (+).
3. On the right, in **Name**, enter a unique name for the behavior.

- To add a root node to the behavior, choose a node list, and then choose a specific node type.
- To add more nodes to the behavior, select the radio button on an existing node, and use the node-type menus to add children nodes.



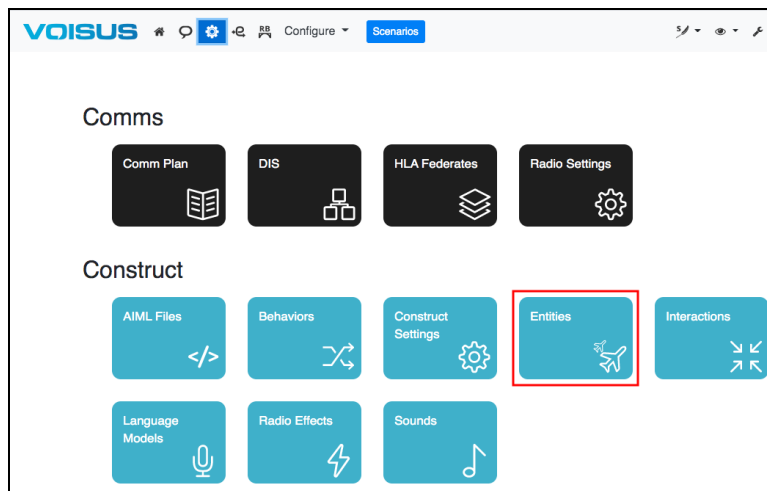
*Figure 38: Behaviors*

- To show or hide a node's configuration settings, select the eye icon (👁️).

## 11.2 Map new behavior to entity

To map the new behavior to an entity, follow these steps:

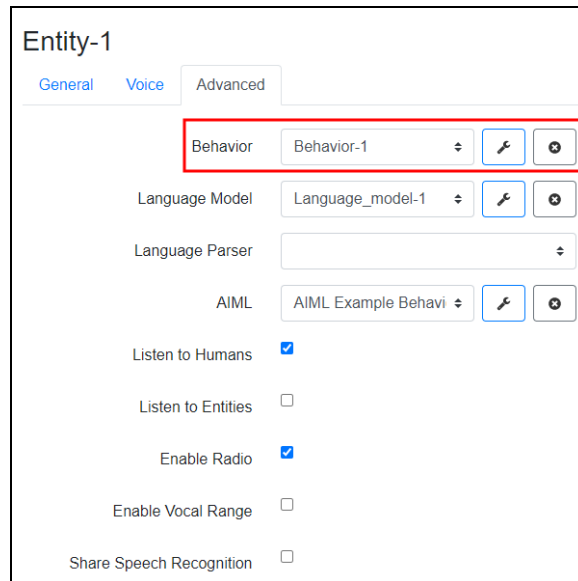
- From the top-left navigation bar, go to **Construct** (⚙️) > **Entities**.



*Figure 39: Entities navigation*

- Under **Entities**, choose the entity you will link to the behavior.
- Go to **Advanced**.

4. Select **Behavior**, and choose the behavior you created in Section 11.1, "Create a new behavior" on page 35.



The screenshot shows the 'Entity-1' configuration window with the 'Advanced' tab selected. A red rectangle highlights the 'Behavior' dropdown menu, which is currently set to 'Behavior-1'. Below this, there are several other configuration options: 'Language Model' set to 'Language\_model-1', 'Language Parser' (empty), 'AIML' set to 'AIML Example Behavi', and a series of checkboxes: 'Listen to Humans' (checked), 'Listen to Entities' (unchecked), 'Enable Radio' (checked), 'Enable Vocal Range' (unchecked), and 'Share Speech Recognition' (unchecked).

*Figure 40: Behavior entity mapping*

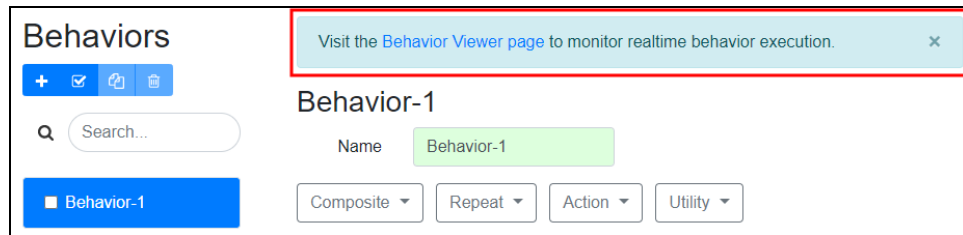
If the scenario is running, the behavior starts executing immediately. Changes made to behaviors do not take effect in real time. To enact changes to existing behaviors, reinstall the scenario. Alternatively, go to **Entity**, and clear and select the behavior in **Behavior**.

## 11.3 Behavior execution

The behavior executes several times a second until it completes or the scenario stops. Behavior execution is measured in ticks that occur several times a second. With each tick, the root node executes one step of logic, which in turn may tick some or all of the descendant nodes, depending on the structure of the tree and the individual node types. Each type of node has its own strategy or style of execution that determines how long it executes and which processing takes place during each tick. For example, a **Repeat Always** node always executes its child node once per tick and never completes execution itself. On the other hand, an extremely simple behavior might consist only of a single action node that completes execution in a single tick.

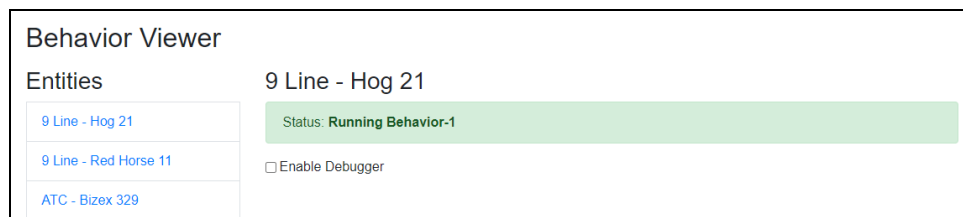
When a node finishes execution, it returns success or failure. The parent node may use this status value when deciding how to carry on. In some cases, a failure means the entire behavior fails and stops executing immediately. In other cases, the behavior tries again and succeeds once the sub-task succeeds, no matter how many attempts it takes.

To access **Behavior Viewer**, select **behavior viewer page** on **Behaviors**.



*Figure 41: Behavior viewer page link*

**Behavior Viewer** shows behavior execution and supports inserting breakpoints on nodes to suspend behavior execution for analysis. This tool becomes especially useful when building and debugging large behaviors.



*Figure 42: Behavior Viewer*

## 11.4 Behavior node types

Behavior node types and their parameters are described below. Many behaviors in Construct focus on listening for keywords using speech recognition, then generating a corresponding voice response.

**Listen** and **Say** are two types of action nodes, which cause the entity to actually take some specific action in the environment.

## 11.4.1 Composite nodes

The following table shows composite behavior nodes:

Node Type	Function
<b>Selector</b>	Runs its children in sequence until one succeeds. Returns success as soon as a single child succeeds.
<b>Sequence</b>	Runs its children in sequence until one fails. Returns success only if all children succeed.
<b>Parallel</b>	Runs its children in parallel, with the return value determined by the <b>Policy</b> setting. If <b>Policy</b> is <b>require one</b> , then success is returned when any child succeeds. If <b>require all</b> is set, then success is returned only when all children have succeeded. Parallel execution of children here means that for each tick of the <b>Parallel</b> node, each of its child nodes run one tick.

*Table 3: Composite nodes*

## 11.4.2 Repeat nodes

Table 4, "Repeat nodes" below shows repeat behavior nodes:

Node Type	Function
<b>Repeat-Always</b>	Repeatedly runs its child to completion regardless of the child's success or failure. This node type never returns.
<b>Repeat-Until-Succeed</b>	Similar to <b>Repeat-Always</b> , but returns success when its child succeeds.
<b>Repeat-Until-Fail</b>	Similar to the above, but returns success when its child fails.

*Table 4: Repeat nodes*



### 11.4.3 Action nodes

The following table shows action behavior nodes:

Node Type	Function
<b>Say</b>	<p>Speak text using text-to-speech (TTS) or a sound if one is selected. Supports text substitution using variables whose names are prefixed with "\$." For example, "Roger, this is \$callsign" returns success if the speech completes or fails if speech variables couldn't be resolved. Upon the speech finishing, a variable <i>_lastSaid</i> is set on the entity blackboard containing the text of what was spoken, or the sound ID if a sound was played. When the node is about to succeed, three variables are temporarily set on the entity blackboard:</p> <ul style="list-style-type: none"> <li>• <i>rec_text</i></li> <li>• <i>rec_meaning</i></li> <li>• <i>rec_conf</i></li> </ul> <p>The expression is then evaluated, which may read these values and save them elsewhere, then the three variables are cleared from the blackboard.</p>
<b>Listen</b>	Blocks until a voice message is received, at which time it succeeds or fails based on whether the specified Keywords and require conditions are matched. Keywords are a comma-separated list of words in the message text. Require is a behavior expression (explained below) that is evaluated to True or False.
<b>Assert</b>	Succeeds or fails based on the evaluation of the specified expression.
<b>AssertPosition</b>	Succeeds if the entity is currently within distance meters of the point specified by the X, Y, and Z coordinates.
<b>VBS Command</b>	Executes a scripting command on an attached Virtual Battlespace (VBS) instance, if one exists. Returns success if the command executes or fails if VBS isn't connected.
<b>Event</b>	Waits until a specific named event occurs, at which time success is returned. Fails if a different event is raised first.
<b>Wait</b>	Waits Time seconds and then succeeds.
<b>Wait-for-Silence</b>	Waits for radio silence and succeeds when silence is detected or fails if Timeout seconds elapses first.
<b>Expression</b>	Evaluates the expression against the entity blackboard.

**Table 5: Action nodes**

### 11.4.4 Utility nodes

Utility nodes support exactly one child node and modify the child's execution or return value in some way. The following table shows utility behavior nodes:

Node Type	Function
<b>Timeout</b>	Executes the child until it returns, or until time has elapsed. The <i>FinalValue</i> parameter determines the success or failure of this node upon timeout.
<b>Limit</b>	Limits the execution rate (tick rate) of its child to a specified interval in seconds.
<b>Flip</b>	Runs its child until completion but returns an inverted success or failure result.
<b>Exc-Handler</b>	Catches and suppresses exceptions raised when executing its child. Returns the success or failure value from the child, unless an exception is raised, in which case it always fails.

*Table 6: Utility nodes*

## 11.5 Behavior expressions

Several node types evaluate expressions during execution as a means to modify the entity and influence behavior execution. Expressions are entered by the behavior developer in a simple syntax that should look familiar to programmers that have used imperative programming languages (e.g., C, Python).

All expressions are evaluated against the entity's blackboard, which is a simple storage area for each entity's data. For example, the expression **count = 1** sets a variable count to the value 1 in the current entity. Blackboard variables can then be used as variables in speech or be referenced for other purposes by other nodes in the behavior.

In addition to being able to read and write arbitrary variables on the entity blackboard, expressions can access the entity's web page attributes through an implicit blackboard variable named entity. For example, to change the run time entity's name, use the **entity.name = "John Oliver"** expression.

Expressions:

- May contain multiple statements separated by semicolons
- Support creating and manipulating data in the JavaScript Object Notation (JSON) format
- Are executed by a simple interpreter that does not support complex statements
- Retrieve and set variables on the entity blackboard

Example expressions that demonstrate the available operators and functions:

Expression	Note
count = 1	Number assignment
count += 2	Number increment
count -= 2	Number decrement
count == 3.3	Number equality
count != 3.3	Number inequality
x = count * 2	Number multiplication
name = "Striker 7"	String assignment
name == "Striker 6"	String equality
name != "Striker 6"	String inequality
"three two" in text	String contains
times = [1,2,3]	List assignment
times.append(5)	List append
times.remove(5)	List remove
time = times[1]	List access
flight = { "number": 42 }	Object assignment
flight["number"] = 43	Object item assignment
flight["number"]	Object item access
x    y	Logical OR
x && y	Logical AND
tmp = count; count = 0	Chaining multiple statements
a=1; b=2; c=a+b; d=c*55;	Chaining multiple statements; accessing variables

**Table 7: Example expressions**