ALC Toolchain Documentation

Release 23.03.16

Institute for Software Integrated Systems - Vanderbilt University
Contact: ALC Team (alc-team@vanderbilt.edu)

Mar 16, 2023

TABLE OF CONTENTS

CHAPTER

ONE

RELEASE NOTES

March 16, 2023

This is the initial open-source and public release of the ALC toolchain developed as part of the DARPA Assured Autonomy project.

The release demonstrates the technologies developed as part of this project with a simulation platform for BlueROV2, an Unmanned Underwater Vehicle.

CHAPTER

TWO

STANDARD SETUP

Note: The release was succesfully tested on GTX10xx (Pascal), RTX20xx (Turing) and RTX30xx (Ampere) family.

2.1 Prerequisites

Setup/configure:

- Linux OS (tested with Ubuntu 16.04 & 18.04)
- NVidia GPU and Driver
  - Nvidia Driver version should be compatabile with CUDA 11.2
  - For existing driver installations, use the command "nvidia-smi" to check the current driver version
- Docker
  - Tested with docker version 20.10.(7,8)
  - Configure your user account to use docker without 'sudo'
  - Be sure to log out, then log back in so group changes will be applied
- Docker Compose File Format
  - Version should be >= 2.3 and < 3.0
- Nvidia Docker runtime

> **Warning:** 'nvidia-docker2' has been deprecated in favor of 'nvidia-container-toolkit'. However, Docker Compose has not caught up with these changes, and 'nvidia-docker2' is still required for ALC. Specific installation instructions can be found here.

- Other tools required as part of the setup.
  - Install Git

```
sudo apt-get update
sudo apt-get install git
```

  - Install Git LFS
  - Install openssl (Version 1.1.1. or above). Please refer to this link to update the openssl version to 1.1.1.

- Install openssh-client

```
sudo apt-get install openssh-client
```

2.2 Installation

1. Setup environment

   i) Edit ~/.bashrc and add the following Environment Variables:

   - ALC_HOME - Directory where ALC repository is/will be installed
   - ALC_WORKING_DIR - Workspace directory used for storing job results (simulation, training, etc.)
   - ALC_DOCKERFILES - Runtime directory for docker containers (contains WebGME MongoDB database, configuration files, etc.)

   Note: These directories can be anywhere on the system, but the user must have write permissions within each directory.

   If any permission issues are encountered, change the directory owner with the following command:

```
sudo chown $USER:$USER <desired_dir>
```

   Example bash_rc addition:

```
# ALC Variables
export ALC_HOME=$HOME/alc
export ALC_WORKING_DIR=/hdd0/alc_workspace
export ALC_DOCKERFILES=$HOME/alc_dockerfiles

# Here /hdd0 is the mount point of a secondary drive with ample␣
↪storage for large
# datasets. Secondary drive is NOT required.
```

   ii) Source updated bashrc and create directories

```
source ~/.bashrc
sudo mkdir $ALC_WORKING_DIR
sudo chown $USER:$USER $ALC_WORKING_DIR
```

2. Get ALC images and codebase.

   i) Clone the alc repository

```
git clone git@github.com:AbLECPS/alc.git $ALC_HOME
```

   ii) Pull the docker images (about ~20 GB) (Please read the notes below.)

```
$ALC_HOME/docker/alc/pull_images_from_server.sh
```

   Note: The docker image in all would be around 20GB. The docker images and containers are stored in /var/lib/docker. If you don't have space on the disk where this folder exists, please consider the

configuring the docker daemon to use another directory on a drive with sufficient space. Instructions on how to set this up can be found here

Note: The docker images supplied as part of this release were built off a GPU-based tensorflow (v 2.6.2) docker image requires a minimum of CUDA 11.2.

3. Copy data and pre-trained models

Warning: This step will download several MB of data. The Bluerov model execution depends on the data and trained weights. It also includes bluerov2 description file for running BlueROV simulation in Gazebo.

```
$ALC_HOME/docker/alc/pull_data.sh
```

4. Setup the ALC toolchain

```
cd $ALC_HOME/docker/alc
./setup.sh
```

5. Perform Slurm workload manager configuration

```
cd $ALC_DOCKERFILES/slurm/etc
```

Using your preferred text editor, edit "slurm.conf" and "gres.conf" files within this directory to match your machine's hardware (CPU count, available Memory, & GPU information). Note that the default compute node configuration info is located near the bottom of the "slurm.conf" file, and looks as follows:

```
# COMPUTE NODES
NodeName=alc_slurm_node0 CPUs=16 RealMemory=30000 Sockets=1␣
↪CoresPerSocket=8 ThreadsPerCore=2 State=UNKNOWN Gres=gpu:turing:1
```

Slurm configuration requires that provided values be strictly less than or equal to actual available hardware. For example, a computer with 16 GB of RAM may actually report 15,999 MB available. In this case, setting 'RealMemory' to 16,000 will cause an error. Must be set <= 15,999.

The "gres.conf" file may also need to be edited depending on the available GPU(s). The information in this file must match what is specified in the "Gres" field of the "slurm.conf" configuration, and looks as follows:

```
NodeName=alc_slurm_node0 Name=gpu Type=turing File=/dev/nvidia0
```

Note: The GPU type "turing" in each of the above files refers to the Nvidia GPU architecture code name. Listing of the different code names and the corresponding GPUs can be found at https://nouveau.freedesktop.org/wiki/CodeNames/. See Slurm documentation on these two files if more information is needed.

By default Slurm is configured to use only one worker, the machine where the ALC Toolchain is running. However, Slurm can support multiple worker nodes if desired. Setup instructions for a multiple node cluster can be found at *$ALC_HOME/docker/alc/WORKER_SETUP.md*.

6. Setup and Build Local GIT repository

   The ALC toolset runs a local GIT server the following commands initializes the git repositories associated in the server, checkouts a local copy and builds it so that the toolchain is ready for execution.

```
$ALC_HOME/docker/alc/setup_and_build_repos.sh ALL
```

7. Start ALC toolchain

```
cd $ALC_HOME/docker/alc
./run_services.sh
```

   This script launches the ALC toolchain services using docker-compose. It does not detach from docker-compose by default, and all logs will be displayed in this terminal. To run services in the background, use

```
./run_services.sh -d
```

8. Setting up local docker registry

   In order to use the integrated IDE, you need to setup a local docker registry by executing the script below.

```
cd $ALC_HOME/docker/alc
./update_registry.sh
```

   Note: This needs to be done during inital setup and if the images are updated. The script assumes that the run_services script (mentioned above) is launched and running succesfully.

9. Using the ALC Toolchain

   1) Follow the instructions in *First Login* to access the toolchain.
   2) Follow the instructions in *Initial Activities With ALC* to ensure the toolchain was installed correctly.
   3) Follow the instructions in *Working with IDE* to use the IDE integrated with WebGME for the toolchain.
   4) Read each page in the *Tutorials* section starting with *ALC Toolchain Overview* .
   5) Optional software packages can be installed for additional features. See optional_setup.

2.3 Additional Notes

1. The certificates are self-generated. So browsers prompt warning about the certificates. Please ignore the warnings and proceed to the site.
2. If you are accessing the Toolchain remotely, then replace `localhost` with the address of the server hosting the ALC services.

CHAPTER

THREE

FIRST LOGIN

3.1 Instructions

1) Open your internet browser and navigate to http://localhost:8000/profile/login. You will be redirected to https://localhost

2) Login to the ALC Toolchain

   Default login credentials:

   Username: admin
   Password: vanderbilt

3) Click on "Go to editor" button in the top-right, or visit the url: http://localhost:8000 or https://localhost

Notes/Recommendations:

- Chrome is the recommended browser, but Firefox also works. Other browsers are untested.
- By default, the toolchain can be accessed by anyone who can connect to port 8000 of the host machine. Recommend configuring firewall rules to restrict access as desired.
- Recommend changing the password of the "admin" user account.
- Recommend each user create a new account by accessing http://localhost:8000/profile/login and clicking "Register" in the bottom left of the login prompt. Each new user must be added to the "ALC" group by the admin account in order to access the models.
- A guest account is available by default which does not require any authentication, but does not have permission to access the models.

CHAPTER

FOUR

INITIAL ACTIVITIES WITH ALC

This document points to some initial set of activities (jobs) that you can execute using BlueROV models.

1. As part of the initial setup process you should have run the following command to setup the local repositories and build the sources. If you have not already done so, please run the following command

```
$ALC_HOME/docker/alc/setup_and_build_repos.sh ALL
```

2. **View Existing Results**

   In order to view pre-computed results with BlueROV,

   1) Open the model admin_BlueROV
   2) Navigate to *ALC/2.Construction/Testing*.
   3) Open any of the models (say *CP1**, *CP2*, *CP4**) and click on *Artifact Index* on the left
   4) Click on any of the *Result* link. If the data set had been copied correctly, a jupyter link should open up and you should be able to view the results.

3. **Run Simulation**

   This part deals with running the BlueROV simulations from ALC.

   In order to launch BlueROV simulations of existing scenarios from ALC,

   1) Open the admin_BlueROV model.
   2) Navigate to *ALC/2.Construction/Testing* model.
   3) Open any of the models (say *CP1**, *CP2*, *CP4**).
   4) Click on the LaunchActivity plugin by clicking on the "Play like" button on the top-left and choose LaunchActivity.
   5) This should launch the activity.
   6) When the job is successfully submitted, a green display message should appear on the top right of the ALC browser.
   7) Click on the ArtifactIndex view on the left, the line corresponding to the launched simulation should change colors based on the scheme below. Gray to Yellow (when job is submitted to backend slurm engine), Yellow to Blue (when the execution starts), Blue to Green (when the execution is successfully completed), or Blue to Red(for failure). The log link gives the results.
   8) When the job is running (blue state above), click on the 'Sim' link to view the RViz output. This might take a while as the simulation needs to be setup before the link can

show anything. You might need to drag the RViz window as it could start in a corner of the screen.

9) Once the job is successfully completed (green), click on the result link execute the jupyter Notebook to see the plots.

10) Refer to the *BlueROV2 Activity Definition* for more information on BlueROV2 scenarios.

4. **Execute with ALC IDE**

   Next section deals with executing BlueROV2 scenarios by launching the ALC/IDE.

CHAPTER

FIVE

WORKING WITH IDE

5.1 Launch IDE from ALC

The following set of instructions deal with launching the web-based IDE within ALC. The IDE used here is based on open source VSCode - code-server

1) The IDE setup with an associated repository is available for the models from the BlueROVActivity seed (admin_BlueROV)
2) Login to the ALC Toolchain as described in *First Login*, click on "Go to editor" button in the top-right,and open the model (admin_BlueROV)
3) If you are not in the "ALC" model, click into the "ALC" model that includes the following models *1. Modeling*, *2.Construction* etc.
4) Click on the IDE visualizer in the Visualizer Panel on the left.
5) Launch the plugin *IDE Launch* from the plugin button on the top-left and choose "start" option.

   This should start setting up the IDE at the backend. It does the following

   a) Sets up a private docker daemon and sandbox for the user. The initial setup takes a few minutes. Depending on the server, this could take upto 10-15 minutes.

   b) Clones the git-repo associated with the model. If there is no repo associated with the model, this step is skipped.
6) Once the plugin completes its steps, the links on the table (in the IDE visualizer) are enabled.
7) The user can click on the IDE link to access the VSCode Server IDE.
8) When prompted for password, please type in the following login credential

   Password: vanderbilt1
9) After login, the user is presented the IDE and the repo files appear in the tree-browser. The user perform regular operations as with any IDE and with the source base available.

   The user should be able to

   1) build and run the code (as desribed below for BlueROV)
   2) view the graphics (RViz, RQT, Gazebo) on the VNC link that in the webgme IDE visualizer.
   3) perform updates to the code-base, debug and run and view the results.
   4) commit and push to the underlying git-repository.
   5) therafter the updated code would be available for launching

5.2 Initial Setup in IDE

Once the IDE has been launched with the admin+BlueROV model, open a terminal and run the IDE setup script for the IDE to work with the BlueROV2 code base.

```
./setup_ide.sh
```

Optionally user can run the steps manually also with the following commands:

1.) Use the IDE to open the file */etc/hosts* file and the following lines to the end of the file.

```
172.18.0.2    ros-master
172.18.0.4    aa_uuvsim
```

2.) To update the environment variables, use the IDE to open the file *~/.bashrc* and add the following line.

(Note: Typing ~ on the file dialog will take you to the home folder)

For BlueROV:

```
export ALC_HOME=$REPO_HOME/bluerov2_standalone
```

3.) Open a new terminal window in the IDE and execute the command below to build the ROS Packages for the project.

```
cd $ALC_HOME
./build_sources.sh
```

4.) Connect the IDE (aka the new hostmachine) to the ROS docker network.

```
docker network connect --ip 172.18.0.7  ros  alc_codeserver
```

5.3 BlueROV Execution in IDE

In the IDE open three terminal windows and type in the following commands in each of the windows:

NOTE: You may open a terminal and split it into three windows. In order to arrange the terminals one below another right click on or near the words "Terminal" and click on "move panel to right".

NOTE: Execution in IDE is almost identical as runnig on the host machine - differences are noted.

1.) **Start roscore docker**

```
cd $ALC_HOME/catkin_ws
source run_roscore.sh
```

2.) **Start bluerov_sim docker**

```
cd $ALC_HOME/catkin_ws
source run_bluerov_sim.sh
```

In the docker run the following commands

```
source run_xvfb.sh
source /aa/src/vandy_bluerov/scripts/bluerov_launch.sh
```

Preset scenarios

If you want to run a preset scenario, run eg.:

```
source /aa/src/vandy_bluerov/scripts/cp1_00.sh
```

1) Refer to the *BlueROV2 Activity Definition* for more information on BlueROV2 scenarios.

3.) **Optional for visual representation: start rviz**

```
cd $ALC_HOME/catkin_ws
source run_rviz_ide.sh
```

To view the RViz, go back to the ALC model, and click on the *VNC* link (below the IDE *VSCode* link) and launch the web-based VNC to view the RViz display.

NOTE: On host machine use *source ros_melodic_setup_gazebo_ros_connection.sh* instead of *source run_rviz_ide.sh*

4.) **Optional for visual representation: start RQT PyTrees BT graph**

In a new terminal:

```
cd $ALC_HOME/catkin_ws
source run_rqt_ide.sh
```

To view the RQT py trees graph, go back to the ALC model, and click on the *VNC* link (below the IDE *VSCode* link) and launch the web-based VNC to view the display, and select /BlueROV_tree/log/tree publisher from the top left dropdown.

5.) Once done with using the IDE, please remember to shutdown the IDE as described in the section *Shutting down IDE*

Known issues

- Running simulation in IDE or on the host machine, Gazebo sometimes dies at the beginning of the simulation like this:

```
[gazebo-35] process has died [pid 5144, exit code 134,
cmd /opt/ros/kinetic/lib/gazebo_ros/gzserver
-u --verbose -e ode --seed 27168 worlds/environment_empty.world __name:=gazebo
__log:=/root/.ros/log/163f34f0-1a0a-11ec-b167-0242ac120002/gazebo-35.log].
log file: /root/.ros/log/163f34f0-1a0a-11ec-b167-0242ac120002/gazebo-35*.log
```

  In this case you have to terminate and restart simulation.

- Running RViz in IDE > VNC window has some limitations caused by the virtual GUI.
  - After each simulation, you have to reopen RViz to visualize the running experiment.

5.4 Shutting down IDE

Once you are done working with the IDE, to shut it down

1) Open the BlueROV webgme model and drill to “ALC” model.
2) Click on the IDE visualizer in the Visualizer Panel on the left.
3) Launch the plugin *IDE Launch* from the plugin button on the top-left by choosing the “stop” option. This will shutdown the IDE and the links on the table will turn to normal text.

4) You may restart the IDE at a later point, by launching the *IDE Launch* plugin with "start" option. Unlike the first time, the IDE should be launched relatively quickly. The *VSCode* and *VNC* text on the table will turn to links.

Note : Sometimes the *IDE Launch* plugin returns quickly and clicking on the link leads to a *Bad Gateway* error message. This just means that the docker is still starting up the IDE. Wait for a minute and refresh the page.

5.5 Git and Docker registry support

The section *Git and Registry Support* discusses the git and docker registry support in ALC.

Part I

Tutorials

CHAPTER

SIX

ALC TOOLCHAIN OVERVIEW

Cyber-Physical Systems (CPS) are commonly used in mission-critical or safety-critical applications which demand high reliability and strong assurance for safety. These systems frequently operate in highly uncertain environments where it is infeasible to explicitly design for all possible situations within the environment. Assuring safety in these systems requires supporting evidence from testing data, formal verification, expert analysis, etc. Data-driven methods, such as machine learning, are being applied in CPS development to address these challenges.

The Assurance-based Learning-enabled Cyber-Physical Systems (ALC) toolchain is an integrated set of tools and corresponding workflows specifically tailored for model-based development of CPSs that utilize learning-enabled components (or LECs). Machine learning relies on inferring relationships from data instead of deriving them from analytical models, leading many systems employing LECs to rely almost entirely on testing results as the primary source of evidence. However, test data alone is generally insufficient for assurance of safety-critical systems to detect all the possible edge cases. This set of tools support various tasks including architectural system modeling, data construction of experimental data and LEC training sets, performance evaluation using formal verification methods and system safety assurance monitoring. Fig. 6.1 shows the general order activity for each of these steps. Each step of the process can be refined through iterations to adjust parameters, retrain LECs, adjust testing solution spaces, etc.

Evidence used for safety assurance should be traceable and reproducible. Since LECs are trained with data instead of derived from analytical models, the quality of an LEC is dependent on the history and quality of the training data. Therefore, it is necessary to maintain data provenance when working with LECs to allow the model to be reproducible. Manual data management across the complex tool suites often used for CPS development is a time consuming and error-prone process. This issue is even more pronounced for systems using LECs where training data and the resulting trained models must also be properly managed. With this toolchain, all generated artifacts - including system models, simulation data, trained networks, etc. - are automatically stored as accessible data sets and managed to allow for both traceability and reproducibility.

The design process begins with the initial modeling of the how the system hardware and software components act and interact. A system architecture model based on SysML Internal Block Diagrams allows the user to describe the system architecture in terms of the underlying components (hierarchical blocks) and their interaction via signal, energy, and material flows. System configuration instances are defined and provide parameters for adjustments during testing efforts to allow exploration of the system use cases and optimization of the system elements to meet the desired requirements.

Once the system elements have been modeled, relevant data is created to allow testing and evaluations of the system performance. LECs are built using either supervised or reinforcement learning techniques. Once a LEC is created, it can be retrained given different system scenarios and configurations to optimize the system. Assurance monitoring LECs are created simultaneously either utilizing training data used to create the system LECs (for supervised learning) or created from trained LECs (for reinforcement learning).

The verification, validation and assurance testing provide methods to assess the training models and their ability to meet the system requirements and execute the desired tasks safely. A fundamental problem with LECs is that the training set is finite and it may not capture all possible situations the system encounters at operation time. For such unknown situations, the LEC may produce incorrect or unacceptable results – and the rest of the system may not even know. Continuous monitoring of the LEC performance and level of confidence in the output of the LEC

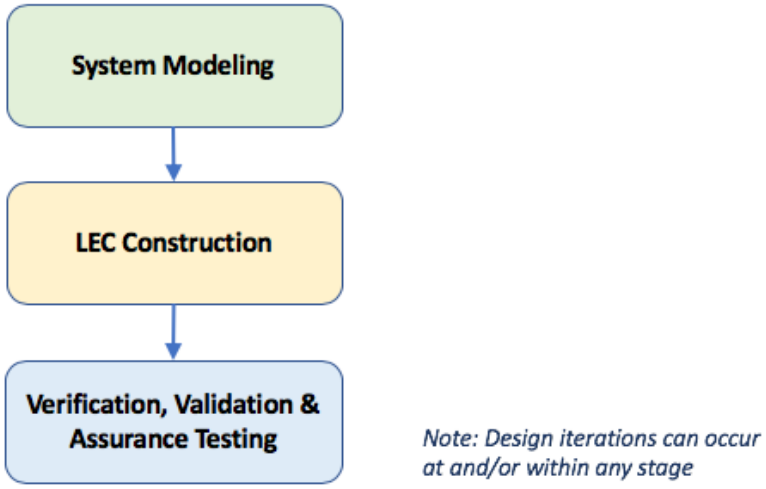


Fig. 6.1: ALC Toolchain Design Flow

enables assurance monitoring, which oversees the LEC and gives a clear indication of problematic situations. Formal verification methods and testing evaluation metrics can be used to determine the solution space possible with the trained LEC and robustness of the system while under adversarial attacks. This information can also be referenced as evidence in static system assurance arguments.

Each portion of the model can be used to iterate on the design for improvement of the LECs models, adjustment of the system design parameters to determine impact or alteration of the testing scenarios to include solution spaces not originally used in the design process to determine performance issues. Workflow tools are available to simplify the automation of these system level iterative tasks.

6.1 Toolchain Resources

The toolchain is built on the WebGME infrastructure which provides a web-based, collaborative environment where changes are automatically and immediately propagated to all active users. The user created system models, data collection and testing activities are created and managed utilizing the WebGME servers accessed using web browsers from remote terminals, as shown in Fig. 6.2. In order to promote reproducibility and maintain data provenance, all models, training data, and contextual data are stored in a version-controlled database and data management is automated.

The toolchain supports embedded Jupyter notebooks within the context of an experiment, training, or evaluation model. The users can configure the code in a Jupyter notebook to execute the model. This allows users to launch their execution instances in an interactive manner and debug their code if required. Additionally, it allows users to write custom code to evaluate the system performance.

Whenever any model is executed, all parameters and configuration data needed to repeat the execution are stored in a metadata file with the results. This metadata file also contains references to any artifacts used as inputs to the model in order to maintain data provenance. Metadata files for LEC training contain the Uniform Resource Identifier (URI) of each data file used in the training set as well as a copy of the parent LEC metadata if training was continued from a previously trained model. Similarly, metadata for an evaluation experiment contains references to any trained LECs used in the experiment. This ensures that the complete history of any artifact can be traced back to the original data regardless of how many iterations of the design cycle are required. Additionally, the toolchain includes a dataset manager for viewing and analyzing this lineage.

The ALC toolchain allows remote deployment of computationally intense tasks on appropriately equipped servers

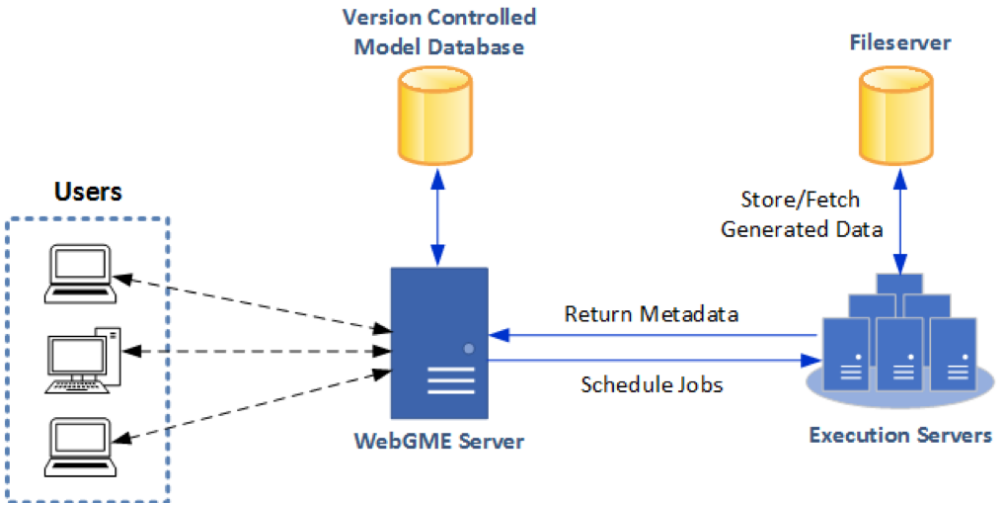


Fig. 6.2: ALC Toolchain Resource Usage

(or execution servers). This enables developers of CPS to configure and launch computationally intensive system execution (or simulation) and training exercises on powerful machines from local web browsers, while collaborating with a distributed team of developers. The execution server is often a remote server utilizing available forms of hardware acceleration, such as graphics processing units (GPU), digital signal processors (DSP), FPGA or ASIC.

Note: This remote execution is not currently available, only one execution server (where the WebGME server is located) is available at this time. Results will be passed back with the Metadata (not stored remotely).

Large data sets (eg. simulation data and trained LEC models) are stored on dedicated fileservers. Each data set is linked to a corresponding metadata file which is returned to the WebGME server and stored in the version controlled model database. The metadata files provide enough information for retrieving a particular data set from the fileserver when needed for other tasks such as LEC training, performance evaluation, or LEC deployment. When experiment results are uploaded to the fileserver, configuration files and other artifacts used to execute the experiments are stored with the generated data. This allows the experiment to be repeated and any generated data to be reproduced as needed. This pattern of uploading the data to a dedicated server and only storing the corresponding metadata in the model frees WebGME from handling large files and improves efficiency as well as model-scalability. Additionally, WebGME provides a version control scheme similar to Git where model updates are stored in a tree structure and assigned an SHA1 hash. For each update, only the differences between the current state and the previous state of the model are stored in the tree. This allows the model to be reverted to any previous state in the history by rolling back changes until the hash corresponding to the desired state is reached. User access to the model data sets is available in a section labeled `DataSets`.

6.2 WebGME Tutorial Model

This tutorial will illustrate the development of an simple autonomous car using the ALC Toolchain to demonstrate the toolchain capabilities, shown in Fig. 6.3. The goal of this system is to drive around a track using images from a forward-looking camera and determining the desired drive parameters from a LEC model. An original training dataset is information collected from the car while a person manually drives it around. The dataset will be used to train the LEC model. This tutorial will show how to:

1. build the system model to represent the car (including the LEC model)
2. build and run a simulation experiment using Gazebo to drive a simulated car around a track (without LEC)

3. training a LEC model using imported dataset from manual data collection
4. run a simulation using the LEC using Gazebo to drive a simulated car around a track

Items expected in the near future will include:

5. training an assurance monitor associated with the trained LEC model
6. testing using the trained assurance monitor in the system testing
7. verification of the LEC model
8. download model generated ROS code to a physical car for autonomous driving

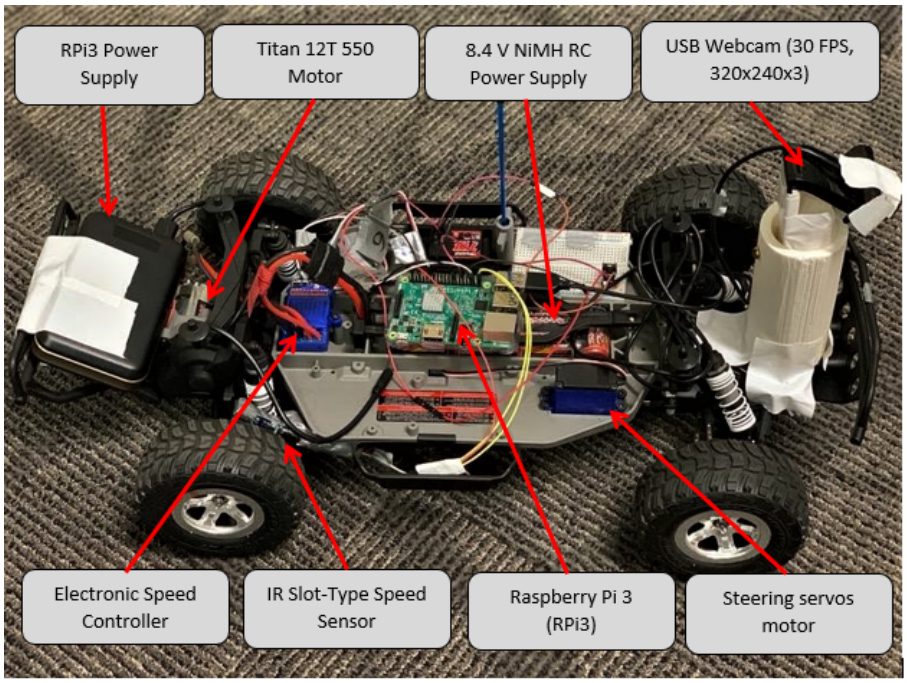


Fig. 6.3: Tutorial Car Image

For more information about the car used in this tutorial, refer to https://github.com/scope-lab-vu/deep-nn-car.

It is assumed that the user is familiar with Machine Learning techniques. The tool builds upon various technologies to create the desired set of capabilities. Key items that are utilized, but will not be explained in this tutorial are as follows. It is recommended that the user read the following documentation, if they are unfamiliar with these technologies.

- WebGME - Documentation and Tutorial Videos
- Robot Operating System (ROS)
- Docker
- Jupyter Notebooks Tutorial
- Gazebo
- TensorFlow
- Keras

A seed project named `ALC_F1_10` is available with the car tutorial. To follow along and see how the car model is built, create a new WebGME project by clicking on the `GME` in the top left of the tool and choose `New Project`

. . . . After providing a project name, select the `ALC_F1_10` seed from the `Choose an existing seed` drop-down menu. This will create a copy of the seed project for you to modify and execute experiments while learning the tool.

The top level (or root) of the ALC model, shown in Fig. 6.4, consists of the 3 key model elements outlined in the *ALC Toolchain Overviews*: System Modeling (**Modeling**), LEC Construction (**Construction**) and Verification, Validation and Assurance (**V&V&A**). In addition, there is a **Workflows** block that allows design of multiple workflow models to exercise the key model elements given desired adjustment parameters that enable further exploration of the design. As experiments are run and LECs training sets become available, the **DataSets** block provides easy access to the data available at each stage of data construction and testing phases. This includes configuration parameters, LEC involved, testing results and traces between consecutive dataset generation to provide traceability information.

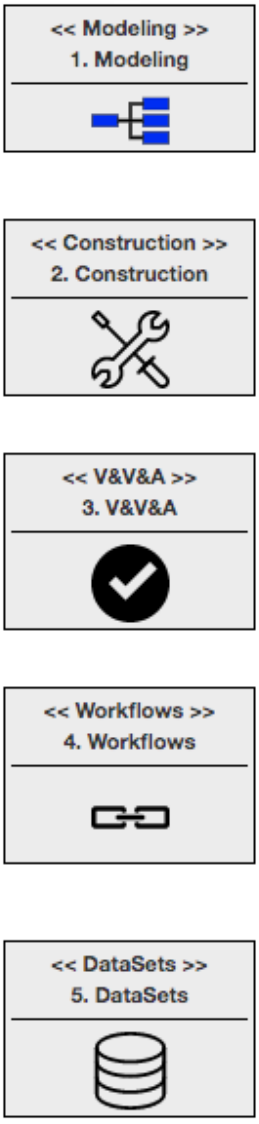


Fig. 6.4: ALC Model Elements

6.3 References

- Charles Hartsell, Nagabhushan Mahadevan, Shreyas Ramakrishna, Abhishek Dubey, Theodore Bapty, Taylor Johnson, Xenofon Koutsoukos, Janos Sztipanovits, and Gabor Karsai. 2019. CPS Design with Learning-Enabled Components: A Case Study. In Proceedings of 30th International Workshop on Rapid System Prototyping(RSP'19), New York, NY, USA, October 17–18, 2019 (RSP'19), 7 pages. https://doi.org/10.1145/3339985.3358491
- Charles Hartsell, Nagabhushan Mahadevan, Shreyas Ramakrishna, Abhishek Dubey, Theodore Bapty, Taylor Johnson, Xenofon Koutsoukos, Janos Sztipanovits, and Gabor Karsai. 2019. Model-Based Design for CPS with Learning-Enabled Components. In Proceedings of DESTION '19: Design Automation for CPS and IoT, Montreal, QC, Canada, April 15, 2019 (DESTION '19), 9 pages. https://doi.org/10.1145/3313151.3313166
- Shreyas Ramakrishna, Abhishek Dubey, Matthew P Burruss, Charles Hartsell, Nagabhushan Mahadevan, Saideep Nannapaneni, Aron Laszka, Gabor Karsai. 2019. Augmenting Learning Components for Safety in Resource Constrained Autonomous Robots. 22nd IEEE International Symposium on Real-Time Distributed Computing (ISORC), Valencia, Spain, May, 2019. https://doi.org/10.1109/ISORC.2019.00032
- Tarang Shah, "ROS Basics 1 - Nodes, Topics, Services & Actions", https://tarangshah.com/blog/2017-04-01/ros-basics-1-nodes-topics-services-actions/
- Guillaume Finance, 2010. SysML Modelling Language explained. http://www.omgsysml.org/SysML_Modelling_Language_explained-finance.pdf
- Sanford Friedenthal, Alan Moore, Rick Steiner, 2019. OMG Systems Modeling Language (OMG SysML™) Tutorial. International Council on Systems Engineering (INCOSE). http://www.omgsysml.org/INCOSE-OMGSysML-Tutorial-Final-090901.pdf
- Dimitrios Boursinos and Xenofon Koutsoukos. 2020. Assurance Monitoring Of Cyber-physical Systems With Machine Learning Components. In Proceedings of TMCE 2020, 11-15 May, 2020, Dublin, Ireland
- Charles Hartsell, Nagabhushan Mahadevan, Harmon Nine, Abhishek Dubey, Ted Bapty, and Gabor Karsai. 2020. Workflow Automation for Cyber Physical System Development Processes. Accepted for publication in Proceedings of the Workshop on Design Automation for CPS and IoT (DESTION '20). ACM, New York, NY, USA, Workflow_transform1-9, https://arxiv.org/abs/2004.05654.

CHAPTER

SEVEN

ACTIVITY MODELS

The ALC models can be extended using *Activity Definition* and *Activity* models. This section describes these models and points to the examples of these models in the projects.

1. Activity Definition Models

   Activity Definition models are used to define any execution that the users might want to perform. In the context of ALC models this could include training, simulation, data pre-processing etc.

   An *ActivityDefinition* includes the following

   - A Definition that includes the parameters of interest to configure the activity. This is presented to the user in the ALC model to configure the activity instances.
   - An ActivityInterpreter code that will work on the user inputs (parameters) to setup and launch the execution.
   - Dockerfile and build scripts for each Docker image that is required to execute the activity.
   - Entrypoint scripts to each of the docker container instances that are launched (if required).
   - Any additional files and artifacts that are required to build the docker image or supply to the docker container.
   - A Deployment file to launch all the docker containers that are configured with appropriate mount points to execute the activity.

   Example *Activity Definition* models exist in the ALCCore and BlueROVActivity projects. From the Root-level of the model, go into the *Extensions* folder.

1) *ActivitySetup* plugin

   This plugin helps build and register the *Activity Definition* on the ALC instance. This allows the *Activity Definition* to be available for initializing *Activity* models which can be setup for execution (see below for more details)

   Currently, the existing *Activity Definition* models in the ALCCore, BlueROVActivity are registered. These include

   - SL Training to run supervised learning training
   - AM Training to run assurance monitor training
   - Resonate Training for offline design time estimation of conditional probabilities and coeffients for risk analysis.
   - Bluerov_sim to run BlueROV simulations

2) *Activity* models and *ActivityInit* plugin

In the ALC Models under *ALC/Construction*, the *DataCollection*, *Training* and *Testing* models include *Activity* models that can be executed based on their *Activity Definition*.

To setup a new *Activity* instance,

- **Click into one of three models (*DataCollection*, *Training*, *Testing*), say *Testing*** and dragndrop the *Activity* model from the Part Browser on to the Canvas
- Click into the empty Activity model and start the ActivityInit plugin by clicking on the play button on the top-left corner of the menu.
- Known Issue: Sometimes this plugin throws an exception complaining about a missing library. You should be able to re-launch the plugin.
- The plugin presents dialog boxes on which ActivityDefinition needs to be initialized. Choose one and submit.
- **It is possible that another dialog related to the Activity Definition is presented. The user is asked to choose a specific** See the BlueROV_sim activities described below.

3) Update Activity Model

Once the *Activity* model is initialized, with default settings, the user may update the parameters or run with default parameters. In some cases the user needs to set inputs as well (e.g. training data etc.)

4) LaunchActivity Plugin

Once the parameters and the inputs are set, the user may execute the activity by clicking on the LaunchActivity plugin (play button on the top-left corner of the menu).

If a "setup and build repo" dialog is presented, the user may leave it at its default state.

Once the activity is submitted, shift to the *Artifact Index* view and a row is added for the submitted activity.

As explained in the section getting started with ALC, the colors represent the state of the execution.

In simulation scenarios (BlueROV_sim activity), when the link is blue, click on the "Sim" link to view the RViz output from the simulation.

7.1 ALC Core activity models.

This section details some of the activity models that are made available as part of the core-ALC activities.

1) SL Training

This activity is meant for training Keras-based LEC models. Examples of this model can be found under ALC/Construction/Training in BlueROVActivity project.

The parameters of interest here correspond to training these LEC models. These include

- LEC Model : Definition/ Code corresponding to the LEC model.
- **DataProcessing / formatter: Code corresponding to formatting/ normalization of the LEC input** as well as any formatting for the ground truth (output) during training. This also includes the topics of interest to be loaded from the bag file and shape of the inputs
- DataProcessing / dataset_name: You can choose from existing data loaders for bag files and csv files. Alternately, this can be set to custom, and you can specify the loader function in the custom_loader field.
- DataProcessing / custom_loader: If dataset_name is set to custom, this field should include a definition of a Torch DataSet class to load the data.
- DataProcessing / input_topics, output_topics may be left blank as they are specified as part of the formatter.

- Execution/ timeout: Maximum time after which the training is terminated.
- **TrainingParams: This includes parameters that are used to configure the Keras training session.** This includes parameters such as epochs, batch_size, loss, metrics optimizer, callbacks. This also includes parameters that sets the amount of data to use (useful_data_fraction) and as well as the portion of this data that is to be used for training (training_data_fraction)
- TrainingParams/callback: If this is set to custom, the user should fill the callback function in the LEC Model definition.

Inputs to this activity include

- **ParentLECModel** [Reference to the weights of a previously trained session of the same model.] If specified, the LEC training starts from these weights.
- TestData : Data set to be used for testing.
- Training Data: Data sets to be used for training
- Validation Data: Data sets to be used for validation. If validation set is empty, then it is set based on reminder of the *training_data_fraction*.

2) AM Training

This activity corresponds to the training of the assurance monitors.

The user choses which type of assurance monitor is to be executed from the list of - VAE, VAE_w_Regression, SaliencyMap, SelectiveClassification, SVDD.

Thereafter the user may update the Assurance Monitor template model based on the input size, the embedding size, size of hidden layers etc.

The user also configures parameters for DataProcessing similar to the ones mentioned in SL Training. These include providing the formatter, dataset_name, custom_loader.

For the training parameters, the user can twaeak the parameters in the AMTrainingParams.

Parameters of interest here are

- training_data_fraction: specifies the fraction of training data to be used. Rest is used for validation if validation data is not specified.
- parameters related to training session such as number of epochs (num_epochs), milestone epochs, random seed (rng_seed), learning rate (lr) etc.
- epsilon, and Window size: assurance monitor detection parameter parameters.

Inputs to this activity include

- LECModel : Reference to the trained LEC model for which the assurance monitor is being built.
- CalibrationData : Data set to be used for calibration i.e. computing non-confirmity scores.
- Training Data: Data sets to be used for training
- Validation Data: Data sets to be used for validation. If validation set is empty, then it is set based on reminder of the *training_data_fraction*.

7.2 BlueROV Simulation activity

Refer to the section *BlueROV2 Activity Definition* for a brief description on setting up and running bluerov simulation as activity model.

Note: The release was succesfully tested on GTX10xx (Pascal), RTX20xx (Turing), RTX30xx (Ampere) family. Nvidia GPUs in the RTX30xx (Ampere) family is compatible with Tensorflow 2.6.2 but not with PyTorch version 1.10 used in the toolchain. To use the toolchain with Ampere or newer GPUs please use the *use_cuda=false* argument in the activity. This way tensorflow will still use cuda, but Pytorch will run on CPU.

CHAPTER

EIGHT

SYSTEM MODELING

System modeling provides mechanisms to represent components and their interfaces, define ROS message types used to communicate between components, compose a collection of system architecture models, configure specific model instances for testing (assemblies) and represent the environment in which the model operates within by using world models. Fig. 8.1 shows how these elements work together to create a system model. On the right are the ALC Toolchain blocks that represent each of these modeling elements. Components are abstract building blocks of the system architecture, and may be hierarchically composed into complete system architecture models. Message types are similar to C-style data structures, and are defined in a message library. During the design of a CPS, there may be multiple implementation options for a particular component which fulfill the component's functional requirements. There may also be various component combinations or architecture models that can accomplish the similar functionality or makeup system options available for exploration. Assembly models refine an architectural model by selecting a concrete implementation from the available options for each component. World models represent the CPS system operational environment, allowing designers to specify specific configurable parameters which can be adjusted during the execution of the model. The ALC toolchain is intended to be utilized for both simulated and real-world experiment environments.

8.1 Message Types

The message library defines the structure of all ROS message types used in the system. Fig. 8.2 shows the messages available in the tutorial with the available modeling elements on the left side of the image. Message types can be **topics** (publish/subscribe), **services** (synchronous request/response) or **actions** (asynchronous interface with goal, feedback, and result). Each message type is named for both the ROS package where it is defined and the data it represents. Hierarchy in the package structures can be indicated in the naming layers provided or could be visually grouped using meta packages (**Package**) to allow collapsible grouping of like messages.

ROS topics for publishing and subscribing are defined as **MessageType** and the message is outlined as a structure of data elements, shown in Fig. 8.2. Message type structures can be defined by specifying other messsage types within a new message type, as shown by the use of `geometry_msgs/PoseWithCovariance` in the `nav_msgs/Odometry` definition. Constants can also be defined within the message type definition, see `sensor_msgs/Range` example in Fig. 8.3 where the enumerated options are `ULTRASOUND` and `INFRARED`.

While services are synchronous, actions are asynchronous. Services (or **ServiceType**) are defined by inputs/outputs which are separated by a **—** line, as shown by `gazebo_msgs/GetModelState` message in Fig. 8.4.

Similar to the request and response of a service, an action (or **ActionType**) uses a goal to initiate a behavior and sends a result when the behavior is complete. But the action further uses feedback to provide updates on the behavior's progress toward the goal and also allows for goals to be canceled. Actions are themselves implemented internally using topics. It is essentially a higher-level protocol that specifies how a set of topics (goal, result, feedback) should be used in combination. Each section is separated by a **—** line, as shown in the `nav_msgs/ActionExample` message in Fig. 8.5.

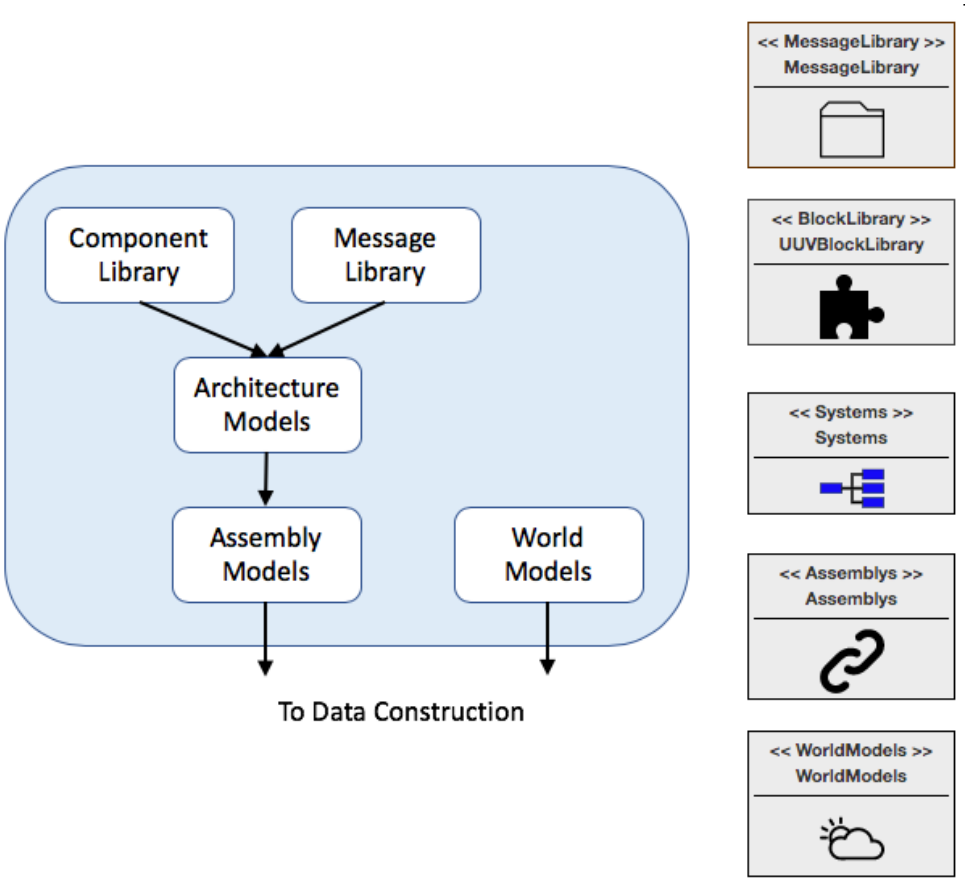


Fig. 8.1: System Modeling Elements

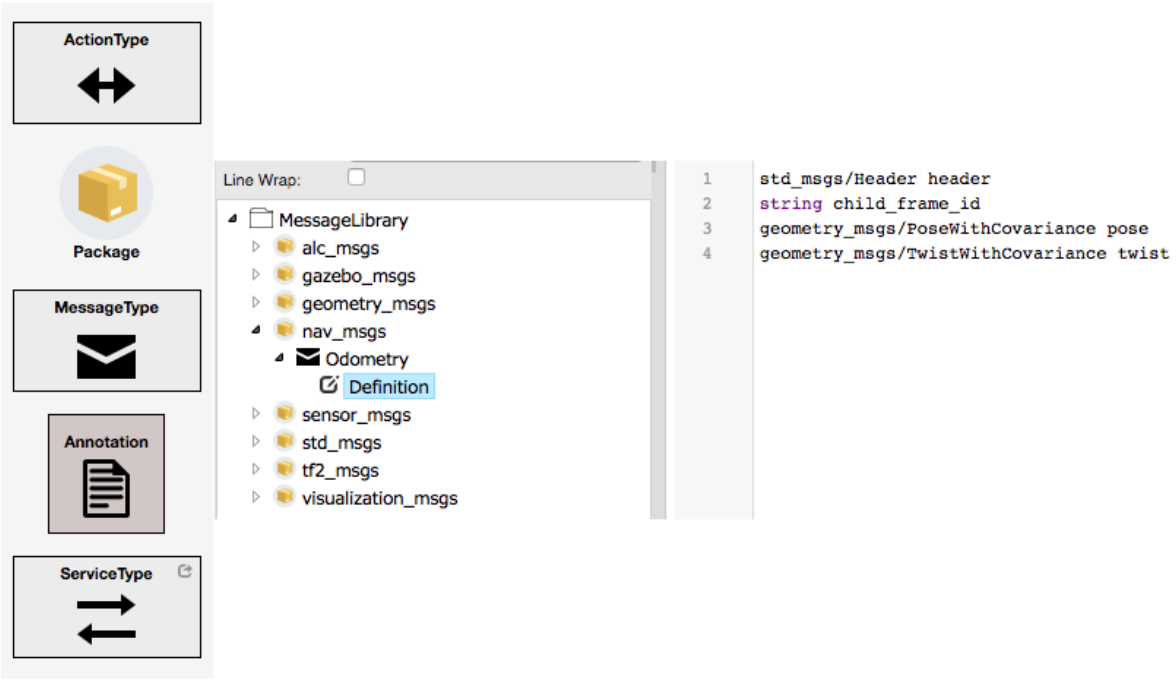


Fig. 8.2: Message Library with MessageType Example

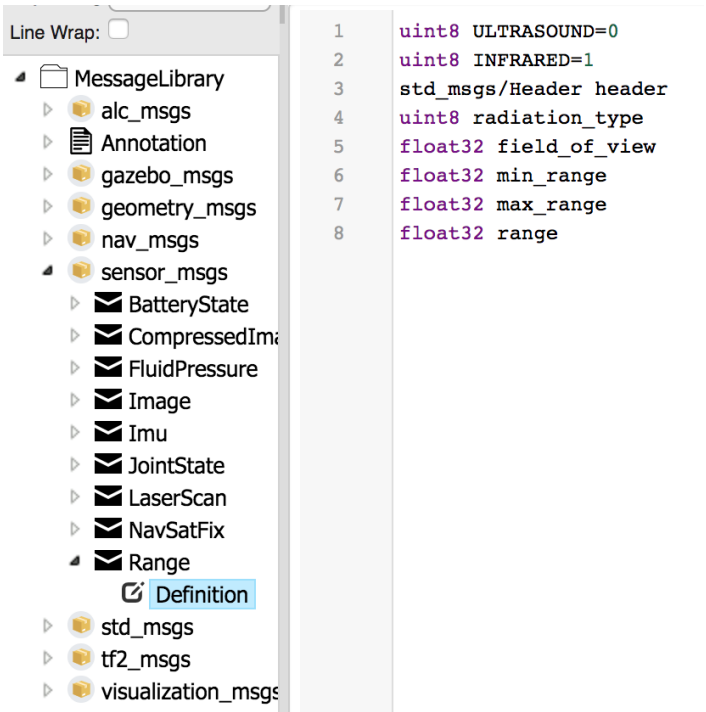


Fig. 8.3: Example of Constants in Message Definition

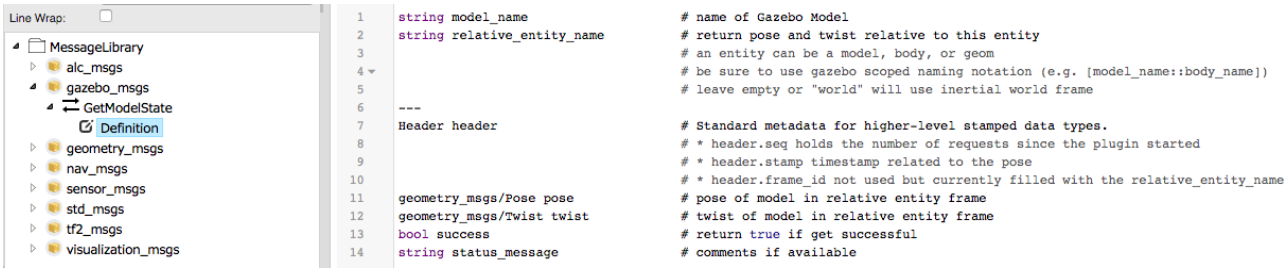


Fig. 8.4: Example of Service Message Definition

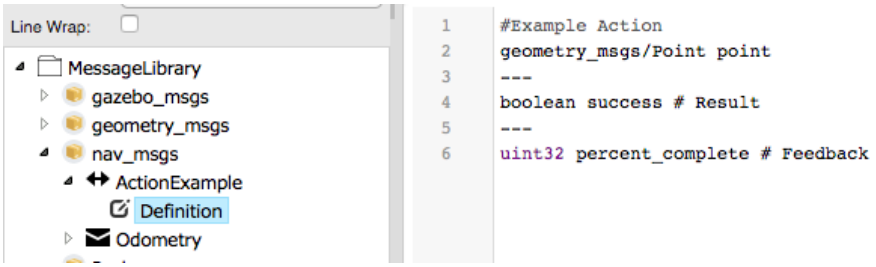


Fig. 8.5: Example of Action Message Definition

Note: The F1 Tenth tutorial model does not have any action messages.

Messages and their packages can be added to the library by selecting the **Designer** view (top left section titled Visualizer Selector). When in the **Designer** view, the available model elements on the left side (i.e. **Package**, **MessageType**, . . .) can be dragged into the model window. Fig. 8.6 show the current message packages available in the tutorial (also shown in Fig. 8.2). The package name is defined in the *name* attribute located in the bottom right panel (the **property editor**).

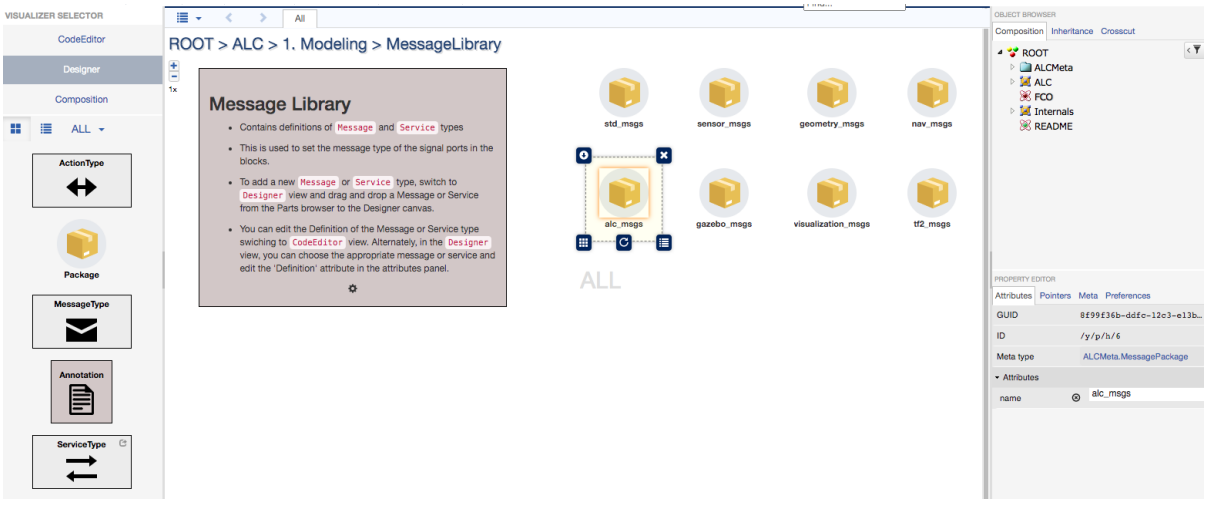


Fig. 8.6: Designer View of Message Packages

Once the desired packages are setup, drill down into a package (by double clicking on the package) to add the desired messages by adding a message element (**MessageType**, **ActionType** or **ServiceType**) and naming each message (using the *name* attribute). Fig. 8.7 shows the messages setup for the *alc_msgs* in the tutorial. Then the message format can

be defined using the **CodeEditor** view (from the top left Visualizer Selector panel), as shown in Fig. 8.8.

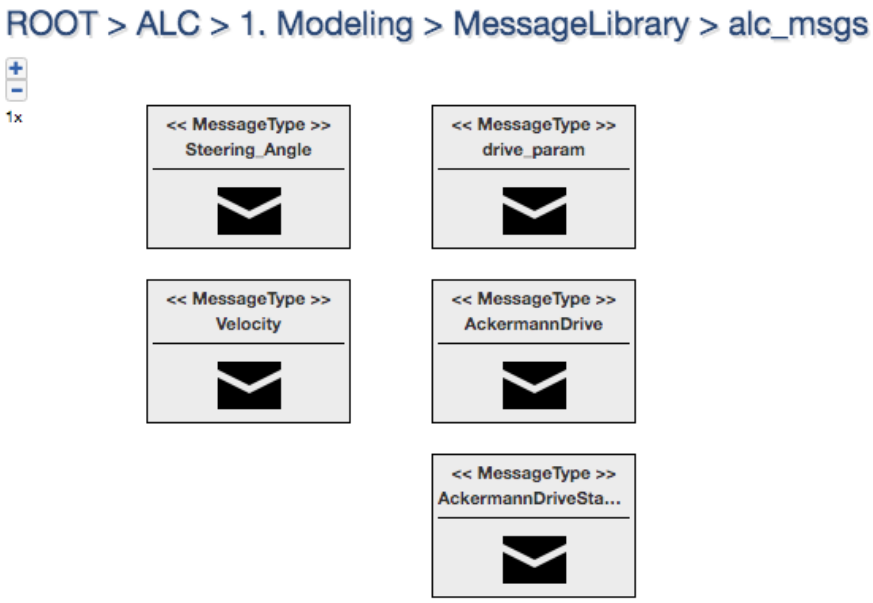


Fig. 8.7: Designer View of Message in a Package

8.2 Block Library

Components are first defined in a block library before instances of a component can be created in a system model. This approach promotes reusability and maintainability of components across many system models. Blocks can represent both hardware and software components. Subsystems can be created using multiple lower level library components to represent logical groupings of components within the system. There may also be one or more concrete implementation alternatives available to fulfill a component's functional requirements, which will be represented as different library blocks.

To assist in the top level organization of the block library, a **Package** should be created to combine similar items with project appropriate naming. For ROS software in the system, these packages will be utilized to form the catkin build packages. If no package is defined, the catkin build package name will be the project name. For the car tutorial, there are three packages setup as shown in Fig. 8.9. The `tutorial_blocks` package will contain the blocks the user will develop to build the system. The `f1tenth_car` package contains open source third-party blocks utilized by the system. The `alc_ros` package includes blocks that assist the ALC toolchain in providing housekeeping actions or results back to the user, these items will be utilized by the tool to provide support for controlling experiments and data collection methods.

Component interfaces are defined using directional **ports** which can represent **material** (m), **power** (p) or **signal** (s) flows as shown in Fig. 8.10. Materials ports represent the material aspects, such as representing the structural coupling between components. Power ports represent energy ports, where connection between ports represent energy propagation. Signal ports are used for digital signals transmitted between blocks, which can be software or hardware signals. A `Direction` attribute (found in the Property Editor) provides an indication if the port is an `Input`, `Output` or `Input/Output`. The `name` attribute allows indication of the ports functionality within the system.

Software components use directional signal ports to model data flow between components, and each signal port produces or consumes a particular message type defined in the message library. For software components, implementation models contain the information necessary to load, configure, and initialize the software `nodes` when the system is

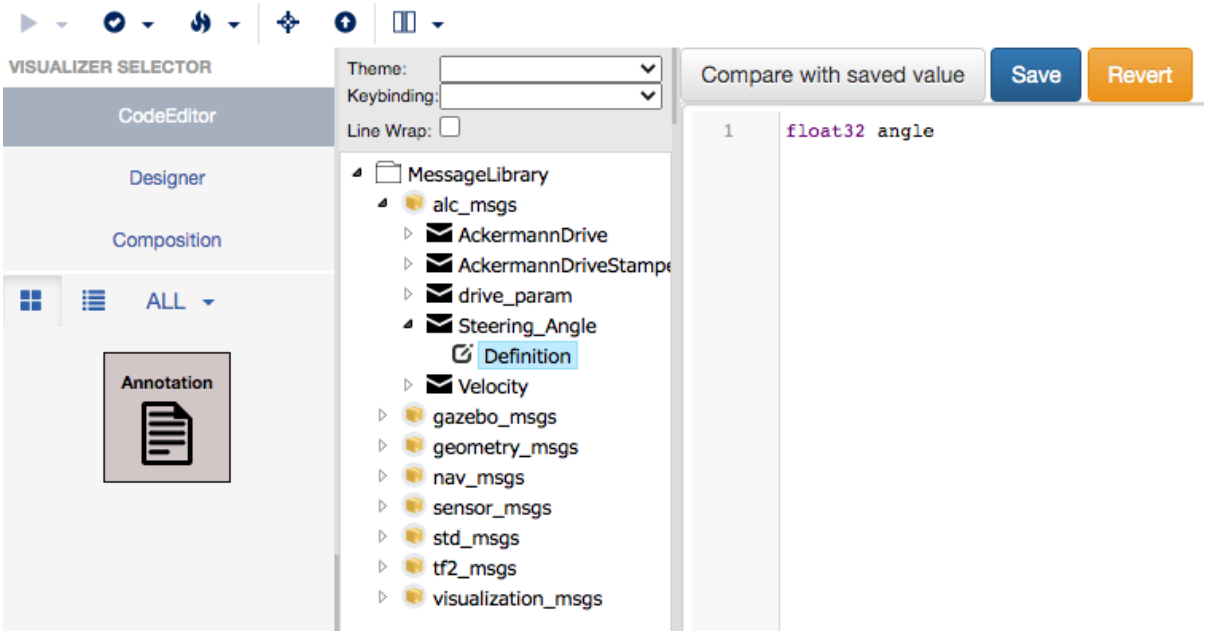


Fig. 8.8: CodeEditor View of Message Library

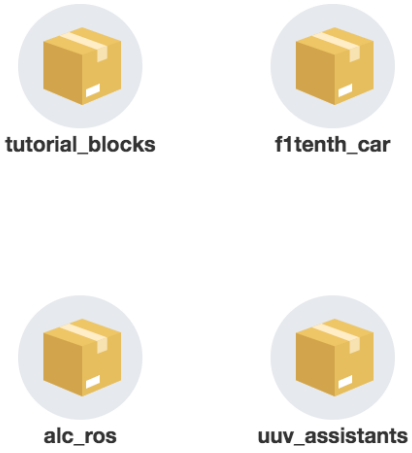


Fig. 8.9: Tutorial Model Block Library Packages

Fig. 8.10: Available Block Port Types

deployed, as well as the business-logic associated with its runtime implementation. More on signal port definitions in the *Signal Ports* section.

Each component **Block** can be labeled with a descriptive name. An example block is shown in Fig. 8.11 where the *Perception* block is shown at the top, the contents of this block is shown below it and the available modeling elements are shown at on the right. Typical elements of a block definition are input and output ports (material, power or signal), any sub-blocks used to further breakdown the block element, specification parameter values outlined (Params) and an annotation block for documentation of the component design information. Also available are specific modeling objects for the LEC models, the ROS software elements and the ability to attach an informational file from other tools to help users identify the object's interfaces.

Note: It is good practice to add an **Annotation** block throughout the project model to help other users understand quickly what the particular section is addressing and any special assumptions made in during the design cycle. To add text to the annotation block, click on the gear icon located in the top right of the block to bring up an editing window. Text can be entered in markdown syntax. There are icons in the lower right of the edit window that allow the user to see previous of the text as it will appear in the block (full screen allows side-by-side editing and previous screens).

A **Block** role attribute assists the user in understanding the block's functionality. The table below outlines the different roles available for blocks. The possible sub-block column indicates in which block roles can exist within that role type. When a block instance is created the role defaults to `Block`.

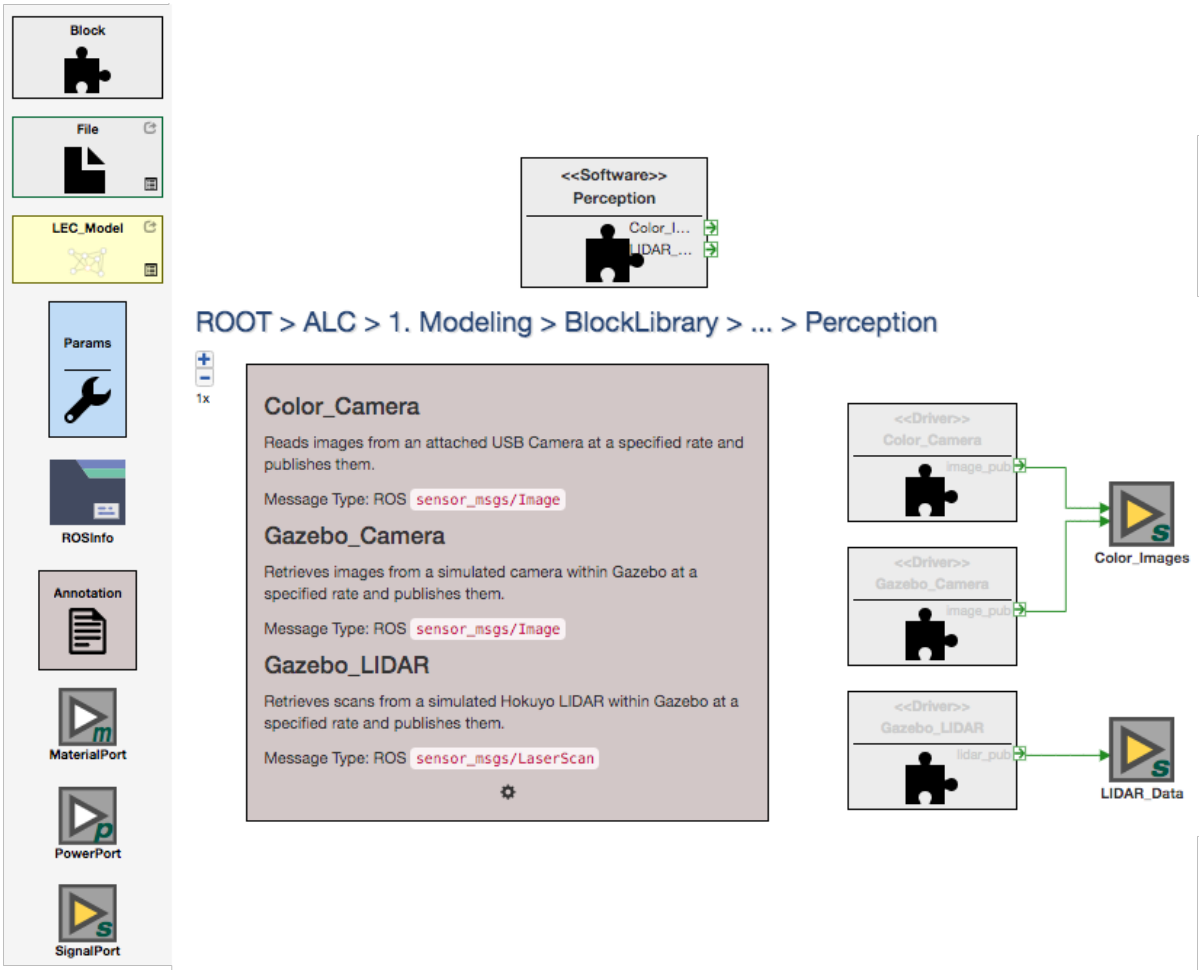


Fig. 8.11: Example of Block Library Elements

Block Role	Description	Possible Sub-blocks
Block	Component or subsystem blocks (to reduce visual complexity)	Block, Hardware, Software, Simulation, Node, Driver
Software	Generic block that signifies a grouping of software elements	Block, Software, Simulation, Node, Driver
Hardware	A single or group of lower level hardware components	Block, Hardware
Simulation	External simulation software that is providing or accepting information to the system	Node, Driver
Node	ROS Nodes available to be deployed	Module
Driver	Driver component connecting ROS node to non-ROS blocks	Hardware (instance), Module
Module	Common software elements used by various implementations components	None

The `Block` role is a generic block to allow the users to place context on the segmentation of a system into sub-blocks. It allows the user to create logical groupings in their block library that can be utilized in building the system models. The `Software` role is also a high level block elements for a collection of software related blocks. Upper levels of system architecture and assembly models often utilize this role. `Hardware` blocks can be either a base level block describing a hardware component or a container grouping multiple hardware blocks to provide a logical grouping. Below in Fig. 8.12 is an example of a software block (`Drive_Control`) and inside the block's layout which shows ROS nodes that are available to drive the car. How these blocks are used will be discussed more in the system architecture and assembly sections below.

The `Driver` block provides a way to define blocks that interface pure ROS nodes with hardware and simulation blocks within the system. This block will be launched as a ROS node during experimentation, but unlike Node blocks

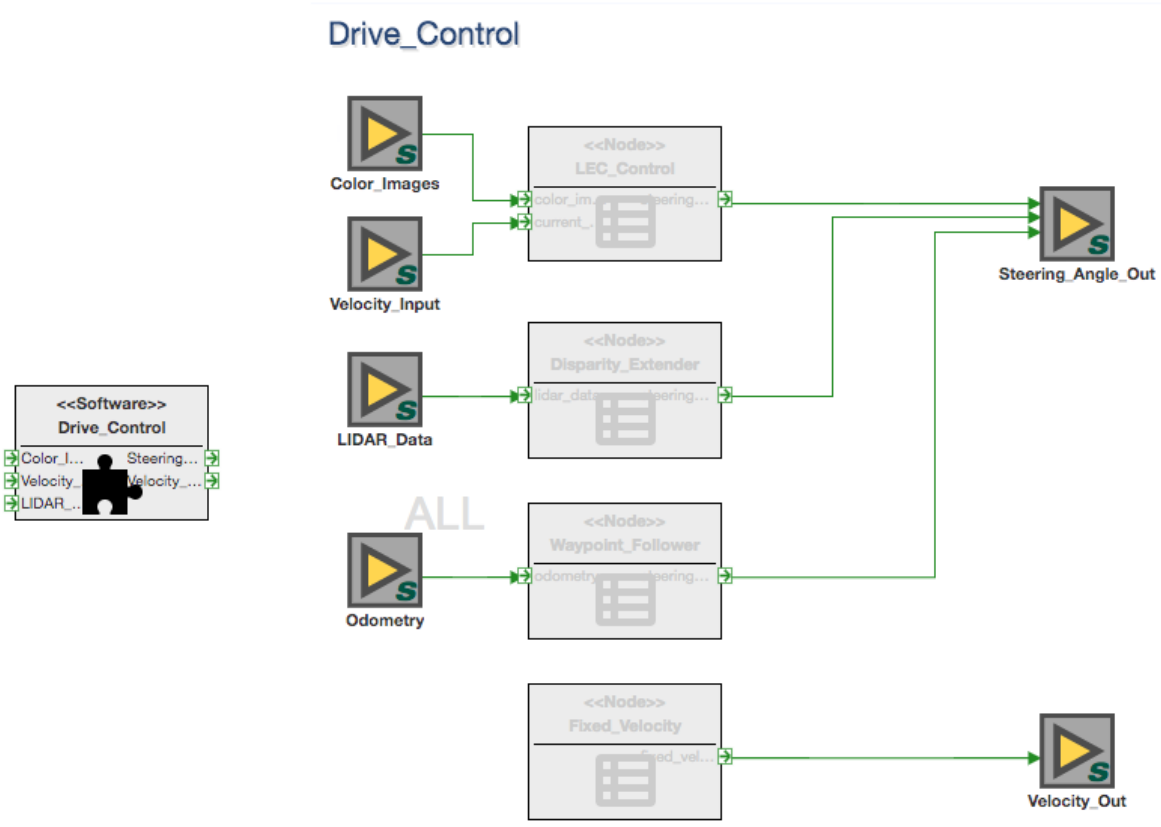


Fig. 8.12: Software Block and Subblock Example

it can contain both ROS and non-ROS signal ports. `Hardware` blocks included within the driver block should be an instance of a hardware block already defined in the block library. `Simulation` blocks are a collection of blocks that represent functions only needed in a simulation environment (such as recording nodes), blocks with simulation environment configuration information (such as world models) or blocks to provide access to simulated objects (such as gazebo plugins). `Module` block allows the developer to breakdown a node or driver block to smaller parts for documentation of the data flow and/or state machine blocks. A common usage of Module blocks would be to represent python code components that help the user understand more about the upper level node or driver block.

For this tutorial, the Gazebo interface is a `driver` block. Fig. 8.13 shows is an example of connecting the car model with a ROS based simulation tool using this `driver` block. The figure includes the hierarchical layers, starting from the top. The `Gazebo_Connector` block will be launched as a ROS node in experiments with the message interface to the rest of the car model as shown by the signal ports. The `Interface` block (in the middle) shows the Gazebo specific messages that are provided from Gazebo, but are not utilized by the car model. Since these interfaces are to an external simulator, the role is `simulation`.

The above `Interface` block could also have been a hardware instance to show how information from ROS messages can be consumed or provided by hardware sensors (or other messaging types, such as ZeroMQ). In the `f1tenth_car` package from the car tutorial, there is a hardware block called `PWM_Applicator` shown on the left side of Fig. 8.14. The right side of the figure shows the signals that make up the block inputs and outputs. A third-party software module is utilized by this component to handle the hardware communications, which is represented by the `Module` block called "PigPio".

`Hardware` components can be associated with models from other tools, such as CAD models. Software related blocks can also include files, such as relevant code and configuration files. These files can represent information helpful to the user in understanding the component and the design parameters utilized for configuration, **Files** can be uploaded as part of the component block library representation using the `Artifact` file uploader on the property editor attribute tab. File content can also be created and saved in the model using the `Edit content ...` option. If a file is located in a different location from the model, the user can indicate the file location and name. File categories can be the following, with associated file types: `Configuration` (JSON, XML), `Data`, `GazeboModel` (GazeoXARCO), `GazeboWorld`, `GeneralArchive`, `LEC Model`, `Make` (CMake), `ROSLaunch`, `ShellScript`, `SourceCode` (C, CPP, Python, ROSLaunch) and `Others`.

8.3 Signal Ports

Connection of signal ports show the flow of information from one block component to another and assists the user in understanding how information is passed across the system. For ROS software components, these signals are messages passed to relay desired information to other interested components or invoke a reaction from another component. The available messages are defined in the Message Library and are utilized by the ports. Non-ROS signals are informational ports to help show system connection points, and may be found within software, hardware, simulation, driver or module blocks.

Port attributes defined from the block library level are **Port Name**, **Direction**, **PortType** and **Message Type**. **Direction** can be `Input`, `Output`, `InputOutput` and is useful for both hardware and software components (found in the attributes in the property editor). The port type, message type and topic attributes describe the messaging behavior for ROS signals, these have no meaning for material, power and non-ROS signals. **PortType** indicates the type of ROS message used: `Action Client`, `Action Server`, `Publisher`, `Service Client`, `Service Server`, `Subscriber` and `Signal`. Blocks defined as driver can have a combination of non-ROS and ROS signal port. Any non-ROS signal port should have the type set to `Signal` so that they are not considered ROS message ports and their direction will need to be identified. **MessageType** links the port message back to the message type defined in the message library. **Topic** is typically defined at the system or assembly level when the developer knows the desired message naming, but a default or generic topic can be setup in the block model if it is expected to be common across models. The port information is summarized in a **ROSInfo** visualizer that can be accessed within the definition of a node (in the upper left panel), shown in Fig. 8.15. This visualizer allows the developer to modify ROS signal messaging behavior from a broader perspective than at each port level. `Type` and `Message Type` columns will

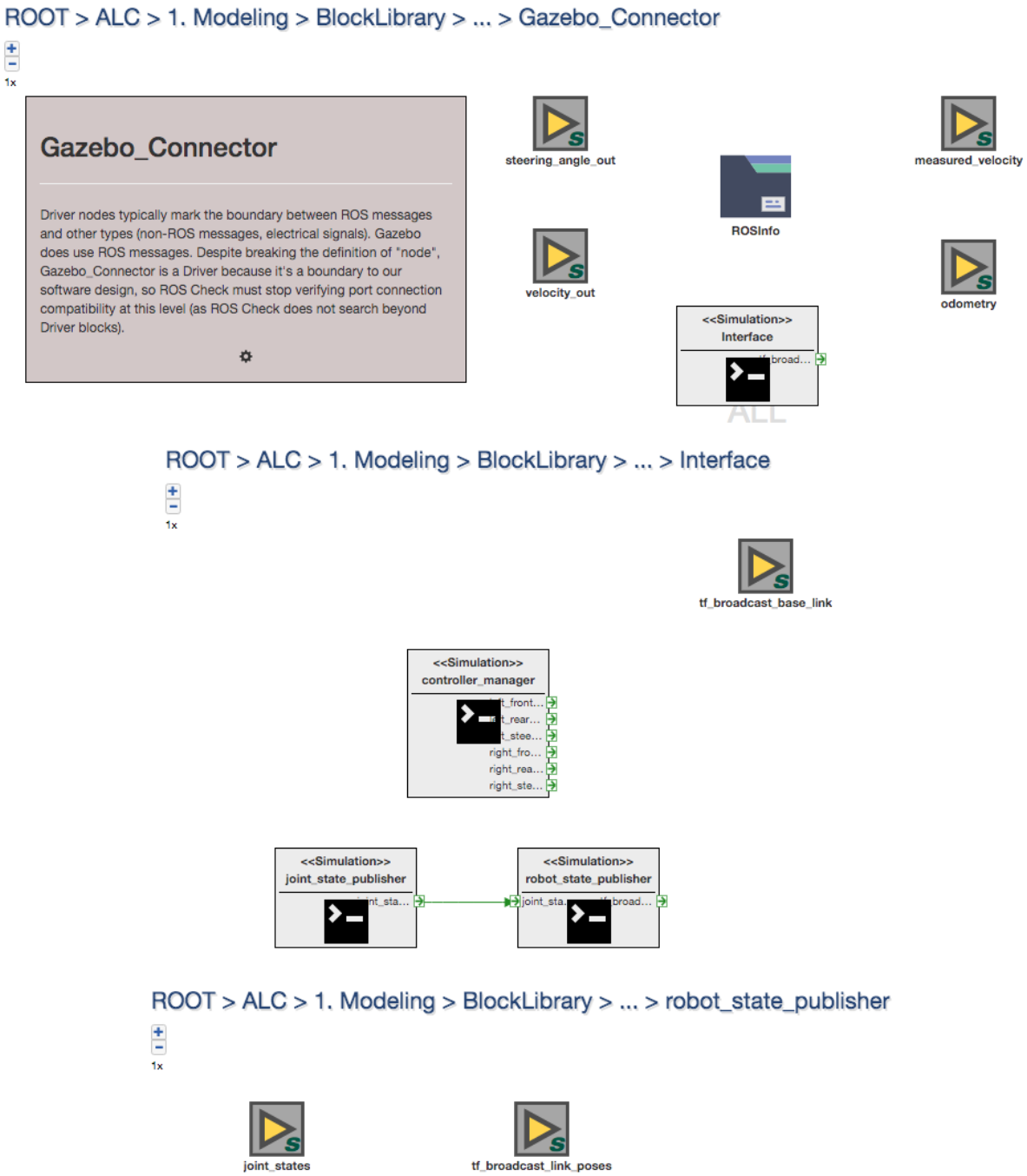


Fig. 8.13: Driver Block and Subblocks Example

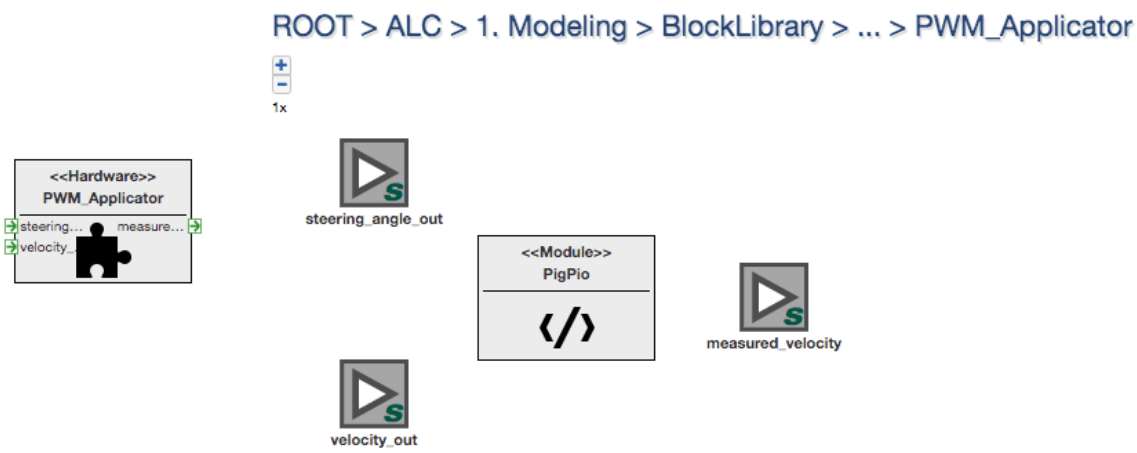


Fig. 8.14: Hardware Block Example

provide the available options to select, while the `Topic` column allows text entry of a value. The `Message Type` shows all the messages previously defined in the message library. This same ROSInfo visualizer is also available at the higher system levels to help users see ROS message across multiple nodes, providing a view of communication pairs. As indicated, topics are typically filled in when developing the system model, where the design becomes tailored to the specific system solution.

ROSInfo

Driver in : Drive_Actuation/Physical_Car_Connector

Port Name	Type	Message Type	Topic
measured_velocity	Publisher	M :alc_msgs/Velocity	measured_velocity
steering_angle_out	Subscriber	M :alc_msgs/Steering_Angle	steering_angle_out
velocity_out	Subscriber	M :alc_msgs/Velocity	velocity_out

ROS Interfaces in : Drive_Control/Disparity_Extender

Port Name	Type	Message Type	Topic
lidar_data	Subscriber	M :sensor_msgs/LaserScan	lidar_data
steering_angle_out	Publisher	M :alc_msgs/Steering_Angle	steering_angle_out

ROS Interfaces in : Drive_Control/Fixed_Velocity

Port Name	Type	Message Type	Topic
fixed_velocity	Publisher	M :alc_msgs/Velocity	velocity_out

Fig. 8.15: Example of ROSInfo Visualizer

Note: Orientation of the signal ports on the block at the next level up can be affected by the location of the port on the page. `Input` ports should be located in the left half of the screen, while `output` blocks should be located in the right half of the screen.

A **ROSCheck** plugin is available at the node level which utilizes the model information to find where connections may

have concerns for the developer to resolve. This plugin can be found in the upper left under the play options. Once the check completes, the **ROSInfo** visualizer will include a `Concerns` column when issues need to be addressed in connecting up the ROS nodes. If ROSInfo view is open when running this check, move to the Designer view and then back to the ROSInfo view to refresh the input and see the results of the ROSCheck plugin. The example in Fig. 8.16 shows a mismatch of port type and message type (`steering_angle_out` and `measured_velocity`), along with missing topics which are expected since this check was performed at the block node level. Once issues are resolved, the user should rerun the ROSCheck plugin to verify no issues exist.

ROSInfo

Driver in : Gazebo_Connector

Port Name	Type	Message Type	Topic	Concerns
measured_velocity	Publisher	--NONE--	*None*	Topic is not set For a publisher/ subcriber, the message type should be a MessageType object
steering_angle_out	Service Client	M :alc_msgs/Steering_Angle	*None*	Topic is not set For a Service Server/ Client, the message type should be a ServiceType object
velocity_out	Subscriber	M :alc_msgs/Velocity	*None*	Topic is not set

Fig. 8.16: ROSInfo Visualizer after ROSCheck

Note: ROS is particular about naming of package, node, messages, topics and arguments, please refer to the ROS Naming conventions.

8.4 Parameters

Setup parameters (Params) can be created in any block type to allow indication of the block's specifications that might be tweaked when selecting block components. A functional block may have several implementation options to meet the desired requirements. One way to distinguish between different available options could be to setup a parameter table of the key requirements that will vary and set specific values that meet the desired implementation option. The model designer view will show the list of parameters available for the specific block, as seen in Fig. 8.17. The parameter values are listed as JSON dictionary items with a parameter-value pair. The value is represented by a string value. To modify the available parameters and/or value, drill down into the **Params** block and edit (or add) the parameters in the table, as seen in Fig. 8.18.

8.5 ROS Nodes and Drivers

In addition to defining the signal ports and parameters for ROS nodes, launch information and component code should be defined by the user. There are a couple of ways to define this information in the model: using ROS Generator plugin to create template code based on current model information or importing existing launch files using the ROSLaunchImporter plugin. Fig. 8.19 shows a `Node` block from the tutorial called `Gazebo_Camera`.

The signal port connection knowledge of the model can be leveraged to generate executable ROS launch deployment file and a skeleton ROS component code by running the **ROS Generator** plugin (found in the top left play button options, shown in Fig. 8.20). Before running the plugin, the user needs to identify the ROS signal ports, add a ROSInfo block and layout the expected launch parameters (if any). Launch parameters can be added by selecting the **ROSInfo** block and choosing the **ROS Param Info** visualizer (in the left top panel) to add parameters in the same way as the *Parameters*. An example ROS Param Info table is shown in Fig. 8.19. Then the ROS Generator plugin can be used to create a ROS launch file (launchInfo) and component source code (`srcComponentImpl`). The ROS

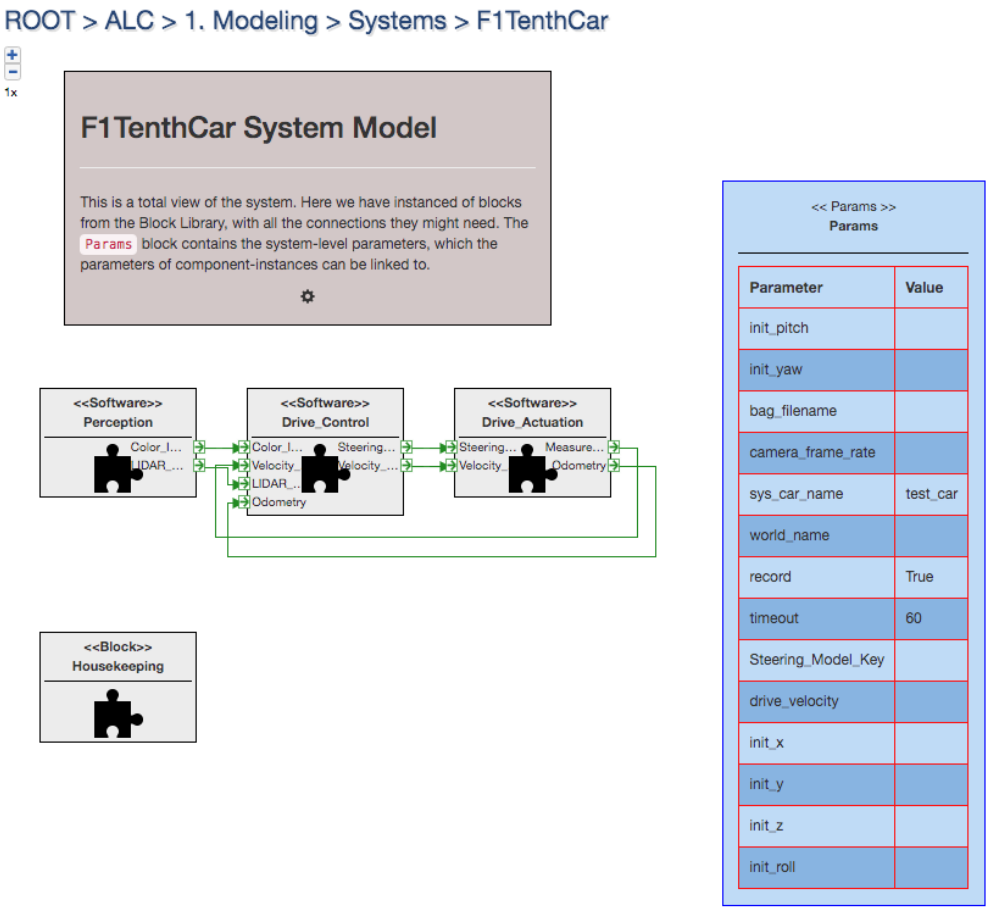


Fig. 8.17: Parameter Table Example

Params

Filter..

Name	Value	Actions
bag_filename		
camera_frame_rate		
drive_velocity		
init_pitch		
init_roll		
init_x		
init_y		
init_yaw		
init_z		
record	*True*	
Steering_Model_Key		
sys_car_name	*test_car*	
timeout	*60*	
world_name		

Fig. 8.18: Parameter Info Editing

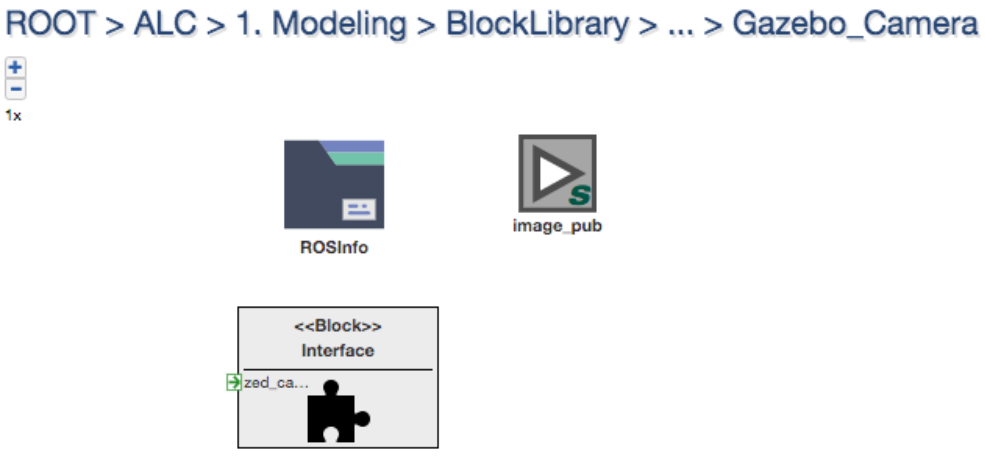


Fig. 8.19: ROS Node Block Example

parameter information will be reflected in the launchFile with **arg** tags. Specific **Namespace** could be specified in the Attributes section of the ROS Info block (shown in Fig. 8.22) and will be reflected in the generated launch file. The **Package** name will be based on the package this node was put into at the top level of the block library. The **Type** attribute indicates the component code filename that will be used in the launch file. The ROS generator creates a filename based node block name, appending it with `_impl.py`. The **LaunchFileLocation** will provide information needed to create system level deployment launch files for each ROS node, knowing the package name and the node block name.

Fig. 8.20: ROS Code Template Generation Plugin

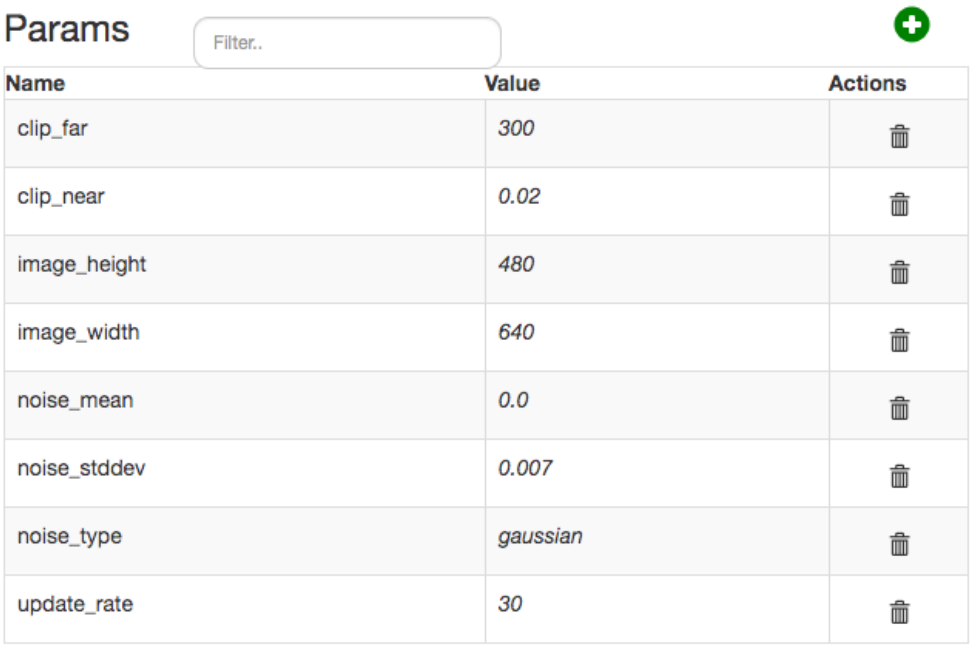


Fig. 8.21: ROS Param Info View

The user can then add the behavioral code needed for their system implementation using Python 2.7. Protection markers are available in the code to allow code generation updates while protecting the user specific code. These markers are noted by `#**********protected region ... begin **********#` and `#**********protected region ... end **********#` comments in areas where additional code is expected to be placed, such as package imports, initialization, callbacks and update functions. Additional functions can also be placed within protective markers.

Fig. 8.22: ROSInfo Block Attributes

The **UpdateFrequency** attribute in the ROSInfo block can be set greater than 0 Hz when a time-triggered behavior is desired, instead of the default event-triggered behavior. Then when the ROS Generator plugin is used to create the boilerplate code, it will add an update function.

If a launch file already exists, the **ROSLaunchImporter** plugin (found in the top left play button options) can be used to create the ROSInfo block. The user will upload the existing ROS launch file for the node or driver, then needs to indicate the location of the existing ROS launch file in a ROS-readable form (i.e. $(find arospackage)/launch/afile.launch). ROS launch arguments (**ROSArgument**) will be created based on the **arg** information in the launch file. When running the ROSLauncherImporter, there is an option to import this launch file as a parameter block. This will import arguments from the launch file and place them in a parameter block instead of a ROSInfo block. This option is typically used for system-level parameters shared between many components.

Some systems utilize a hierarchical approach to ROS node and driver parameter definition. The ROS node and driver level parameters in the block library should setup default values for parameters that do not require system level knowledge to determine the desired value. The system level parameters can be left blank for arguments that should be passed down from the system model. A parameter mapping visualization tool (located in the top left panel) will be available to connect the system parameter to the node level parameter, this will be explained in the *System Architecture Models* section. Fig. 8.23 shows an example of a ROS node parameter set where parameters defined by the system are left blank, which will then require the mapping to occur from a top level system parameter.

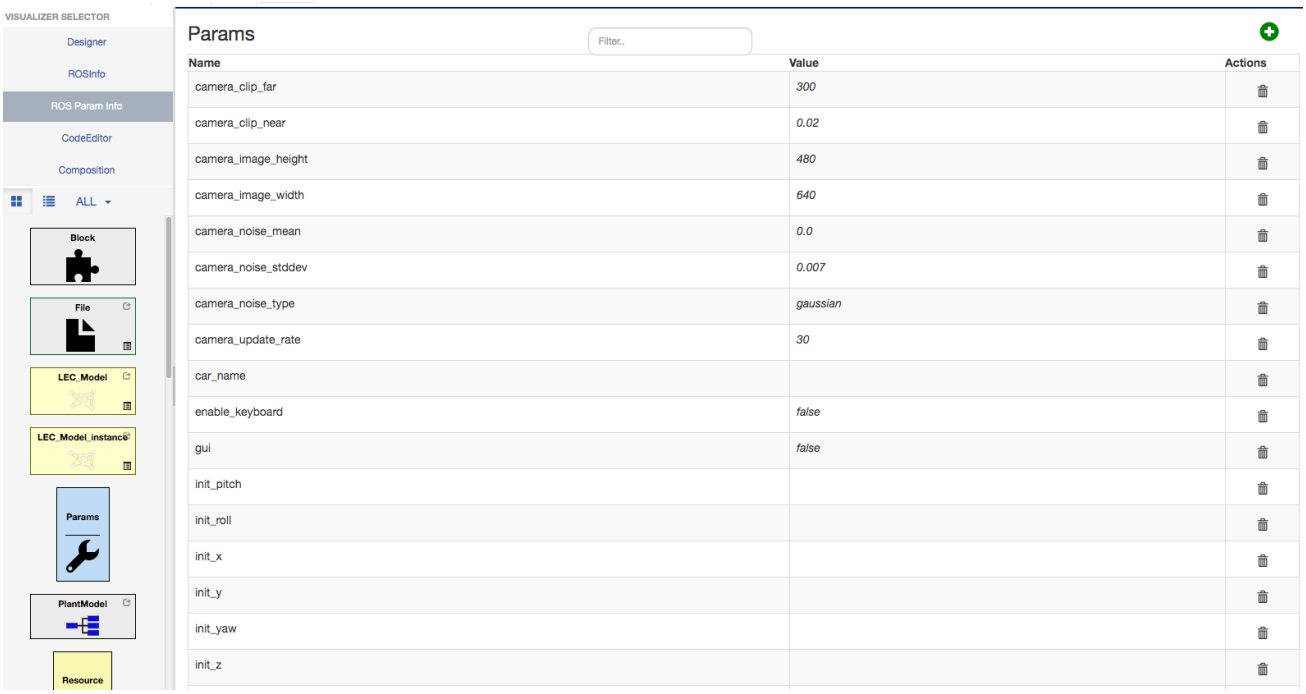


Fig. 8.23: Hierarchical Parameter Definition (Node Level)

Once all the ROS nodes and drivers are defined and ready, the ROS packages must be deployed to the execution server. Fig. 8.24 shows the tutorial `f1tenth_car` package. The **Deploy ROS Package** plugin (found in the top left play button options and shown in Fig. 8.25) can be run in the top level of a ROS package. This will copy all the code from each of the ROS nodes and drivers within the package into the `$ALC_WORKING_DIR/ros/src/$PACKAGE_NAME/nodes` directory and run a catkin build on the package. If there the package has an error during the catkin build, it will not be deployed and an error message is returned. If the package deployment fails, re-evaluate the ROS element naming convention mentioned earlier (i.e. lowercase names with _).

Note: At this time, all ROS packages from any deployed model are stored in the same location on the execution

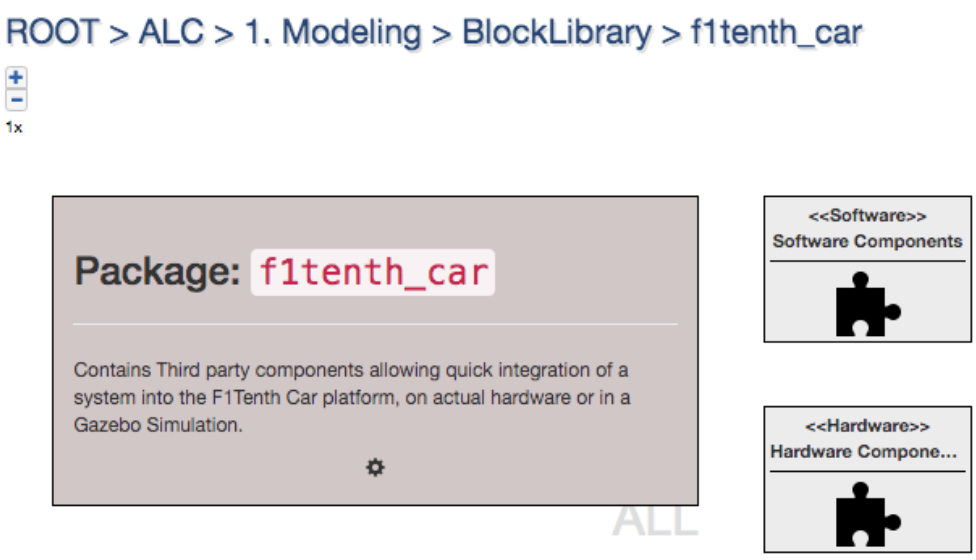


Fig. 8.24: ROS Package Example

Fig. 8.25: Deploy ROS Package Plugin

server. It is advised to take care in creating unique package and node names.

For typical activities related to simulation tasks that are not specific to a system implementation, the `alc_ros` package can be utilized, shown in Fig. 8.26. Currently this library includes ROS nodes to allow ROS bag recording (`recording`) and shutdown of all nodes after a specified timeout to allow the ROS bag recording to complete and be available for analysis (`ros_timeout`). As the toolchain progresses, these type nodes will be available in a library of packages that the user can include in their model. At this time, this package should be created and the nodes included manually.

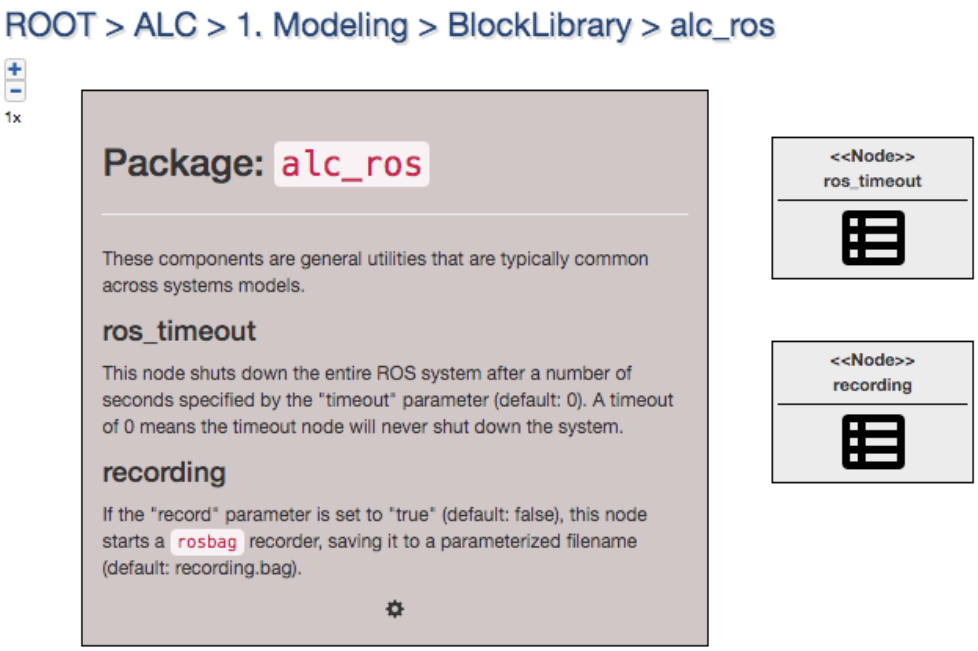


Fig. 8.26: Available Library Package

8.6 LEC Models

A sub-block defining a LEC, will contain a **LEC_Model**, any signal ports and a **ROSInfo** block, as shown in Fig. 8.27. The **LEC_Model** blocks provide an indication of where LECs are expected to be utilized in the system. The development and training of available LECs will occur in the LEC construction segment of the model where the desired LEC implementation can be chosen when configuring the specific test parameters. Any runtime execution logic associated with utilizing the LEC model in a system should be setup in a **ROSInfo** element within the `srcComponentImpl` file. This allows the user to define message handling and any other logic needed when the LEC is utilized in either a testing run or during a reinforcement learning training session.

The **LEC_Model** block defines a deployment key attribute, LEC model configuration and translation functional code. The deployment key is a command line parameter provided to an associated ROS launch file indicating the URL or directory location of the trained model to be loaded for this LEC. For the block library, the only information required is the deployment key. For the car tutorial, the attribute panel (bottom right panel) sets the deployment key to `Steering_Model_Key` as shown in Fig. 8.28.

The LEC model definition is a code set created by the user to specify a neural network using the TensorFlow and Keras libraries. Selecting or drilling down into the LEC block, the user can use the CodeEditor visualizer (on the left menu) to see and create the code set describing the LEC (shown in Fig. 8.29). While a default model definition could

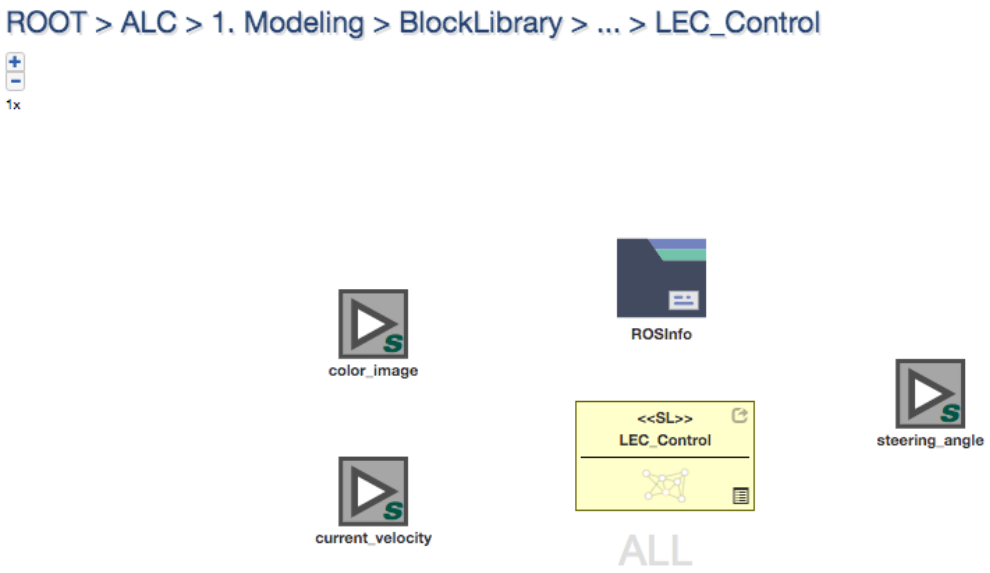


Fig. 8.27: Car Tutorial - LEC Control Block Example

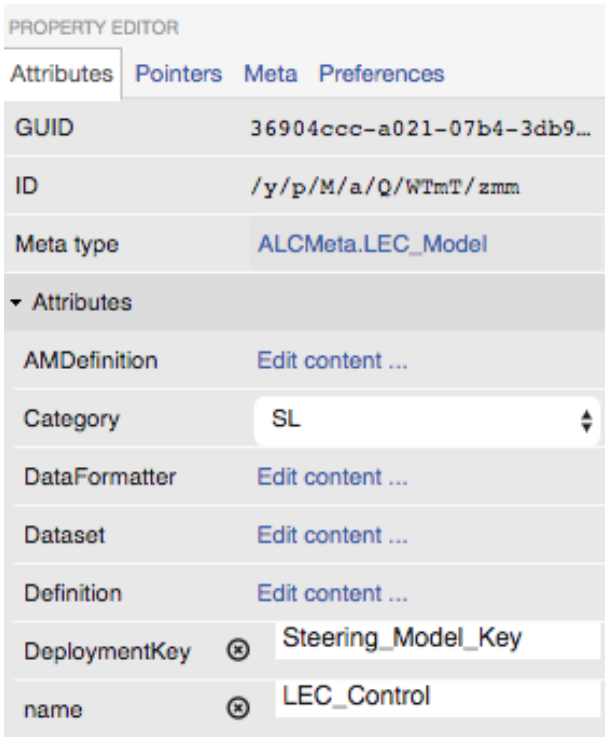


Fig. 8.28: LEC_Model Attribute

be defined in the block library, this definition is typically configured in the LEC Training part of the model and will be discussed further in that section. The translation functional code (DataFormatter and DataLoader) are also typically configured in the LEC training part of the model.

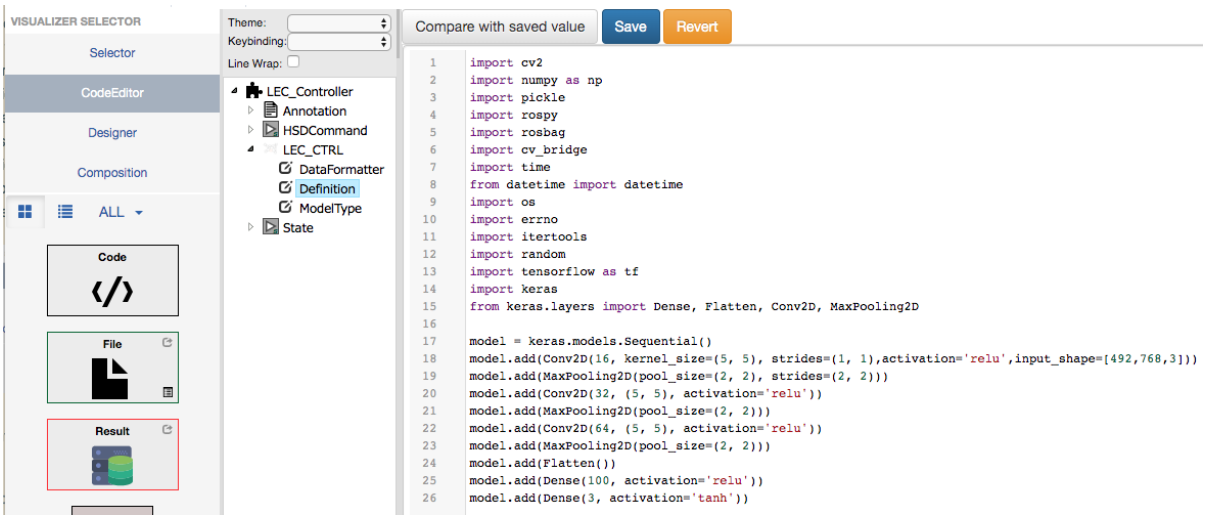


Fig. 8.29: LEC Model Definition for Supervised Learning Example

When generating code for a ROS node that has a LEC model element (using **ROSGenerator**), there are some helper functions that are added to the component code. The `init_sl` function initializes the LEC model network interface used to train the model. At each step when a prediction needs to be made, `execute_sl` needs to be invoked with the appropriate data. This execution step is typically performed within the callback for the data subscriber. In the car tutorial, the `callback_color_image` contains this execution step.

- LEC ROS Node Helper Code

```
#initialize_sl
def init_sl(self,model_folder,**kwargs):
    from alc_utils.network_interface import NetworkInterface
    self.network_interface = NetworkInterface()
    self.network_interface.load(model_folder)

def execute_sl(self, raw_input, use_batch_mode=True):
    return self.network_interface.predict(raw_input,batch_mode=use_batch_mode)
```

- Example Code to Execute the LEC Model

```
#callback for subscriber - color_image
def callback_color_image(self, msg):
    """
    callback at reception of message on topic color_image
    """
    self.in_color_image = msg
    self.in_color_image_updated = True

    #********** protected region user implementation of subscriber callback for color_
↪image begin **********#
    # 0.6108652353 is a normalization factor
    pred = self.execute_sl(msg)[0]*0.6108652353
    msg = Steering_Angle()
    msg.angle = pred[0]
```

(continues on next page)

(continued from previous page)

```
    self.steering_angle_.publish(msg)
    #********** protected region user implementation of subscriber callback for color_
↪image end   **********#
```

For reinforcement learning, the user needs to define ROS code that can utilize a LEC model in the system to provide operational states and react to requested actions and termination commands. More will be discussed about this code in the reinforcement training discussion in *Reinforcement Learning* section. While the learning model training code is independent of the component models, the ROS code is specific to the system implementation and belongs in the block library definition.

- Reinforcement Learning ROS Node Helper Code

```
def init_lec_input_output_publisher(self):
    #need one or more message types
    #self.pub_lec_input_output = rospy.Publisher(self.lec_topic_str, LEC_Input_Output_
↪Message_Type, queue_size=1)
    pass

def publish_lec_input_output(self,states, actions):
    #convert states and actions to message and publish
    # return lec_input_msg( i.e state) and lec_output_msg (i.e. action)
    pass
```

There are also routines available when an assurance monitor is trained for the system. Like the LEC, there is an `init_am` function to initialize the assurance monitor interface used to train the model. After the LEC is executed, the assurance monitor is executed using the `step_am` function. More will be discussed about how this operates in the system in the *Assurance Monitoring* section.

- Assurance Monitoring ROS Node Helper Code

```
#initialize assurance monitor
def init_am(self, model_folder):
    self.assurance_monitor_paths.append(model_folder);
    import alc_utils.assurance_monitor
    self.ams = alc_utils.assurance_monitor.MultiAssuranceMonitor()
    self.ams.load(self.assurance_monitor_paths)

    #set up publisher for assurance monitor
    if (self.ams and self.ams.assurance_monitors):
        self.pub_assured_network_output = rospy.Publisher(self.am_topic_str,␣
↪AssuranceMonitorConfidenceStamped, queue_size=1)

def step_am(self, states, actions):
    #do this for assurance monitors
    #lec_input_msg, lec_output_msg = self.publish_lec_input_output(states, actions)

    #invoke assurance monitor with the messages
    #if (self.ams and self.ams.assurance_monitors and self.pub_assurance_monitor_
↪output):
    #    assurance_result = self.ams.evaluate(lec_input_msg,lec_output_msg)
    #    if (assurance_result is not None):
    #        assurance_msg = AssuranceMonitorConfidenceStamped()
    #        assurance_msg.header.stamp = rospy.Time.now()
    #        for i in range(0, len(assurance_result)):
    #            confidence_msg = AssuranceMonitorConfidence()
    #            confidence_msg.type = AssuranceMonitorConfidence.TYPE_SVDD
    #            confidence_level_bounds = assurance_result[i][:3]
```

(continues on next page)

(continued from previous page)

```
    #             confidence_msg.values = confidence_level_bounds
    #             assurance_msg.confs.append(confidence_msg)
    #         self.pub_assured_network_output.publish(assurance_msg)
    pass
```

8.7 System Architecture Models

While the block library generally reflects the functional view of the system by laying out the components in logical groups (or blocks), the architecture model is focused on the implementation view of the system. Blocks from the component library are instantiated in a system architecture model and their ports can be connected to form a composed system model. Since all component blocks in a model are instances of the original block defined in the component library, any changes made to the original block are automatically propagated to all instances. This approach promotes reusability and maintainability of components across multiple system models.

Note: To create an instance of a block library object for the system model, find the desired block in the object browser (found on the top right) and drag it to the system model canvas. The block library is located under `ALC -> 1. Modeling`. To keep it linked with the original block library model, choose `Create instance here`. Fig. 8.30 shows the object browser and the pop-up menu that appears when the object selected is dragged into the modeling definition space (middle window).

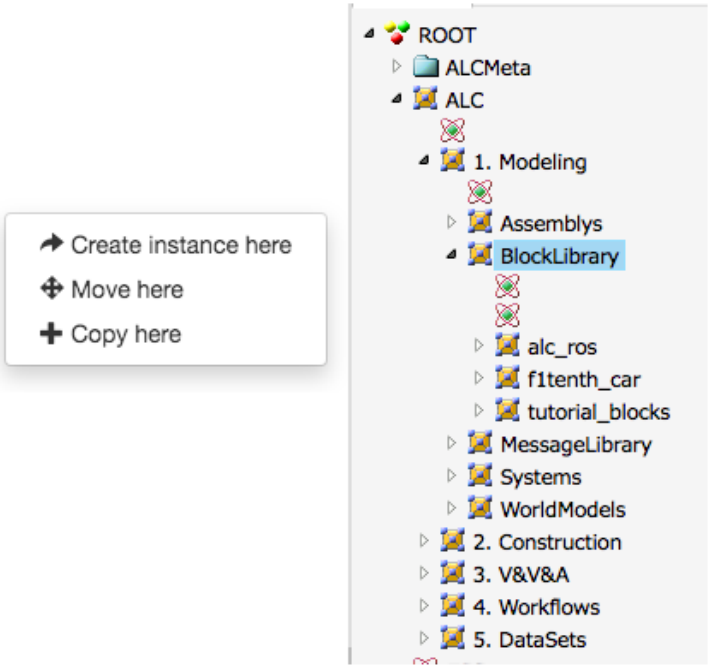


Fig. 8.30: Adding Existing Model Elements

At times, multiple system configurations could be created to meet the same system requirements. This could result in the use of different numbers of components connected in unique ways for the different system configurations or a similar architecture with different drop-in components. If the system design options are diverse in how they are connected and the components used, it makes sense to create multiple system models that could be utilize by different assembly models. In doing this, you lose the ability to see where the design options are located in the system model.

Another way would be to create one system model that reflects all the options available and utilize subsystem hierarchy in the model layout to highlight the available options, this is the preferred methodology and will be discussed moving forward.

Subsystems in the block library can consist of multiple concrete implementation alternatives that satisfies the requirements of the subsystem. Each design option can be wired concurrently to show which blocks are possible in the subsystem and how they are used, as seen in Fig. 8.31 for the drive control of the car tutorial. The **hasOption** attribute should be enabled for blocks containing these implementation. The **IsImplementation** attribute provides a way to indicate that a block is an implementation option available to the subsystem and it can be selected for activation during the system experimentation. The **isActive** attribute is the mechanism used to indicate if the block will be utilized for the experiments. The block is grey when isActive is set to false. Default implementation alternatives can have the isActive flag enabled in the block library or system model, but typically the decision of which option to utilize happens when defining the assembly model for the experiments. In the tutorial system level block for `Drive_Control`, there are four available ROS nodes: `LEC_Control`, `Disparity_Extender`, `Waypoint Follower` and `Fixed_Velocity`. Since the intended use in an assembly model determines which of these nodes are needed, they are left with isActive set to false at this level of the model.

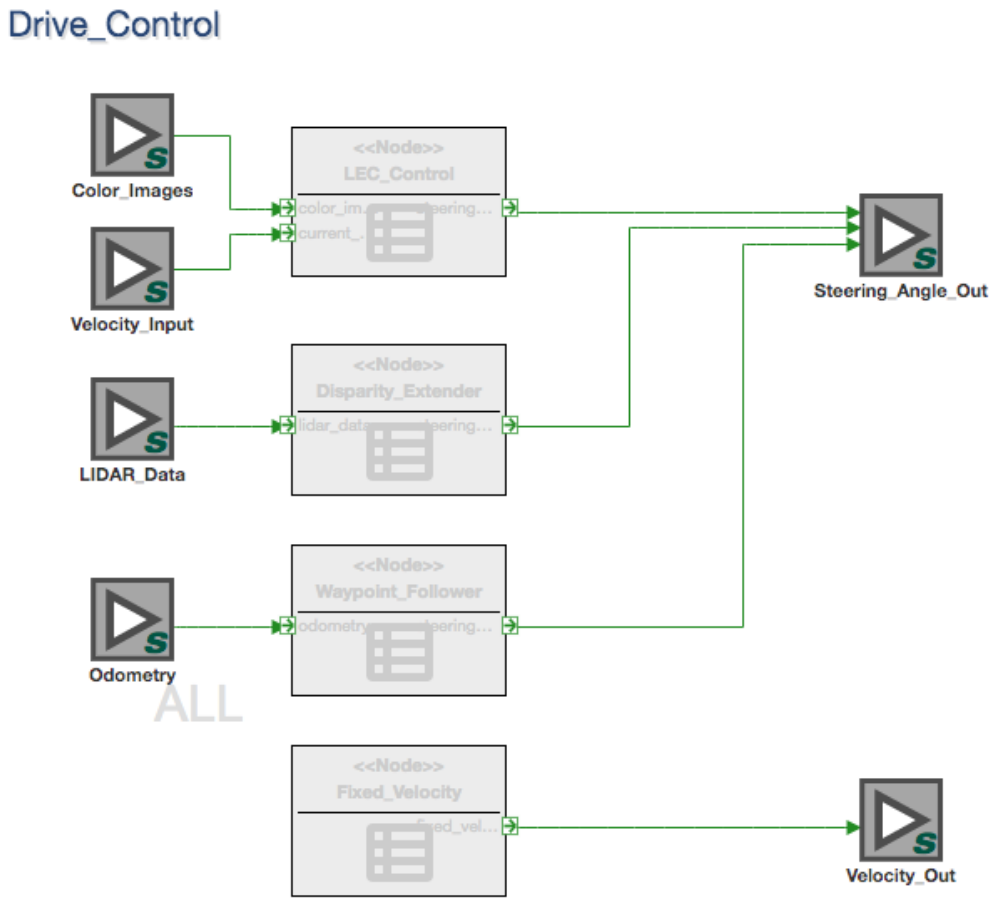


Fig. 8.31: Implementation Options Example

A system level params block at the top of the system definition provides a localized way to setup the system default configuration parameters, shown in Fig. 8.32. These values can be adjusted when running experiments. When defining the ROS nodes in the block library, some parameter values may have been left blank to indicate parameters that the system model needs to define in order to launch the node. The system developer may also overwrite ROS node parameters here by defining a system parameter that can then be mapped to the appropriate ROS node parameter. The **Parameter Mapping** view (available at the top level of the system model and shown in Fig. 8.33) allows the developer

to see all the ROS node parameters (ROS Argument) in the system and their definition provided by the block library elements (listed as Default). System defined parameters can be mapped to the appropriate ROS argument by selecting the parameter in the Parameter Map column. If an entry is left as `--NONE--`, the lower level parameter value will be used. The developer should verify that there is be a value provided for each parameter in the listing.

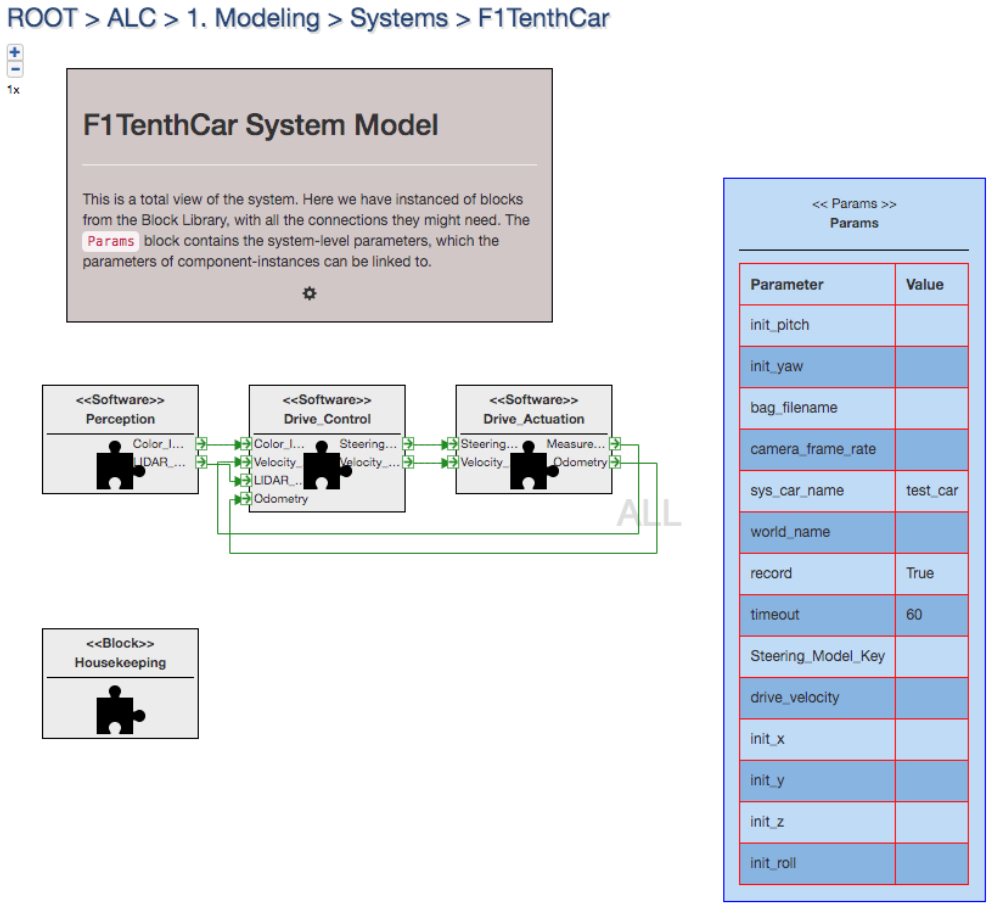


Fig. 8.32: System Model Parameters

Like at the block library level, the **ROSCheck** plugin (available as a play option in top left of tool) is available to make sure the port and message connections are complete. This is a good level to complete the system `Topic` names using the **ROSInfo** view since they are typically specific to the system model created. Running the ROSCheck plugin after filling the topic names is a good way to find unexpected typing errors, as shown in Fig. 8.34. It is also a good way to verify matching messaging pairs across nodes. But, keep in mind that any changes in the system model will not translate down to the block library since it is just an instance of the block library elements, whereas any changes made at the block library level will be reflected in the system library. So, changes to `Type` and `Message Type` typically should be made back in the block library so that they are available for any other system model that utilizes the block.

Note: If ROSCheck is run while in the ROSInfo view, remember to switch to a different view and then back to the ROSInfo view in order to refresh the information on the screen.

ParamMap

in : Drive_Actuation/Gazebo_Connector/ROSInfo

ROS Argument	Default	Parameter Map
camera_clip_far	300	--NONE--
camera_clip_near	0.02	--NONE--
camera_image_height	480	--NONE--
camera_image_width	640	--NONE--
camera_noise_mean	0.0	--NONE--
camera_noise_stddev	0.007	--NONE--
camera_noise_type	gaussian	--NONE--
camera_update_rate	30	camera_frame_rate
car_name		sys_car_name
enable_keyboard	false	--NONE--
gui	false	--NONE--
init_pitch		init_pitch
init_roll		init_roll
init_x		init_x
init_y		init_y
init_yaw		init_yaw
init_z		init_z

Fig. 8.33: Hierarchical Parameter Mapping

ROSInfo

ROS Interfaces in : Drive_Control/Disparity_Extender

Port Name	Type	Message Type	Topic
lidar_data	Subscriber	M :sensor_msgs/LaserScan	lidar_data
steering_angle_out	Publisher	M :alc_msgs/Steering_Angle	steering_angle_out

ROS Interfaces in : Drive_Control/Fixed_Velocity

Port Name	Type	Message Type	Topic
fixed_velocity	Publisher	M :alc_msgs/Velocity	velocity_out

ROS Interfaces in : Drive_Control/LEC_Control

Port Name	Type	Message Type	Topic
color_image	Subscriber	M :sensor_msgs/Image	color_image
current_velocity	Subscriber	M :alc_msgs/Velocity	measured_velocity
steering_angle	Publisher	M :alc_msgs/Steering_Angle	steering_angle_out

ROS Interfaces in : Perception/Color_Camera

Port Name	Type	Message Type	Topic
image_pub	Publisher	M :sensor_msgs/Image	color_image

Fig. 8.34: System Level ROSInfo View

8.8 Assembly Models

Assemblies specify one implementation of a system architecture and is an instance of the specified system model, as shown in Fig. 8.35. Multiple assembly configurations can be defined to provide different options when training or testing the system. A deployment model represents how the system should be deployed to an execution environment. For ROS implementations, top level launch files can be created for the different system implementations to indicate nodes utilized in the execution. For example, there could be a launch file to start a simulation run and a launch file to start a training (LEC) run.

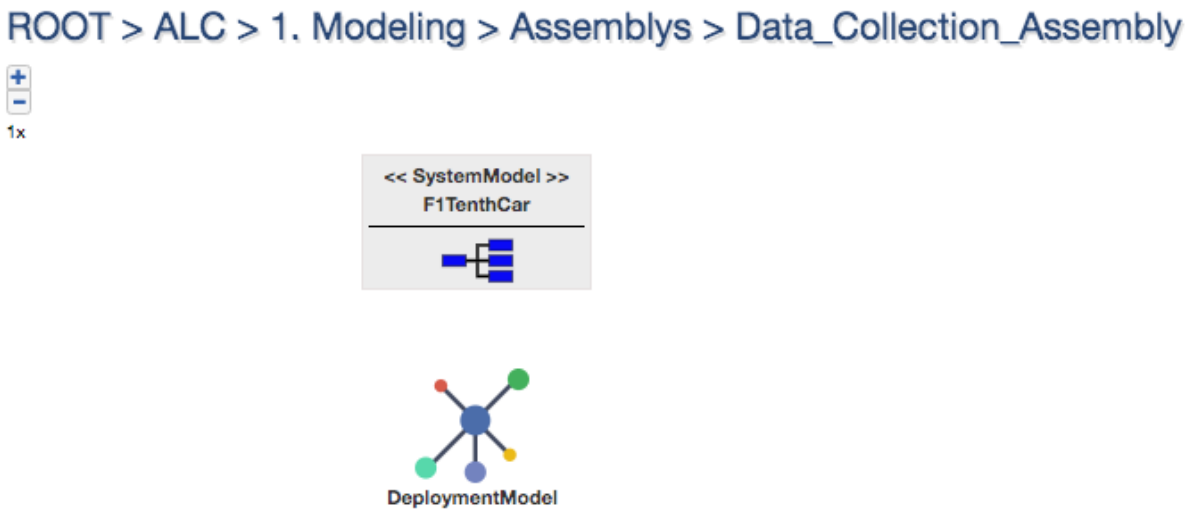


Fig. 8.35: Assembly Model Example

When assembly models have multiple implementation options available for blocks within the model (i.e. for `Drive_Control`), the options can be selected by switching to the Implementation visualizer at the top level of any assembly model and shown in Fig. 8.36. This view contains a list of blocks with alternative implementations and allows the user to select which option to utilize for the specific assembly. Selecting the implementation option here, sets the **isActive** flag in the assembly model instance and the block will turn black, as shown with `Disparity_Extender` and `Fixed_Velocity` in Fig. 8.37.

Parameter values within the assembly instance of the system block may be tailored for the specific assembly setup, but parameters may not be added or removed here. The developer should make any parameter additions and deletions in the block library or system level models.

The **DeploymentModel** holds the top level executable ROS launch files as shown in Fig. 8.38. Each launch file at this level will run in specified dockers. The user will add the desired LaunchFile element to the model and name it. The ROS deployment model launch files can explicitly be linked to any ROS nodes in the assembly's system model. Inside a launch file block, the Selector visualizer (in top left panel) provides a list of all possible system ROS nodes available for implementation by the launch file. Fig. 8.39 shows the available node options for the car tutorial example. If multiple launch files are available, nodes already selected by a launch file will not be available to the other launch file(s). There is a `default` attribute to indicates where the majority of nodes will be launched. The user should set `default` for one launch file and then only select the needed nodes that should be placed in the second launch file. This will place all the unselected nodes into the default node's launch file.

The `container_info` attribute located in the `Property Editor` (on the bottom right and shown in Fig. 8.40) will provide information about about the docker container associated with this launch file. The `image` value is the name of the docker image which will deploy the selected launch file. Other settings are setup based on the code that runs on the server and is not typically modified by the user.

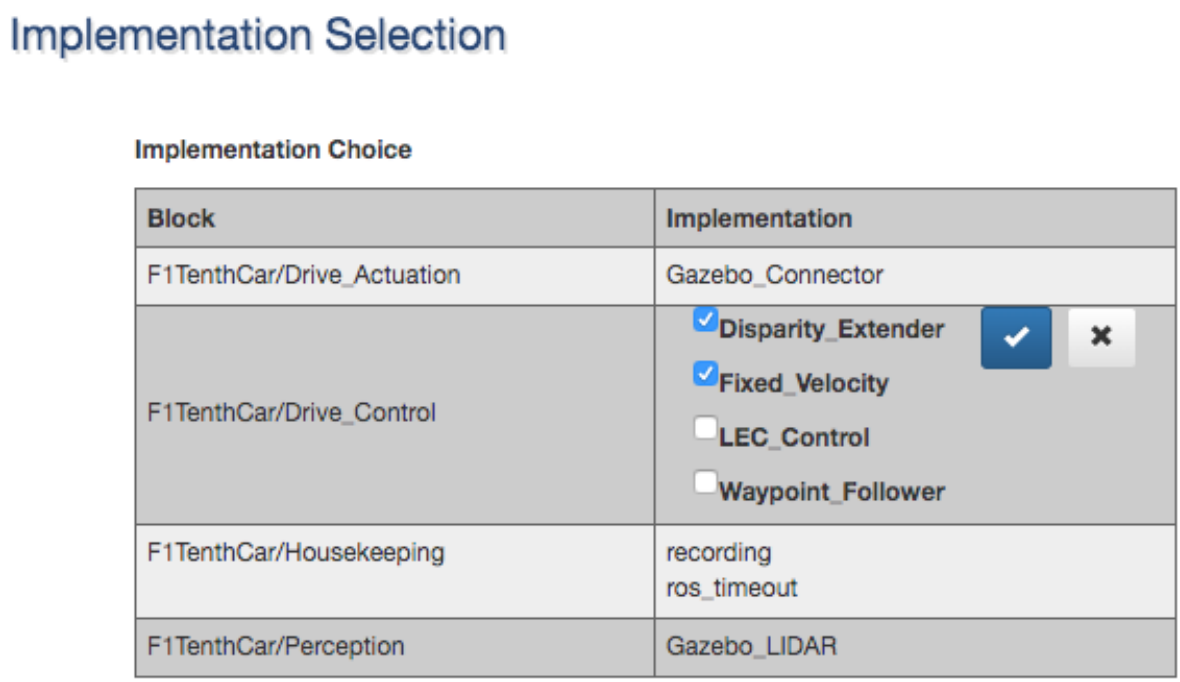


Fig. 8.36: Implementation Visualizer Example

```
{ "name": "f1_10_lec",
  "image": "alc_tf_ros:latest",
  "command": "$F1_UTILS_DIR/f1_10_lec.sh",
   "input_file": "parameters.yml",
   "options":
   {
     "hostname": "f1_10_lec",
      "runtime": "nvidia",
      "volumes":  {"$ALC_HOME/f1_10": {"bind": "/f1_10", "mode": "rw"}, "$ALC_WORKING_
↪DIR/ros": {"bind":"/alc_workspace/ros", "mode": "rw"}},
      "working_dir":"/f1_10"
   }
}
```

When the launch file definition is complete, the user will use the **ROSLaunch Generator** to create launch files based the assembly model. The **ROSLaunchGenerator** plugin is available at the top level of the selected assembly model block and is shown in Fig. 8.41. The plugin will look for ROS Node and Driver code created by the ROS Generator plugin in the block library first, alternately it will use uploaded launch file location information. When the generation completes, a green banner will appear in the top right of the screen (shown in Fig. 8.42). Clicking on the banner will bring up the results of the plugin which will include a `generated_launch_files.zip`, see Fig. 8.43. Clicking on this file will download it to the user's system. These results can also be seen by selecting the WebGME `Show results...` option under the play button (shown in Fig. 8.44).

8.9 World Models

The ALC toolchain is intended for use with both simulated and real-world experiment environments. World models include definitions of environment models to be used during the construction activities of data collection, training and

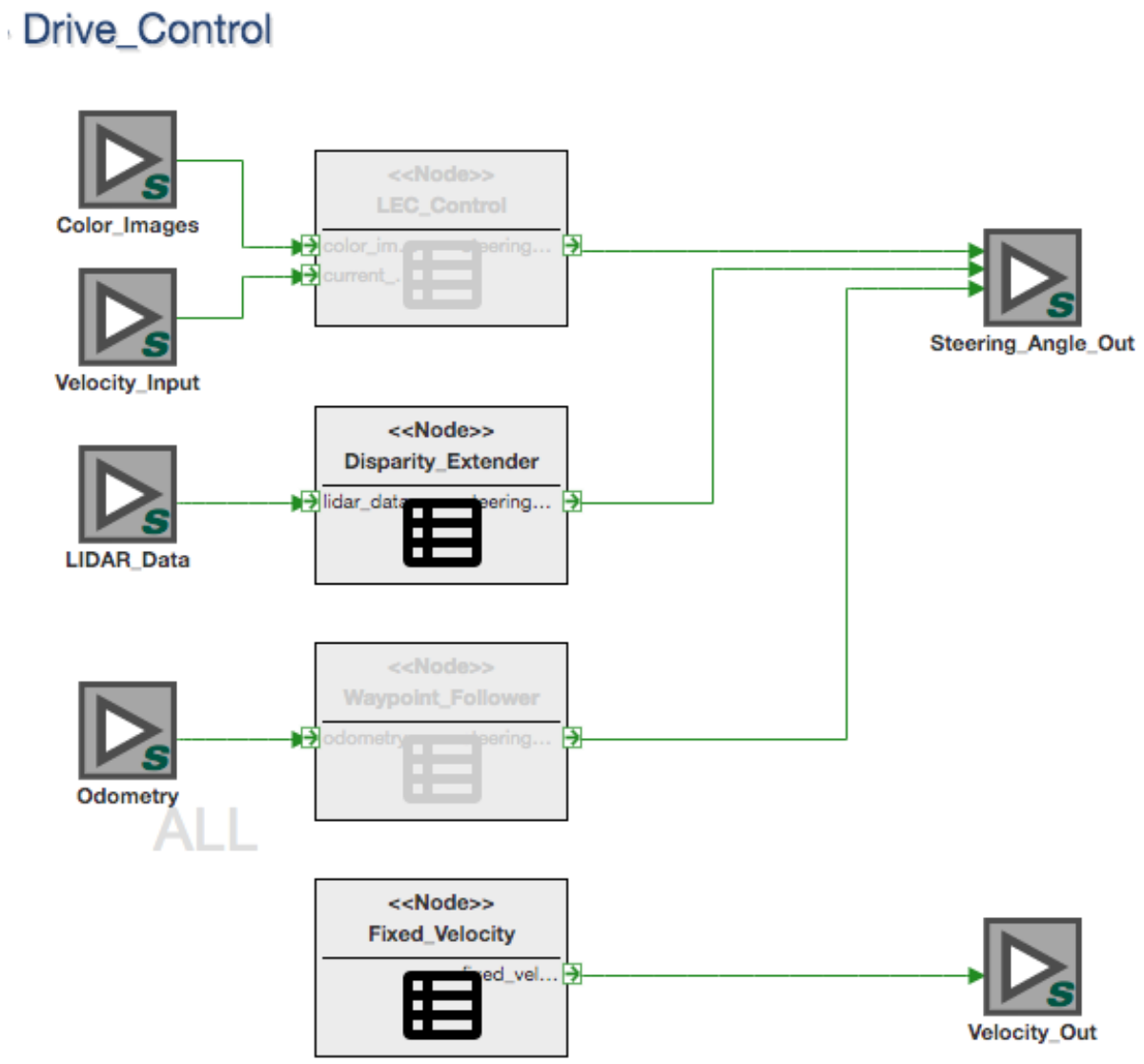


Fig. 8.37: Assembly Implementation After Selection

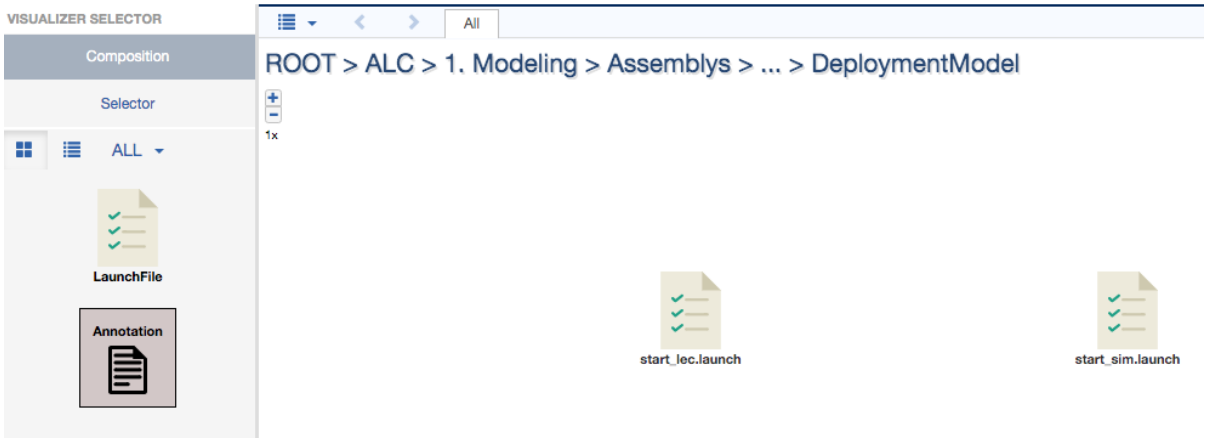


Fig. 8.38: Assembly Deployment Launch Files

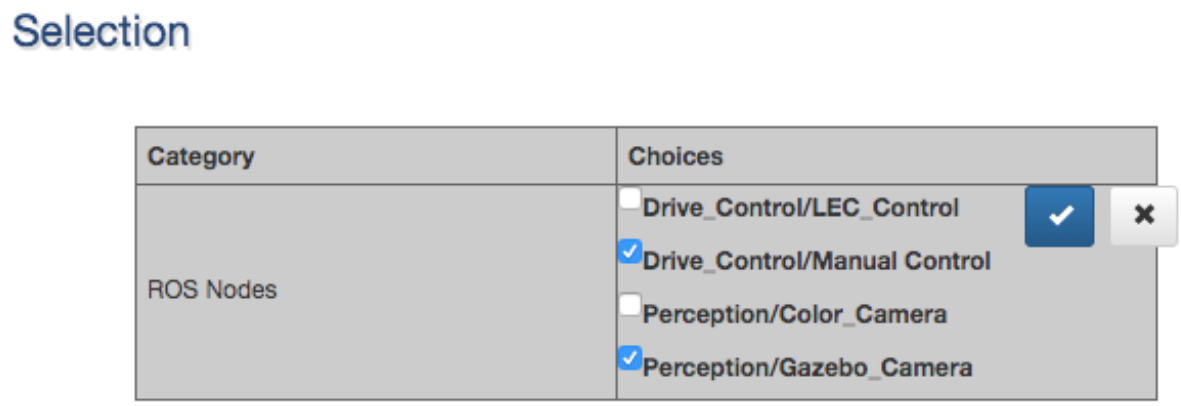


Fig. 8.39: Launch File ROS Node Selection

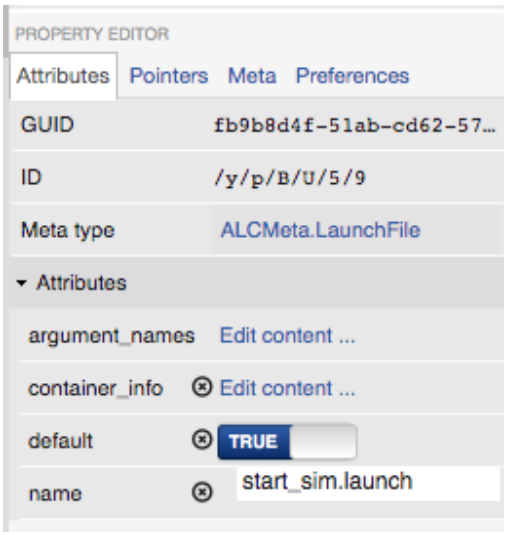


Fig. 8.40: Deployment Launch File Attributes

Fig. 8.41: ROSLaunchGenerator Plugin

Fig. 8.42: ROSLaunchGenerator Finished Banner

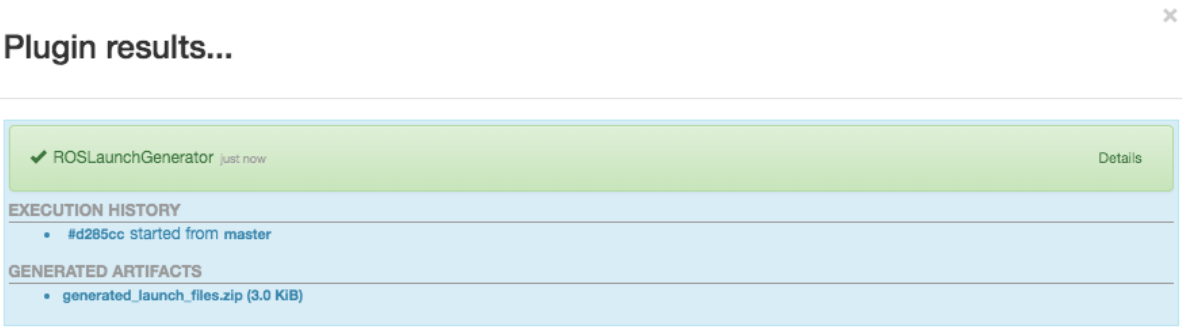


Fig. 8.43: ROSLaunchGenerator Results

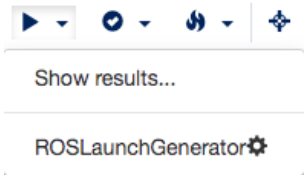


Fig. 8.44: WebGME Show Results

testing. The environment model indicates the configurable set of parameters using Params tables, as done in the block library. An example world model used in the car tutorial is shown in Fig. 8.45.

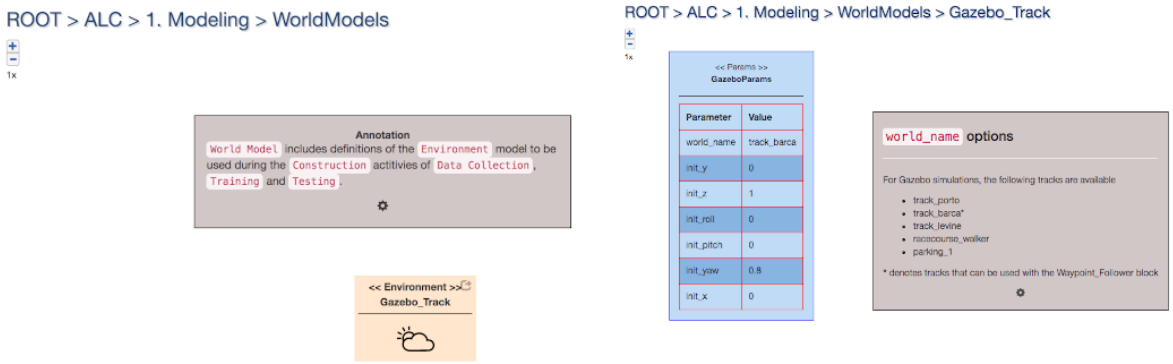


Fig. 8.45: World Model Example

Note: Currently the toolchain is manually configured in the backend processing on a per project basis to work with different simulation environments, also called `gyms`.

CHAPTER

NINE

LEC CONSTRUCTION

Once the system architecture and available assembly models have been defined, the user can move to the next step in the development workflow: LEC construction. LEC construction involves multiple sub-steps (shown in Fig. 9.1): data collection, LEC learning (or training), and LEC evaluation through integrated system testing. Data collection involves designing experiment models to create training data for the LEC models or uploading an existing data set. In the LEC training step, the user configures the LEC model and specifies the associated training data sets. The training data sets could be an uploaded data set, data created by a data collection experiment or the model could be retrained with previous trained data results. LEC evaluation could be running through integrated system testing using the LEC model or evaluating the model with a test data set. Post processing can be designed in both the LEC training and evaluation steps to provide insight into how well the model is performing.

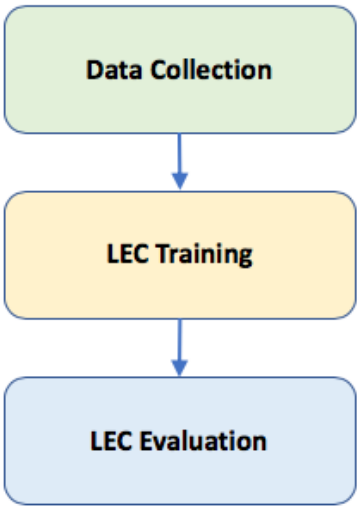


Fig. 9.1: LEC Construction Elements

9.1 Data Collection

Data collection section of the model is a place to gather data sets for use in the training process. This can be done by either importing an existing data set into the model or running experiments using the system model to create a data set. It is good practice to add an **Annotation** block to explain the expected use of each experiment to help other users understand quickly what this experiment will capture and how it is configured to meet the desired goals.

9.1.1 Import Existing Data Set

Note: This section describes how external data sets can be uploaded to the ALC Toolchain and used for LEC training. Completion of this section is not required to follow the remainder of the tutorial. If the reader does not wish to use externally-generated data, recommend proceeding to the *Designing an Experiment* section.

If a LEC model will be initially trained using an external dataset, this information needs to be uploaded and imported into the project model. The dataset must first be copied (using tools such as `scp`) to the server running the toolchain into a directory location accessible to the experiment executors (typically in `$ALC_WORKING_DIR`). Then create an Experiment block for this data and add a results block in that experiment, shown in Fig. 9.2. Inside of the experiment block, the **Import Data** plugin (execution option available in the top left corner by the play button) will provide a method to indicate the dataset folder location (**Directory to Upload**) and identification of the dataset (**Record Name**). The plugin will create a record of the uploaded dataset location and make it available for use as an input in the LEC training. For this car tutorial, a data set was uploaded to the execution server and placed in the `$ALC_WORKING_DIR/jupyter/f110_data folder` (which would be the folder to upload) and a Record Name of `dataset1` was given as shown in Fig. 9.3.

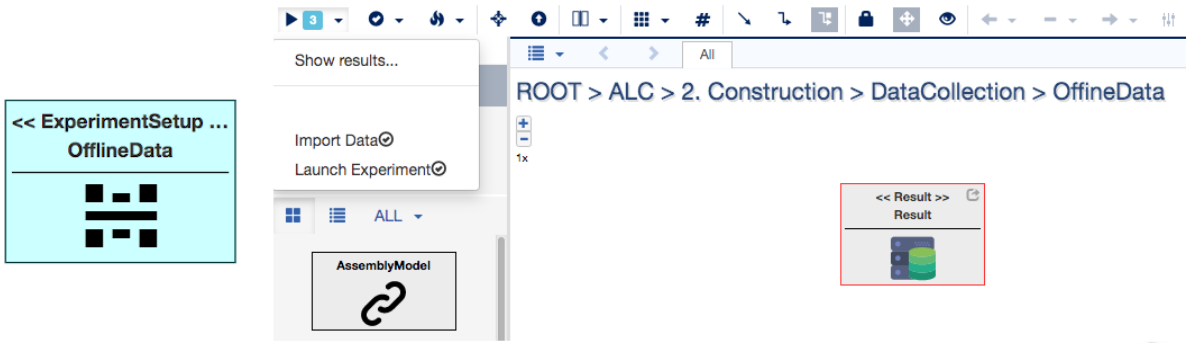


Fig. 9.2: Offline Data Experiment

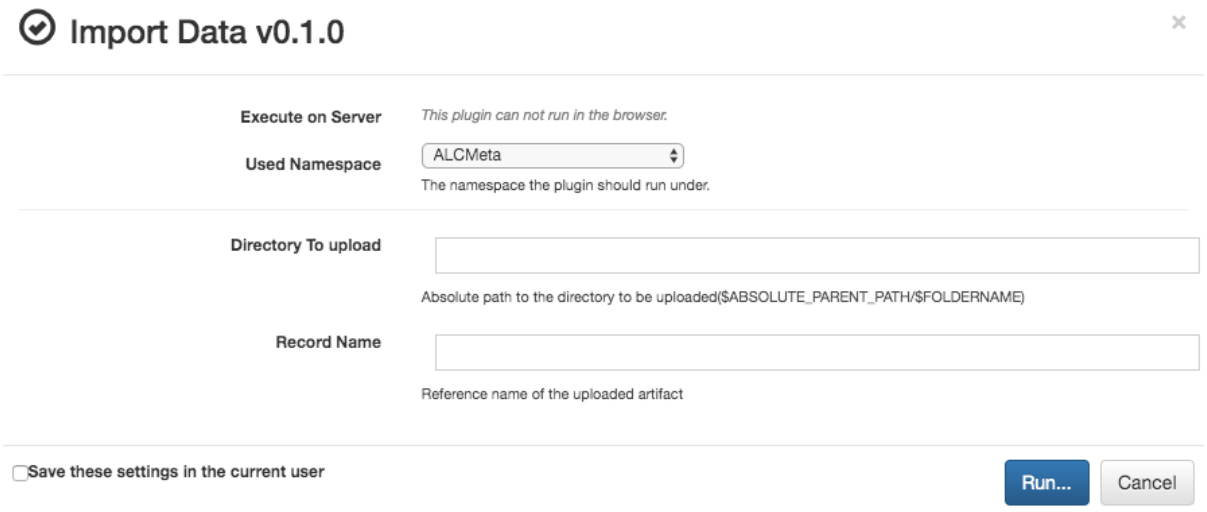


Fig. 9.3: File Upload Plugin

There is also an option to copy the dataset to a FTP fileserver location on the toolchain server (`$ALC_FILESERVER_ROOT/files/alc_workspace`) using a **Path Prefix on fileserver** and setting **Create Metadata only** to `False`. This process will create a new folder with the path prefix name under the FTP fileserver location to provide a specific location for dataset information. This method has been known to be a very slow since dataset tends to be large amounts of data and is not recommended.

9.1.2 Designing an Experiment

The experiment model captures all information necessary for configuration and execution of a system, Fig. 9.4 shows an example from the car tutorial. Construction of an experiment model begins by selecting the desired system configuration from one of the available assembly and world models (see implementation note below). Then the assembly model of the system under test is refined to select the desired implementations for each component that includes multiple implementation alternatives. Users can setup as many experiment models as desired for complete system testing.

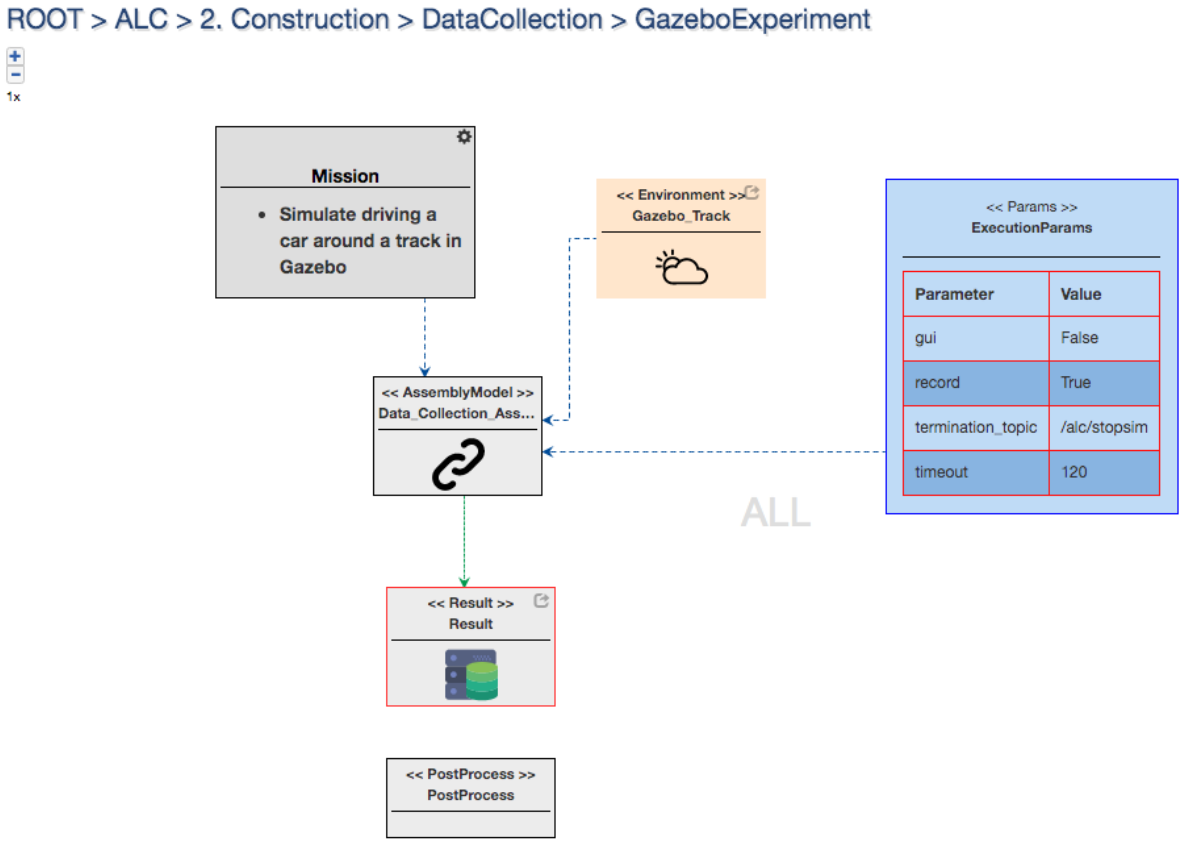


Fig. 9.4: Data Collection Experiment Example

Note: To create an instance of a modeling library object for the experiment model, find the desired assembly and/or world model in the object browser (found on the top right) and drag it to the experiment model canvas. The modeling library elements are located under `ALC -> 1. Modeling`. To keep it linked with the original model element, choose `Create instance here`.

Experiment mission objectives are stated in a **Mission** block, along with specific configuration parameters related to the mission. Add text to this block in the same way as the annotation block (by clicking on the gear icon located in the top right of the block to bring up an editing window). Any associated mission parameters should be placed in a **Params** block inside of the mission block. Fig. 9.5 shows an example of mission parameters added to the mission block from the car tutorial. It is also helpful to add annotation blocks to explain the execution of this experiment, the anticipated results and expectations on how the resulting data set will be utilized.

Execution of the assembly model requires additional parameters (**Params** blocks) divided into three sets: mission (already discussed), environment, and execution. Environment parameters define any environmental variables (such

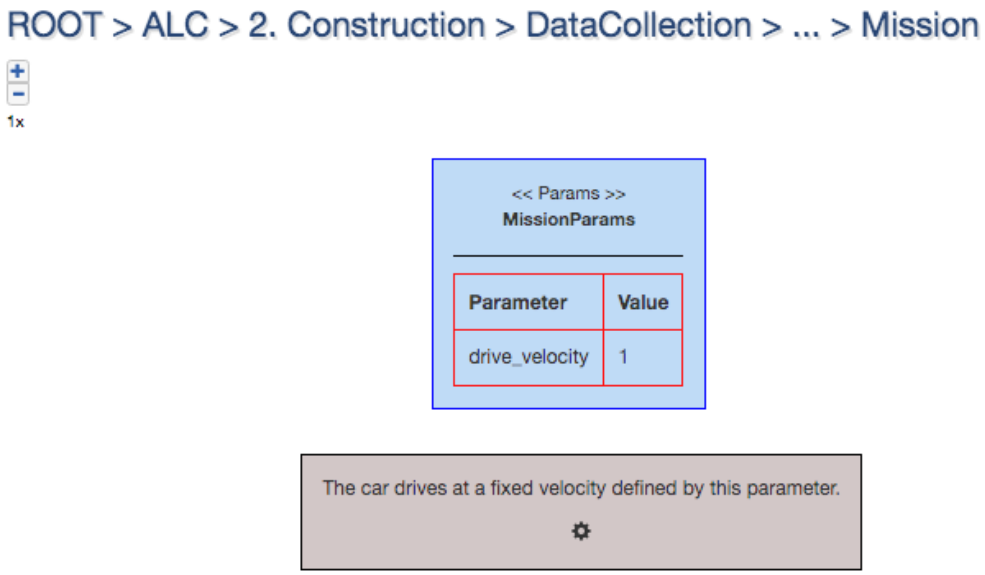


Fig. 9.5: Mission Block Parameter Example

as track name or initial car position) necessary for launching the simulation. This params block is located inside the world model instance and can be modified for this specific experiment setup. Execution parameters define additional housekeeping parameters for simulation management (e.g. simulation timeout, startup delays, etc.). These parameters are located at the top level of the experiment model. Below is a list of some typical execution parameters and their meaning. The list used should be tailored to the particular experiment configuration needs, some parameter may not be required for some system setups (such as unpause_timeout and num_episodes).

Parameter	Description	Units
record	Record simulation results, provides a recording.bag file in the result folder	true/false
gui	GUI interface used in the Gazebo simulation runs, typically set to false since run remotely and GUI will not be available to user	true/false
unpause_timeout	Time to wait before starting the simulation to allow all resources to be loaded	seconds
timeout	How long a simulation should run before being terminated by the execution runner	seconds
termination_topic	Topic that execution runner can listen on and terminate the dockers once a message is received (default = /alc/stopsim)	string
num_episodes	Number of simulated (randomly-generated) scenarios to run	integer
generateROSLaunch	Sets default launch experiment behavior to generate the ROS launch files from the model	true/false

Note: The timeout parameter allows the completion of the ROS run so that any desired recording can be correctly terminated and provide the resultant recording.bag file. Since this parameter is used by some ros-launch files to

terminate themselves, a buffer time is added by the execution runner before the dockers are stopped

The user may also adjust the values of the system parameters within the assembly model instance for the particular experiment by drilling down into the assembly model and the system model within it. Additionally, the system parameters could be duplicated in the execution parameter table to make it obvious to a user that this experiment changed particular parameter values. Any ROS parameter configured (with matching names) in the execution parameter tables will be passed as command line arguments when the ROS code is launched for the experiment and will overwrite the model defined value.

A post processing block (**PostProcess**) is often used for data analysis tasks such as calculating performance metrics, plotting relevant data, etc. The experiment results are presented in a Jupyter notebook which will copy the post processing block code into the notebook. The user can write Python 2.7 code to be executed on the generated data after an experiment completes.

To get the experiment results, a **Result** block is added to the experiment model. The corresponding link to the results of any run experiment can be found under this block and will be discussed later.

Once an experiment model is created, the user can launch an experiment by selecting the **Launch Experiment** plugin under the top left play button, see Fig. 9.6 for the form used to launch the experiment. Choose an `Execution name` that is relevant to the dataset results that will be provided. The resultant dataset will be available for use in the training runs later. To utilize the ROS launch files created from the system model, select the `Generate ROS Launch files` option. The experiment execution typically occurs as a non-blocking background process that reports its status and the results, but the user could request to interactively control and watch the execution using a Jupyter notebook. To get this notebook, the user would select the `Setup Jupyter Notebook` option in the launch experiments dialog. The notebook will appear on the experiment model and can be opened to start and view the execution steps.

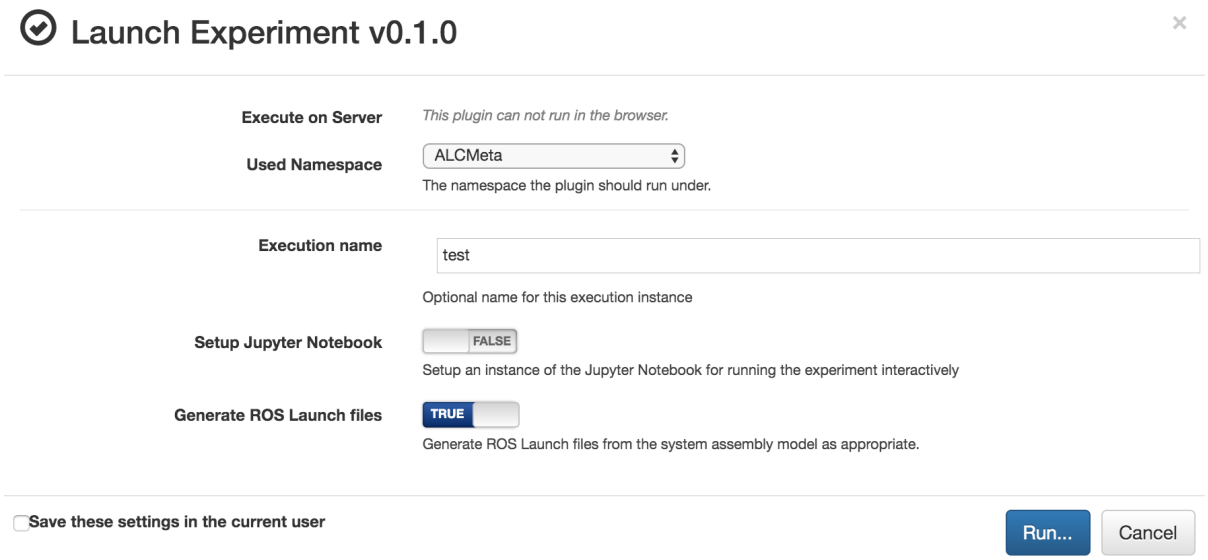


Fig. 9.6: Launch Experiments Plugin

Note: The `generateROSLaunch` parameter is needed when using this experiment in a workflow job. In manually run experiments, this parameter does the same thing as the `Generate ROS Launch files` option. For workflow jobs (discussed in *Utilizing Workflows*), this option is not available to the user, so setting this as a parameter will allow the experiment to be launched with these generated files.

Once the experiment completes, all generated data and execution traces are automatically stored and made available in the **Result** block for review. The **ArtifactIndex** view shows the available results from any executed experiments,

an example is shown in Fig. 9.7. As the job is being launched, an entry is added to the results list and will be colored yellow to indicate that it made it into the job queue. Once the job execution begins, the entry will turn blue and a *Log* link is available to follow the execution progress. Once the job completes, the entry will either turn green (succeed) or red (failure). Successful runs will project a link to the result Jupyter notebook.

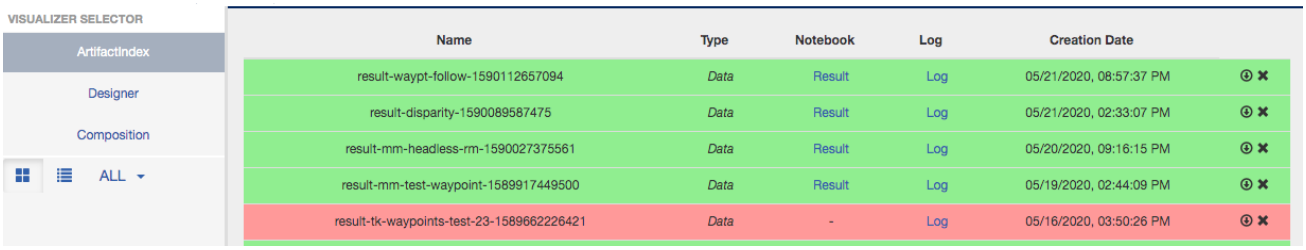


Fig. 9.7: Results Block - ArtifactIndex Visualizer

Note: An experiment result can be renamed by double clicking on the name and changing it to a new descriptive name.

Once the experiment completes and a `Result Notebook` is available, the user can open the Jupyter notebook to see the results and interact with the data. The initial view of the notebook has three sections to run as shown in Fig. 9.8. When the third section is executed, the script provided in the PostProcess block will be copied to this location in the notebook as shown in Fig. 9.9. To execute the post processing, select the section again and run the code. A typical output of the post processing is graphic and metric information illustrating the performance of the system during the experiment. An example result notebook for the car tutorial is shown in Fig. 9.10.

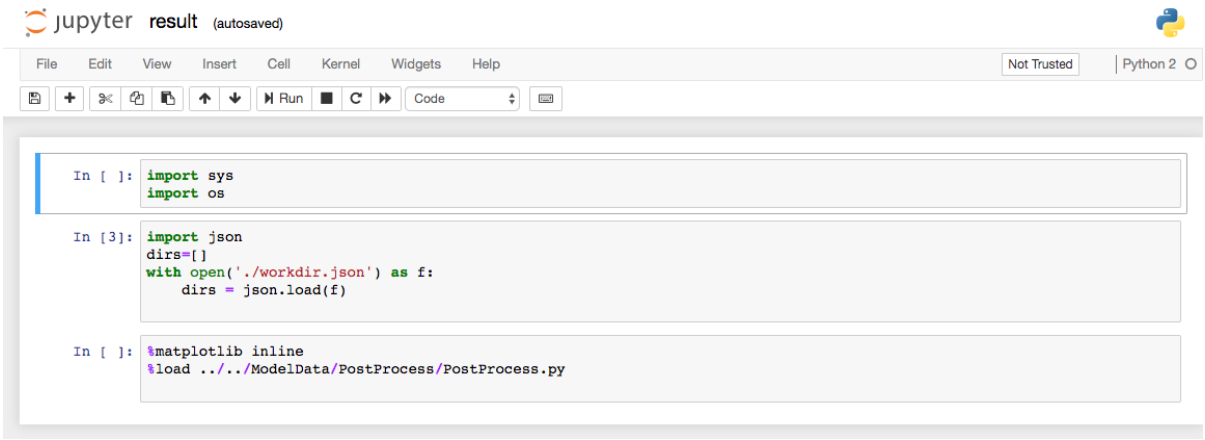


Fig. 9.8: Results Jupyter Notebook - Initial View

Note: Be mindful of the utilization of the GPUs when running post processing activity in the Jupyter notebook. If the processing is utilizing a GPU, be sure to shutdown the kernel in the notebook before closing the results tab in your browser (under `Kernel` menu there is a `Shutdown` option).

During the development phase, the user may want to look at the ROS logs to locate any issues with their component code. To access the directory with the results, use `File` and then `Open` in the notebook. Fig. 9.11 shows the folder system where the results reside on the server running the experiment. If the `record` parameter was set to true, all available simulation data will be stored by the ROS recorder utility in the ROS bag file format (recording.bag) and will be available under the `results` folder. To retrieve this recording file for analysis on a different platform, the

jupyter result Last Checkpoint: 05/21/2020 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Not Trusted Python 2

Run Code

In [1]:
```
import sys
import os
```

In [2]:
```
import json
dirs=[]
with open('./workdir.json') as f:
    dirs = json.load(f)
```

In [5]:
```
# %load ../../ModelData/PostProcess/PostProcess.py
# Load up the ROS Bag recording file
import os
import rospy
import rosbag
import matplotlib.pyplot as plt
import matplotlib
import time
from cv_bridge import CvBridge, CvBridgeError
import nav_msgs
import geometry_msgs

# Specify the path to the bag, and topic names
BAG_PATH = './results/recording.bag'
# If you don't include the / you are done (smh...)
TOPIC_NAMES =['/camera/zed/rgb/image_rect_color', '/steering_angle_out', '/odometry'];

def loadMessages(bag_filename, topic_list):
    # specify topic and get messages
    # FIXME: Why does this function load only the first topic name in the list of topics?
    #START_TIME = rospy.Time(0)
    bagfile = rosbag.Bag(bag_filename)
    msgs = bagfile.read_messages(topics=topic_list)
    # create a dictionary that will store the messages in sequential order
    msg_dict= {}
    for tp in topic_list:
        msg_dict[tp]=[]

    for topic, msg, t in msgs:
        msg_dict[topic].append((msg,t))
    bagfile.close()

    return msg_dict
```

Fig. 9.9: Results Jupyter Notebook - PostProcess Code Inserted

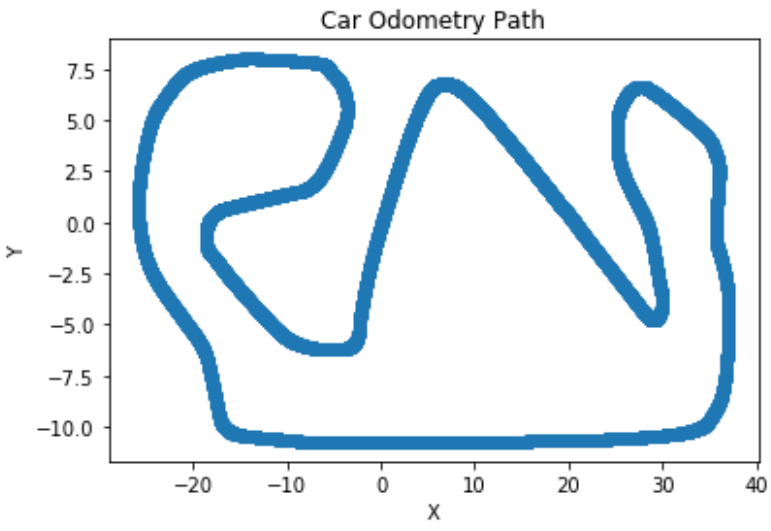


Fig. 9.10: Results Jupyter Notebook - Example Metrics Graphs

user should transfer the file from the execution machine using a `scp` command. The file location on the execution machine will be `$ALC_WORKSPACE/jupyter` and then add the file path seen at the top of the notebook. The `recording.bag` file can be replayed and viewed using RVIZ, a ROS bag visualization utility.

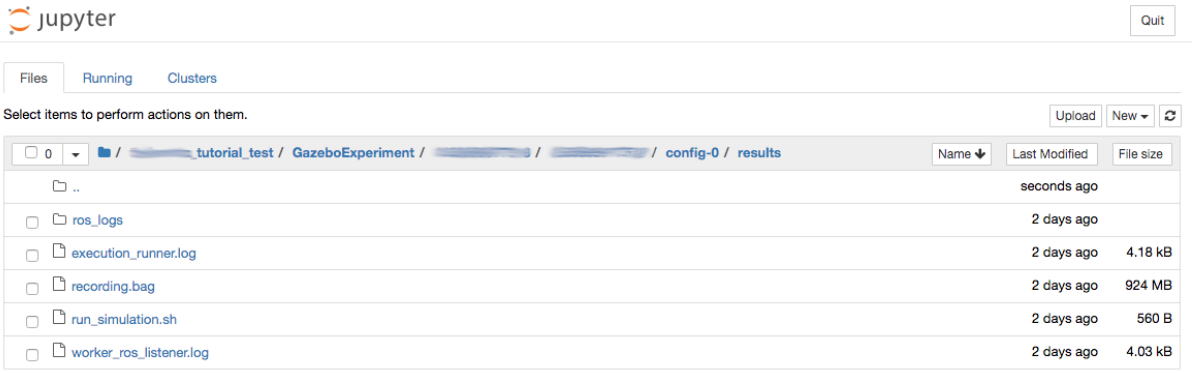


Fig. 9.11: Results Jupyter Notebook - Result Files

Another way to pull up the result information from any experiment run is to go to the DataSet (at the top level of the ALC toolchain model, shown in Fig. 9.12) information. This provides all the run information from any experiment, training or evaluation runs. The `Config Parameters` column holds the parameters used for the run and is initially collapsed into `....` Click on these dots to expand the parameter information, an example is shown in Fig. 9.13. There is also a link to the result Jupyter notebook in the last column.

An experiment model can be enriched with a Campaign block which configures iterations of an experiment. Fig. 9.14 shows the model element (on the left) that is used to create the campaign block (on right) that can be added to the experiment. Campaigns include a parameter sweep block for varying system, mission objectives or environment parameters over a range of possible values. Campaigns are commonly used to tune system parameters for optimal performance or to gather data in a variety of environments for LEC training. Evaluation jobs are often configured as a campaign where the same evaluation procedure is repeated in a variety of scenarios. When a single campaign block specifies sets of values for multiple parameters, the complete parameter set is defined as the Cartesian product of all the individual sets.

When a campaign is executed, the corresponding experiment is executed once for each valid combination of parameters in this set as shown in Fig. 9.15. The results will be provided on sequential lines of the ArtifactIndex with run indication at the end of the run name (`-0`, `-1`, ...).

9.2 LEC Training

In the system modeling, the expected location of the LEC was specified and a deployment key provided. How the LEC operates and the code to execute in the training process are all specified in this section of the model. The ALC Toolchain supports specification of artificial neural networks (ANN). Design of a suitable ANN architecture (or selection from existing architectures) is often an iterative process driven by empirical results. Performance indicators, including model loss and accuracy against a testing data set, are the most common evaluation metrics. CPS devices often operate in resource-constrained environments and must also consider factors such as model size, inference time, and power consumption. To support rapid design iterations, a new LEC architecture can be tested simply by updating the architecture definition within the LEC model block and executing the training model. Two key LEC development workflows available for designers in the ALC toolchain are supervised and reinforcement learning. Fig. 9.16 shows model elements for both of these types of training methods. Assurance monitors and their training will be discussed in the *Assurance Monitoring* section.

Supervised learning begins with the collection of training data using one or more experiment models. In a supervised learning setup, a LEC model is trained against the collected data set. Then the experiment models may deploy the

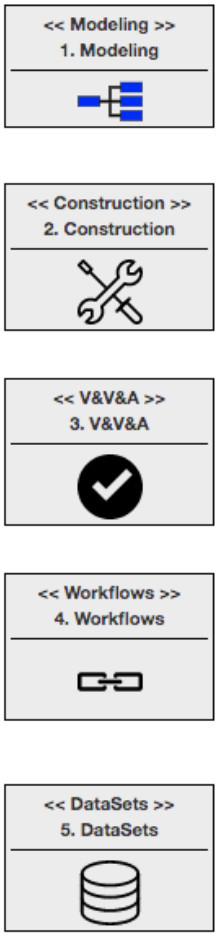


Fig. 9.12: ALC Model Elements - DataSet

Construction/DataCollection/GazeboExperiment

Data-Set	Location	Creation Time	Config Parameters	Trained LEC	Trace	Results
result-mm-headless-rm-1590027375561	timkrentz_tutorial_test/GazeboExperiment/1590027375561/1590027375561/config-0	05/20/2020, 09:16:15 PM	... { "init_roll": 0, "init_z": 1, "init_y": 0, "init_x": 0, "gui": false, "init_pitch": 0, "generated_ros_launch": true, "record": true, "termination_topic": "/alc/stopsim", "world_name": "track_barca", "launchfile": "/alc_workspace/jupyter/timkrentz_tutorial_test/GazeboExperiment/1590027375561/ModelData/launch_files/start_sim.launch", "timeout": 120, "drive_velocity": 1, "project_name": "timkrentz_tutorial_test", "init_yaw": 0.8 }			Result
result-mm-test-1589939770780	timkrentz_tutorial_test/GazeboExperiment/1589939770780/1589939770780/config-0	05/19/2020, 08:56:10 PM	...			Result

Fig. 9.13: DataSet View of Runs and Associated Results

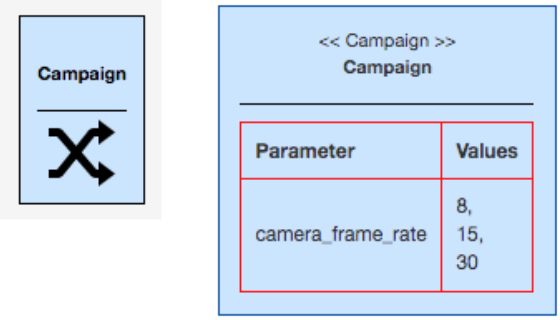


Fig. 9.14: Example Campaign Block

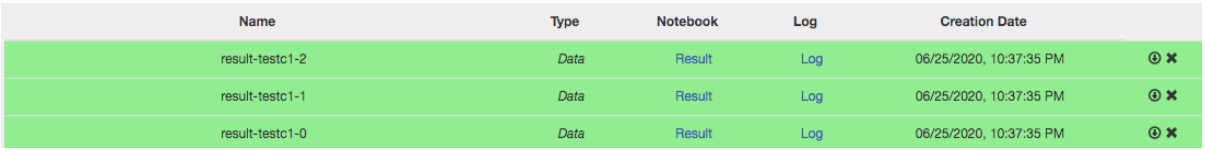


Fig. 9.15: Example Campaign Results

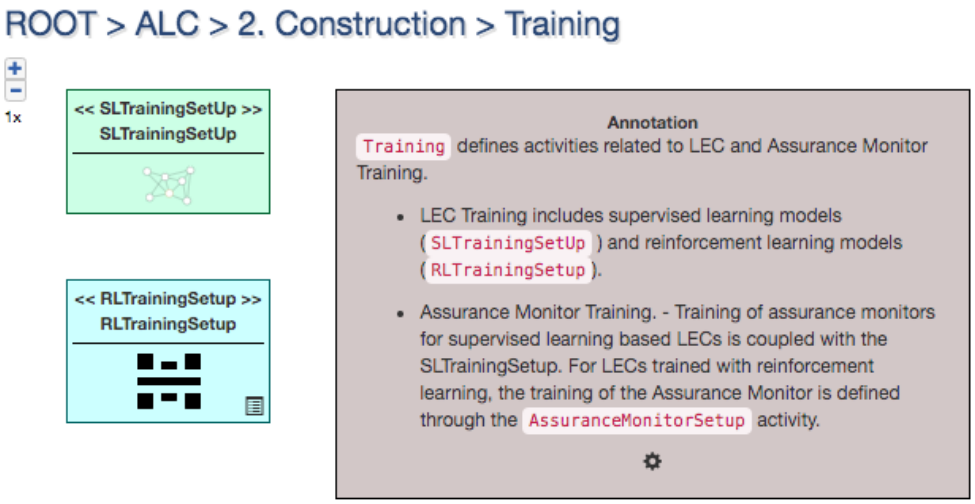


Fig. 9.16: LEC Training Models

trained LEC in the system for integrated system testing and evaluation.

In a reinforcement learning setup, the LEC model is trained while the system or environment is being executed (or simulated). During training, the LEC model is updated based on the environment response (state and reward) to the input action. The training is repeated for many simulated, randomly-generated scenarios known as episodes. During each episode, the LEC explores the possible action space while using the provided reward function to observe which control actions produce desirable results. This process requires a closed loop agent environment simulation.

Training and execution of LEC models is done using the Keras neural network library on top of the TensorFlow machine learning framework. Training the model is done by executing experiments based on the setup provided, similar to the data collection experiments. The user can define post processing code to provide any desired metric or graphical representation of the training results. The resultant trained LEC will become available in the result block and can then be evaluated by executing the LEC in a non-training mode for the LEC evaluation step or the model may also be retrained using new data. A history of the experimentation results and datasets can be found in the model's `DataSets` section. The history includes an indication of which trained (or retrained) LECs were used in any experiment, along with the configuration parameters of the LEC training and experiments performed.

9.2.1 Supervised Learning

In the ALC Toolchain, training of an LEC using a supervised learning approach is defined by a **SLTrainingSetUp** block. Constructing this block begins by defining a LEC model which specifies the desired ANN architecture as well as basic formatting functions used to interface with the toolchain. A typical supervised learning model would consist of a set of labeled training data with known output values, a LEC model to train, the training code and any necessary hyper-parameters. Both classification (discrete output range) and regression (continuous output range) based models are supported in the toolchain. Fig. 9.17 shows an example layout of a supervised learning model.

The training data block (**TrainingData**) specifies which of the available data sets should be used to train the ANN, which could be a user generated datasets (uploaded to the execution server) or results from previously run experiments. For the car tutorial, the uploaded dataset (`dataset1`) was used to train, shown in Fig. 9.18. There are also evaluation data blocks **EvalData** and **ValidationData** blocks available so that data can be designated for each of these types of datasets.

The parameter block (**Params**) captures hyper-parameters relevant to the specific training exercise, how to configure the training algorithm (e.g. batch size, number of epochs, etc.) and any needed execution configurations. Below is a list of some typical parameters and their definitions. The list used should be tailored to the training configuration needs (i.e. which machine learning libraries are used). LOSS, OPTIMIZER and METRICS are typical Keras related parameters options and can be found in the TensorFlow Keras API.

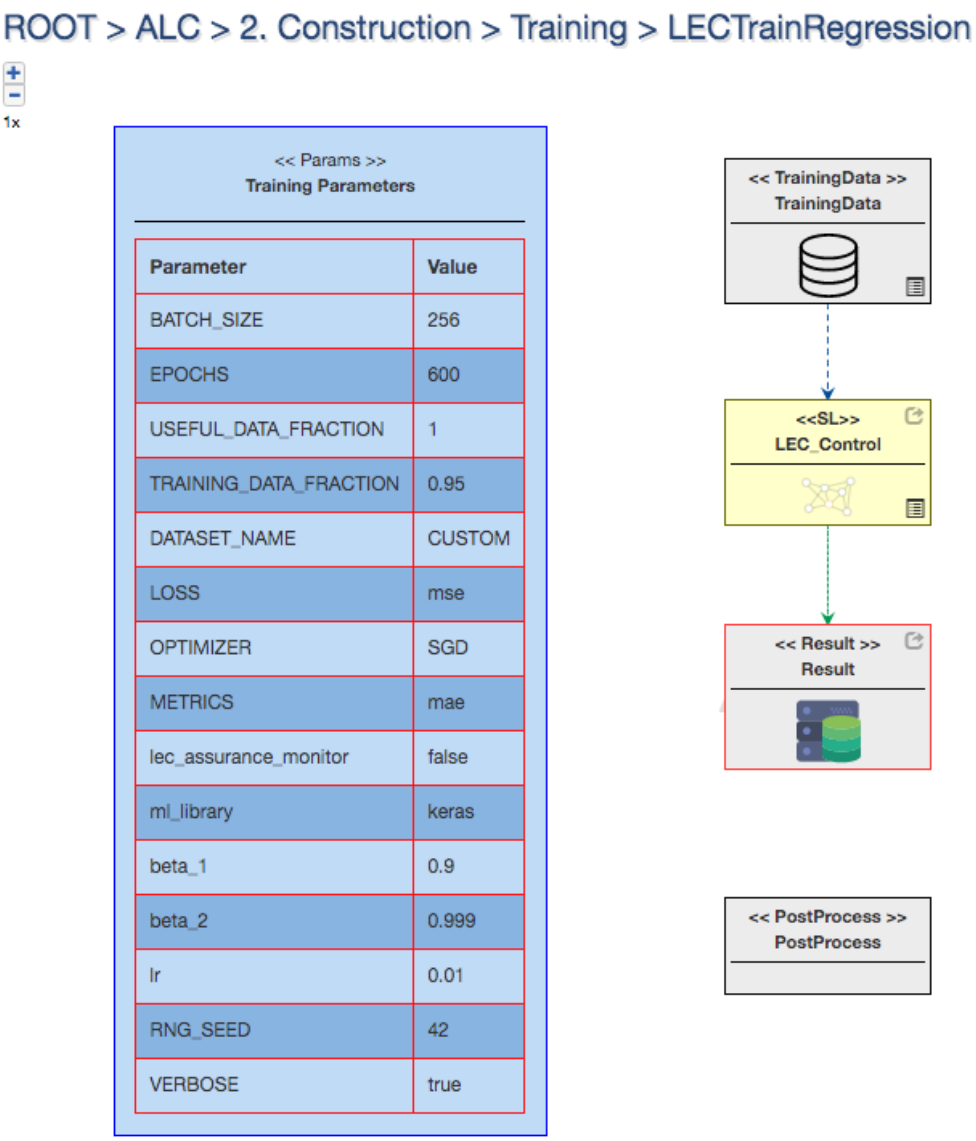


Fig. 9.17: Supervised Learning Model

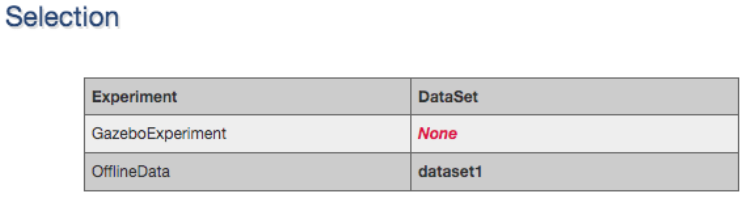


Fig. 9.18: Training Dataset Selection Options

Parameter	Description	Units
BATCH_SIZE	Size of each training batch	integer
EPOCHS	# passes through data for training	integer
USEFUL_DATA_FRACTION	% data used for training & testing	value 0-1
TRAINING_DATA_FRACTION	% data used for training, rest for testing	value 0-1
LOSS	Loss function type (https://keras.io/losses/)	mse(default)
OPTIMIZER	Optimizer type (https://keras.io/optimizers/)	adam(default)/sgd/. . .
METRICS	Model performance (https://keras.io/metrics/)	acc(default)/mse/. . .
DATASET_NAME	Types of dataset to load in model	rosbag/rosbag_hdf5/ csv/png
RNG_SEED	Random number generation seed	integer
ML_LIBRARY	Machine learning (ML) platforms	pytorch-semseg/keras
VERBOSE	Detailed logging from ML libraries	true/false

Note: The datasets are randomly sampled when pulling data for training and testing data per the USE-

FUL_DATA_FRACTION and TRAINING_DATA_FRACTION settings,

Note: If the user would like to define a unique loss, optimizer, metric or dataset input function, the parameter value should be set to `CUSTOM`. For loss, optimizer and metric definitions, add `get_` functions to the LEC model definition. Custom dataset functions are provided in the `Dataset` function. In the tutorial, the optimizer and dataset was customized.

Note: To provide a list of options, as might be done for Metrics, the options should be listed as comma separated strings inside a bracket set, with double quotes around each string — an example: `["mse", "accuracy"]`.

Note: When using VERBOSE logging, the fit call is fixed to provide one line per epoch during training.

Note: Machine learning platform will be extensible as the tool matures.

Within the **LEC model** block, the selector view (found in top left panel) allows the user to indicate the starting point for training a LEC. The first time a model is trained, the `DataSet` will be set to `None` so that new LEC model is created based on the training data, shown in Fig. 9.19. Once trained, a LEC model can be retrained from a different starting point, for example using a previously trained LEC model. This can be done by choosing any of the pre-trained LEC models that are available and then running the training experiment again.

Fig. 9.19: Retraining with Previous Models - Selection

For the **LEC model** definition, a code set is created by the user to specify a neural network using the TensorFlow, PyTorch and Keras libraries. Selecting or drilling down into the LEC block, the user can use the CodeEditor visualizer (on the left menu) to see and edit the code set describing the LEC shown in Fig. 9.20. The different model attributes, on the right in the property editor, bring up a full screen code editor for the same model code files shown in the CodeEditor visualizer. Edits can be made either way and are editing the same files.

The **Definition** code defines the desired Neural Network architecture using Python. The data type of the returned model object must be compatible with the selected ML Library Adapter indicated in the ML_LIBRARY parameter. Boilerplate code is provided for the LEC model functions to provide a framework for the user customization, shown in Fig. 9.21. The `Definition` file provides a `get_model` function which should be filled in by the designer with the model design information.

Below is an example model definition from the car tutorial. This model uses the NVIDIA DAVE-2 Network Architecture.

```
#import the neccessary kpackages
from keras.models import Sequential
from keras.layers.normalization import BatchNormalization
```

(continues on next page)

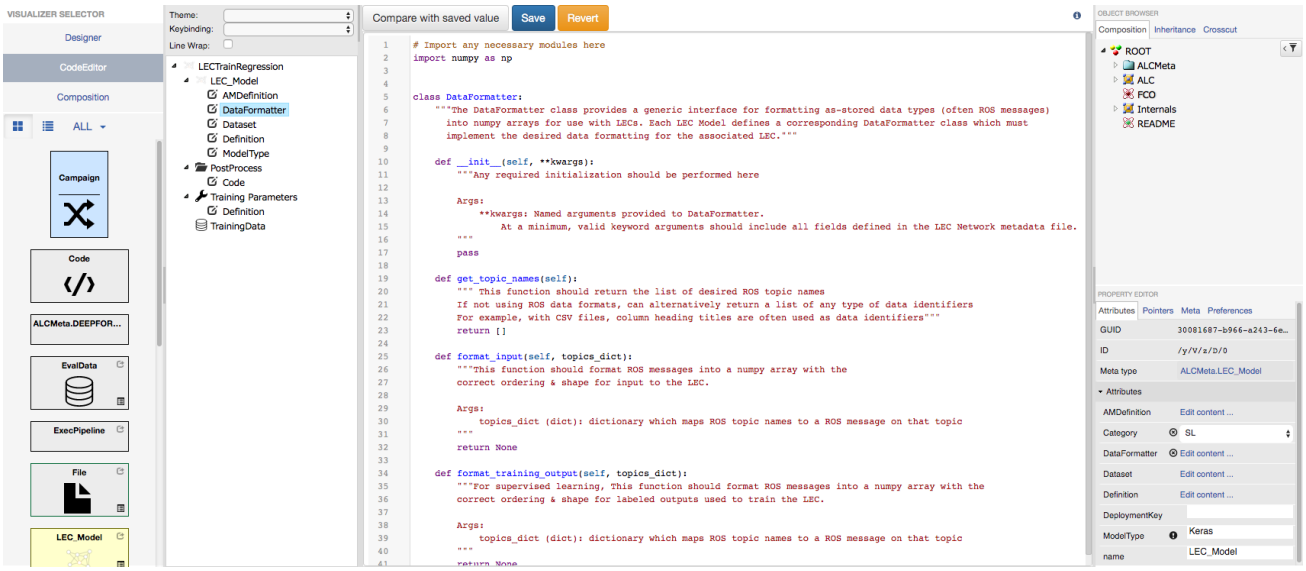


Fig. 9.20: LEC Construction Code Editor

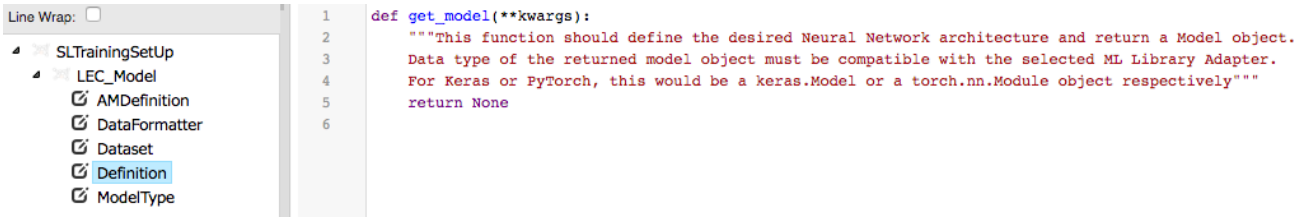


Fig. 9.21: LEC Definition Boilerplate Code

(continued from previous page)

```
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Flatten
from keras.layers.core import Dropout
from keras.layers.core import Dense
from keras import backend as K
from keras.utils import plot_model

#Builds the DAVE2 Architechture described in:
# https://devblogs.nvidia.com/deep-learning-self-driving-cars/
class DAVE2:
    @staticmethod
    def build(height,width,depth,classes,is_scaled=False):
        # initialize the model
        model = Sequential()
        #the shape of our image inputs
        inputShape=(height, width, depth)
        #if we are using "channels first" update the input shape
        if (K.image_data_format()=="channels_first"):
            inputShape=(depth, height, width)

        #first set of CONV=>RELU=> POOL layers
        model.add (Conv2D(24,(5,5),strides=(2,2), padding="valid",input_
↪shape=inputShape))
        model.add(Activation('relu'))

        #second set of 5x5 CONV=>RELU=> POOL layers
        model.add (Conv2D(36,(5,5), strides=(2,2),padding="valid"))
        model.add(Activation('relu'))

        #third set of 5x5 CONV=>RELU=> POOL layers
        model.add (Conv2D(48,(5,5), strides=(2,2),padding="valid"))
        model.add(Activation('relu'))

        #first set of 3x3 CONV=>RELU=> POOL layers
        model.add (Conv2D(64,(3,3), padding="valid"))
        model.add(Activation('relu'))

        #second set of 3x3 CONV=>RELU=> POOL layers
        model.add (Conv2D(64,(3,3), padding="valid"))
        model.add(Activation('relu'))

        #set of fully connected layers
        model.add(Flatten())
        model.add(Dense(1164))
        model.add(Activation('relu'))
        model.add(Dense(100))
        model.add(Activation('relu'))
        model.add(Dense(10))
        model.add(Activation('relu'))

        #output
        model.add(Dense(classes))
        if not is_scaled:
            model.add(Activation('tanh'))
```

(continues on next page)

(continued from previous page)

```
        return model

def get_model(**kwargs):
    #This function should define the desired Neural Network architecture and return
    #a Model object. Data type of the returned model object must be compatible with
    #the selected ML Library Adapter. For Keras or PyTorch, this would be a keras.
↪Model
    #or a torch.nn.Module object respectively

    model=DAVE2.build(66,200,3,1)
    return model
```

The translation functional code to format the input and output data for training, which consists of DataFormatter and Dataset attributes, are also defined in this section of the model. The **DataFormatter** attribute provides a generic interface for formatting as-stored data types (often ROS messages) into `numpy` arrays for use by the LECs. This function should be modified by the designer to fit the needs of their LEC model. Boilerplate code is provided for the `DataFormatter` file and is shown below. The **Dataset** attribute allows the user to create custom code to load the desired dataset in preparation for training by overloading the machine learning library dataset function. When using custom code for the dataset, remember to set the `DATASET_NAME` parameter to `CUSTOM`. For the car tutorial, both the uploaded dataset (jpeg image set) and the generated experiment data (ROS bag file image) are non-standard inputs, so a custom Dataset was created (different for each training data input types). The AMDefinition will be discussed further in the *Assurance Monitoring* section.

```
# Import any necessary modules here
import numpy as np

class DataFormatter:
    #The DataFormatter class provides a generic interface for formatting as-stored␣
↪data
    #types (often ROS messages) into numpy arrays for use with LECs. Each LEC Model
    #defines a corresponding DataFormatter class which must implement the desired data
    #formatting for the associated LEC.

    def __init__(self, **kwargs):
        #Any required initialization should be performed here

        #Args:
        #    **kwargs: Named arguments provided to DataFormatter.
        #        At a minimum, valid keyword arguments should include all fields␣
↪defined in
        #        the LEC Network metadata file.

        pass

    def get_topic_names(self):
        #This function should return the list of desired ROS topic names
        #If not using ROS data formats, can alternatively return a list of any type␣
↪of data
        #identifiers
        #For example, with CSV files, column heading titles are often used as data␣
↪identifiers
        return []

    def format_input(self, topics_dict):
        #This function should format ROS messages into a numpy array with the
        #correct ordering & shape for input to the LEC.
```

(continues on next page)

(continued from previous page)

```
        #Args:
        #    topics_dict (dict): dictionary which maps ROS topic names to a ROS␣
↪message on
        #    that topic
        return None

    def format_training_output(self, topics_dict):
        #For supervised learning, This function should format ROS messages into a␣
↪numpy
        #array with the correct ordering & shape for labeled outputs used to train␣
↪the LEC.

        #Args:
        #    topics_dict (dict): dictionary which maps ROS topic names to a ROS␣
↪message on
        #    that topic
        return None
```

As indicated when discussing the hyper-parameters, several model functions can be customized to allow the user more control of parameters during the training process (such as loss, optimizers and metrics). To handle the additional desired parameters, a `get_` function can be created within the model `Definition` file. Below is an example of a function added to allow the exposure of the learning rate, beta 1 and beta 2 parameters for either an Adam or SGD optimizer algorithm. The training parameters table will include a new parameter (`optimizer_name`) to indicate which of the two targeted optimizers is desired.

```
import keras
def get_optimizer(**kwargs):
    #This is a function to modification of the optimizer parameters for both SGD and␣
↪Adam
    #optimizers.

    opt_type=kwargs.get('optimizer_name')
    lr=kwargs.get('lr')

    if(opt_type.lower()=="adam"):
        beta_1=kwargs.get('beta_1')
        beta_2=kwargs.get('beta_2')
        opt = keras.optimizers.Adam(lr=lr,beta_1=beta_1,beta_2=beta_2)
    elif(opt_type.lower()=="sgd"):
        opt = keras.optimizers.SGD(lr=lr)
    else:
        print('invalid optimizer name for this model - select Adam or SGD')

    return opt
```

As with the Data Collection, the training of the LEC is performed by running **Launch Experiment** plugin. For SL training, the ROS nodes will not be running, so the `Generate ROS Launch files` option should be set to `FALSE`. The **Result** block will hold the results of all training experiments and provide access to the resulting jupyter notebooks that can be run to see the performance metrics setup for the run, shown in Fig. 9.22.

Note: Since the toolchain is a shared resource with other projects and/or designers, it is best to be mindful of how long a training run might take to execute. As an example, a run was done with 600 epochs using the SGD optimizer with a batch size of 256, the training session lasted for 4 hours and 38 mins. The Adam optimizer can be faster, so just

Fig. 9.22: Training Result Artifact Index

consider effects of runs on other model efforts.

The trained model from a training run can be found by opening the file browser in the jupyter notebook, which opens to the location with the training results including the model weights (model.hd5) as seen in Fig. 9.23.

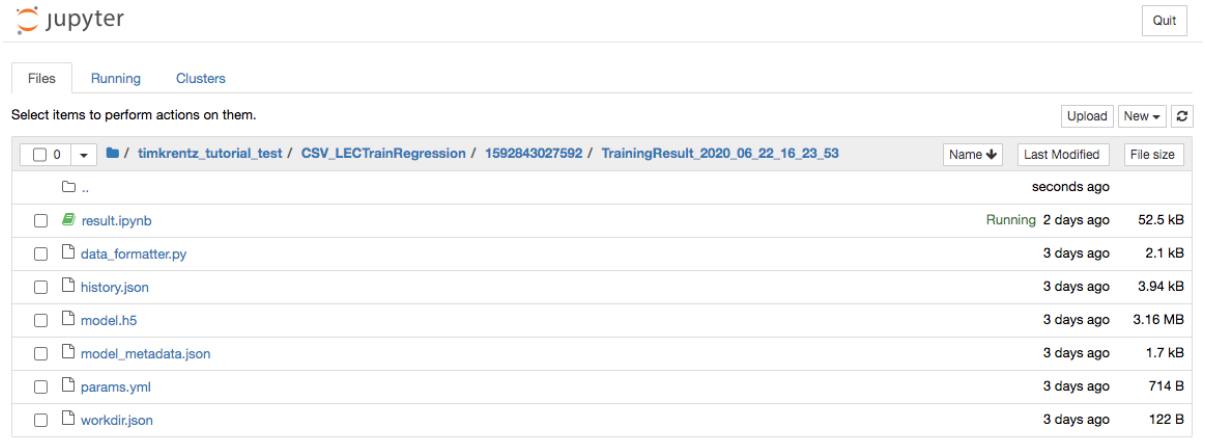


Fig. 9.23: Train Model Location

9.2.2 Reinforcement Learning

Reinforcement Learning (RL) has evolved to be a powerful data-driven technique in which the learning (or training) happens in a closed-loop interaction between the agent and environment. Unlike supervised learning, RL does not require labeled data during the training process. The ALC Toolchain uses an actor-critic method where the goal is to learn a policy of selecting the best action from a list of actions for a given state, that maximizes the reward function. The agent receives a positive reward for correct actions, and a negative reward otherwise during the training phase. The agent utilizes the learned actions which are normally stored as tables, or trained neural networks.

The learning setup definition includes the LEC model to be trained, underlying reinforcement learning algorithm, associated reward functions for agent action given the current state of the system, and any training hyper-parameters. Using the model workflow tool (discussed later), a LEC can be trained over many simulated and randomly-generated scenarios (episodes). During each episode, the LEC explores the possible action space while using the provided reward function to observe which control actions produce desirable results for the system. Each episode can be specified to last a maximum time-limit or step-size.

Just as in the experiment model setup, the reinforcement learning model includes an assembly model (**AssemblyModel**) where the specific component implementation(s) are selected from the possible alternates. Fig. 9.24 show a typical setup of a reinforcement learning model. **Mission** and **Environment** (or WorldModel) blocks provide information on the goals and system setup of the training session that will be launched. **Result** and **PostProcess** blocks provide ways to interact with the resultant experiment data, like in the experiment setup already defined. Remember that the AssemblyModel and Environment blocks are instances copied from the system modeling components.

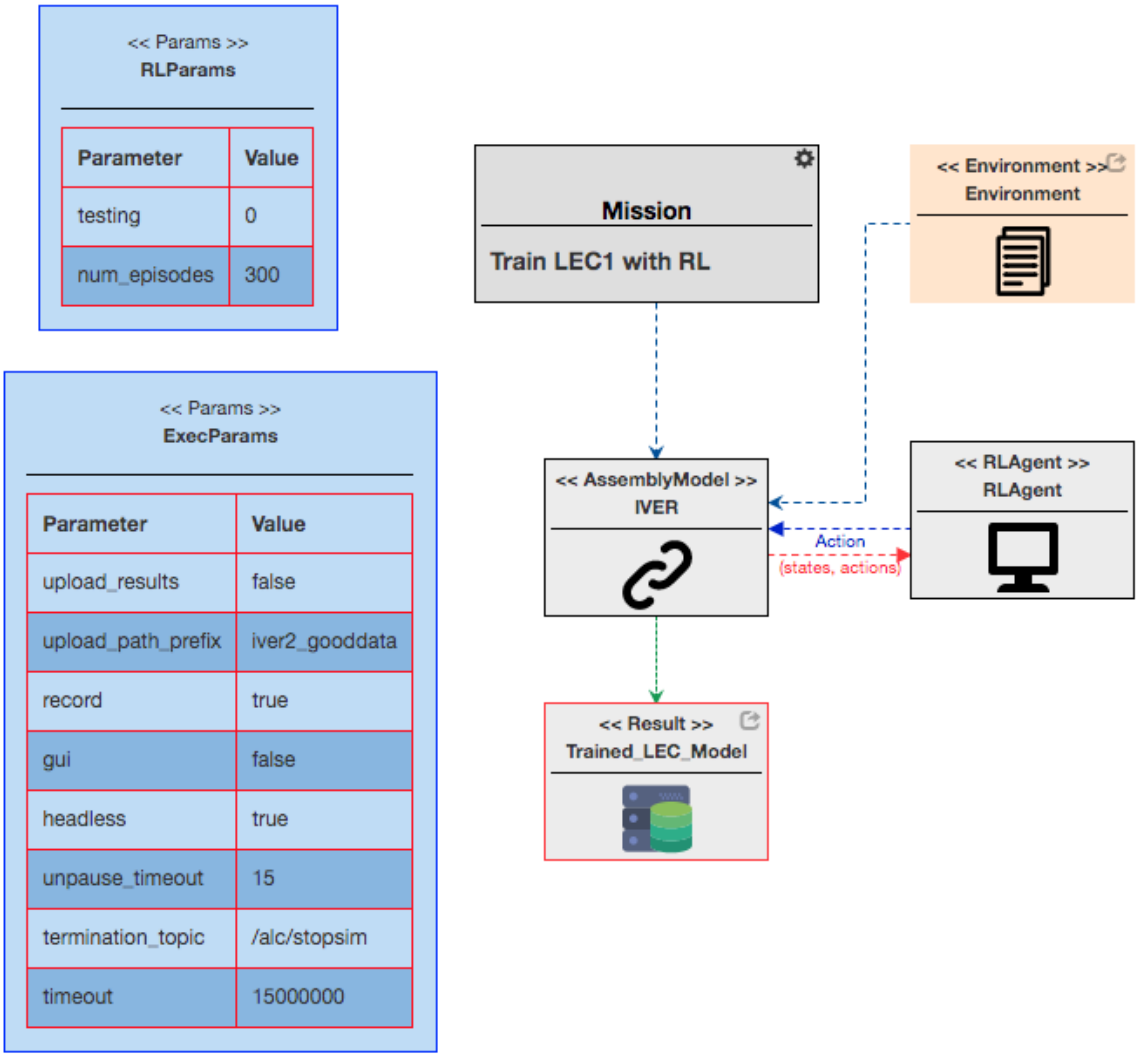
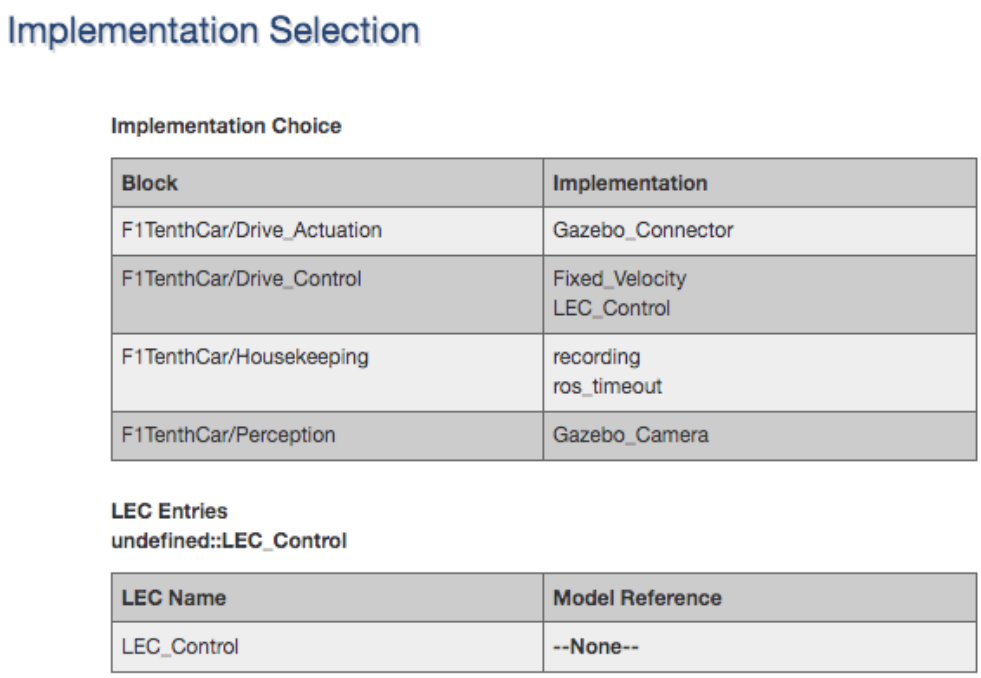


Fig. 9.24: Reinforcement Learning Model Definition

Since each training model is targeting specific LEC models to train, the assembly model needs to be configured to select the appropriate LEC model reference for the LEC being trained. This is done by drilling down into the assembly model and selecting the Implementation view (from top left panel), as shown in Fig. 9.25. To start training the LEC from scratch, the LEC Entries for the desired LEC (`LEC_Control` for the car tutorial) is set to *None*. To retrain starting from a pre-trained LEC model select that available LEC model to use.



Implementation Selection

Implementation Choice

Block	Implementation
F1TenthCar/Drive_Actuation	Gazebo_Connector
F1TenthCar/Drive_Control	Fixed_Velocity LEC_Control
F1TenthCar/Housekeeping	recording ros_timeout
F1TenthCar/Perception	Gazebo_Camera

LEC Entries
undefined::LEC_Control

LEC Name	Model Reference
LEC_Control	**--None--**

Fig. 9.25: LEC Selection for RL Training

The parameter block (**Params**) captures parameters relevant to the specific training exercise and how to configure the training algorithm. Below is a list of some typical parameters and their meaning, the list used should be tailored to the training configuration needs.

Parameter	Description	Units
testing	Used by RL training setup code	testing mode (0) training mode (1)
num_episodes	# randomly-generated simulated scenarios to run	integer
record	Record simulation results in a ROS bag file	true/false
gui	Use GUI in Gazebo simulation runs	true/false
unpause_timeout	Time to wait before starting simulation	seconds
timeout	Simulation execution timeout	seconds
termination_topic	Topic used to terminate simulation	string
verbose	Detailed logging from ML libraries	true/false

Note: Typically `gui` is set to false since run are done remotely and GUI will not be available to user

Note: Some elements of a simulation take longer to load, so an `unpause_timeout` is set to allow all resources to be loaded before starting the simulation. There is also a `timeout` for the simulation execution so that the execution runner can terminate the simulation.

Note: Recorded ROS bag files can be found in the results folder of the simulation run.

Note: The typical `termination_topic` is "/alc/stopsim". The execution runner will listen for the indicated topic

and will terminate the dockers once a message is received.

The LEC model definition, reinforcement learning algorithm and associated reward functions are specified in the **RLAgent** block, shown in Fig. 9.26. Training a LEC with reinforcement learning requires the designer to define a reinforcement learning agent and an environment where the agent can act. The RLAgent model has two sets of wrapper code for executing the RL training: RLAgent and RLEnvironment. `RLAgent (or Code)` outputs the actions from utilizing the underlying neural network (using the step and save methods), depending on the mode of operation (training or testing). `RLEnvironment` computes the state and rewards values (using a step method), along with the termination criteria based on the current states and actions determined by the RLAgent code. Below is the boilerplate code provided for these two wrappers. The designer should fill in logic for the provided functions to fit their needs.

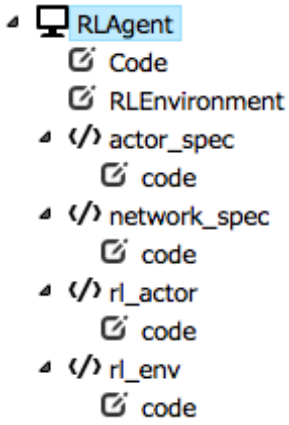


Fig. 9.26: Reinforcement Learning Agent Code Definition

- RLAgent Wrapper Boilerplate Code

```
import os
from alc_utils.rl_agent import RLAgentBase
from rl_actor import RLActor
class RLAgent(RLAgentBase):
    def __init__(self, model_dir, testing, env, **kwargs):
        self.model_dir = model_dir;
        self.testing = testing;
        self.agent = RLActor(model_dir,testing,env,**kwargs)
        self.model = self.agent.agent[0].model;
        self.weights_result= os.path.join(model_dir,'weights','model');
        self.action = None;
        self.observation = None;
        self.rewards = None;
        self.terminal = None;
        self.episode_rewards = 0;

    def save(self):
        #this is implemented by tensorforce agent
        self.model.save(directory=self.weights_result)

    def step(self, states, reward, done):
        self.observation = observation;
        self.reward = self.reward;
        self.terminal = done;
        self.action, terminate = self.agent.learn(observation, reward, done);
```

(continues on next page)

(continued from previous page)

```
        return self.action, terminate

    def getAction(self):
        return self.action;

    def getEnv(self):
        return self.agent.env;
```

- RLEnvironment Wrapper Boilerplate Code

```
from alc_utils.rl_environment import RLEnvironmentBase
from rl_env import Env
class RLEnvironment(RLEnvironmentBase):
    def __init__(self):
        self.env = Env();
        self.states={};
        self.rewards = 0;
        self.terminal = 0;

    def step(self, observation, action):
        self.states, self.terminal, self.reward =\
        self.env.execute(action, observation);
        return self.states, self.terminal, self.reward

    def reward(self):
        return self.rewards;

    def done(self):
        return self.terminal;
```

The designer provides all the code necessary to define the neural network (`network_spec`) and run the training simulations (`actor_spec`) needed for the learning executions using json files. Example code is available and should be adjusted to fit the design. This example utilizes the Tensorforce library. The `rl_actor` code will load these json files and utilize the RLAgent wrapper code in the training process. The `rl_env` is the execution code utilized by the RLEnvironment wrapper code.

- network_spec example

```
[
    {
        "type": "dense",
        "size": 32,
        "activation": "relu"
    },
    {
        "type": "dense",
        "size": 32,
        "activation": "relu"
    }
]
```

- actor_spec example

```
{
    "type": "vpg_agent",

    "update_mode": {
```

(continues on next page)

(continued from previous page)

```
        "unit": "timesteps",
        "batch_size": 64,
        "frequency": 64
    },
    "memory": {
        "type": "replay",
        "include_next_states": false,
        "capacity": 100000
    },

    "discount": 0.9,
    "gae_lambda": 0.5,

    "optimizer": {
        "type": "adagrad",
        "learning_rate": 1e-3
    },

    "baseline_mode": "network",
    "baseline": {
        "type": "mlp",
        "sizes": [32, 32]
    },
    "baseline_optimizer": {
        "type": "multi_step",
        "optimizer": {
            "type": "adagrad",
            "learning_rate": 1e-3
        },
        "num_steps": 4
    },

    "saver": {
        "directory": "/mnt/saved_weights",
        "seconds": 60
    },
    "summarizer": {
        "directory": "/mnt/summary",

        "labels": ["graph", "total-loss"]
    },

    "execution": {
        "type": "single",
        "session_config": null,
        "distributed_spec": null
    }
}
```

Additional code blocks can be added by the user (using the Designer view) to provide supporting implementation code, shown in Fig. 9.27. This code will be hosted in the same directory as the RLAgent and RLEnvironment code. The directory was defined in the LEC block model as the deployment key attribute, see *LEC Models*.

To run the LEC in the system, the block library LEC definition will have ROS code that instantiates the RLAgent and RLEnvironment classes and interface with the rest of the ROS system. This code will include the following functions:

- translate messages into a state dictionary that is provided to the RLAgent

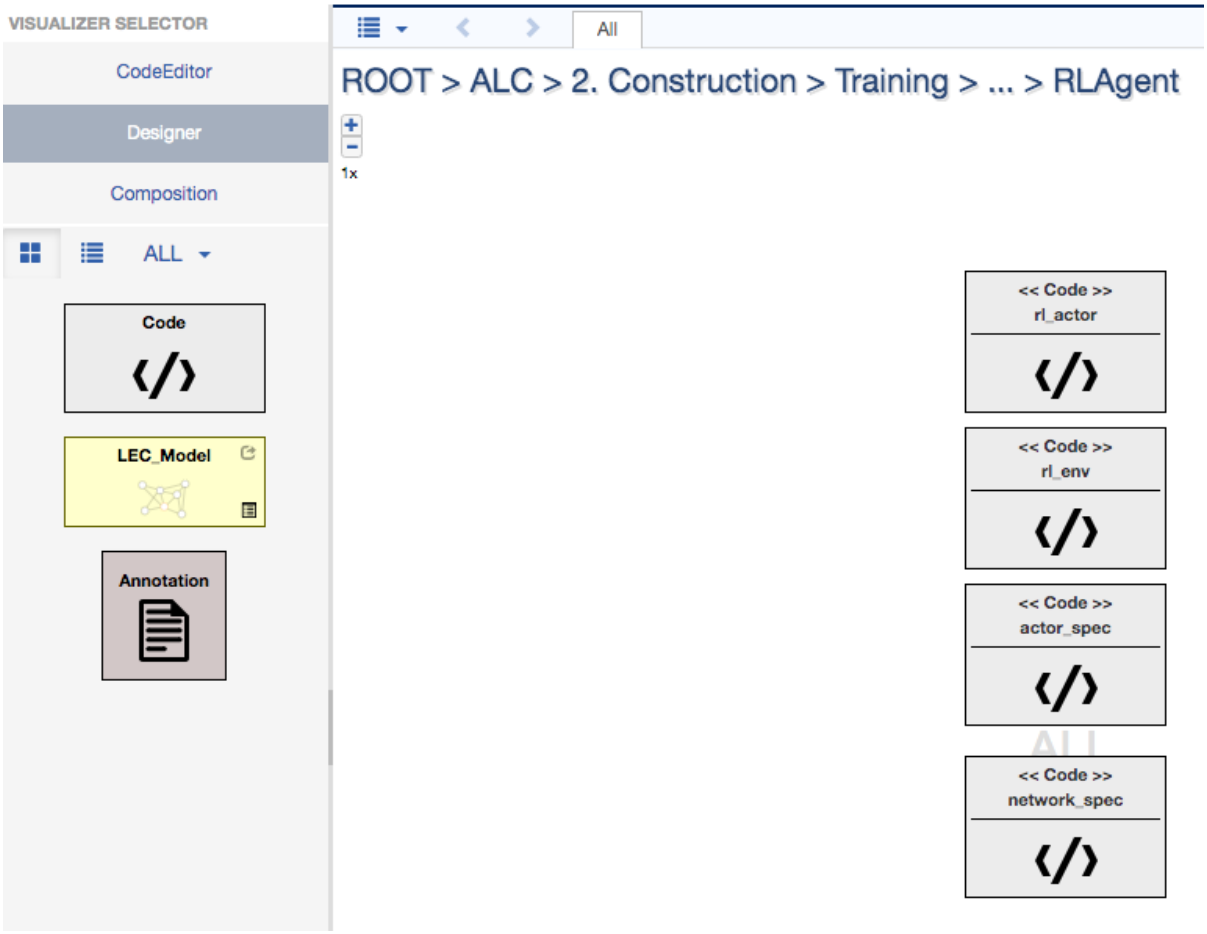


Fig. 9.27: Designer View for Adding Code Files

- receive and convert actions into messages and publish them for the rest of the system to utilized
- receive termination commands and act on them as desired by the designers
- saving training model weights

The trained reinforcement learning model includes all the RLAgent code, configuration files, and the trained weights. During training, it is recommended that user save the weights within the deployment key folder by using the save method in RLAgent `Code` file. This ensures that when the trained model is deployed, the associated code and weights are available for the model to be loaded. Alternately, to keep re-training from the last trained stage, the user might use an additional fixed weights directory inside the RLAgent setup (`rl_actor` initialization code). The weights could be loaded from this directory when the LEC model in the assembly is set to `None` or does not have any pre-trained weights. Thereafter, when the LEC model in the Assembly is set to a trained model, the weights will be loaded from the trained model instead of the set weights folder.

9.3 LEC Evaluation

The training metrics are useful for determining how well a particular ANN has learned to approximate the provided training data set. However, these metrics are not always good indicators for how well a trained LEC will perform when deployed as part of a larger system. This is often a result of an incomplete training data set which does not capture a sufficient range of possible inputs. For LECs used as control components, this issue is more pronounced since a trained LEC controller may drive a system into previously unseen states which were not represented in the training data set. Instead of relying on training metrics, a trained LEC can be evaluated as part of an integrated system using an experiment model (similar to ones created for data collection) or an evaluation can be performed using a test dataset. The evaluation block is primarily meant for offline evaluation of the LEC performance.

9.3.1 LEC System Testing

Experiments are setup as indicated in the *Data Collection* section. The main difference here would be that this experiment is run using a specified trained LEC. In the experiment setup, a newly trained LEC model is selected in the assembly implementation view (under LEC Entries) as shown in Fig. 9.28. A post processing block with additional Python code can be used to create customized graphs and analysis of the resultant data. Results can also be provided to a simulator (like gazebo) to provide a visual perspective on the results of the experiment run, in the form of a bag recording.

For reinforcement learning, the trained LEC is evaluated by executing the reinforcement learning setup in a non-training mode. The hyper-parameters include a `testing` parameter to indicate when the LEC is in training mode (0) or testing mode (1). This testing will not update the LEC network weights.

Warning: When designing your system for experimentation, keep in mind that LEC control components will be utilizing the machine language library (such as Keras and TensorFlow) and may take longer to start running than simple components. In the car tutorial, the `LEC_Control` component predicts the steering angle for the car to drive and the velocity is set at a fixed rate by a separate component (`Fixed_Velocity`). If the velocity is requested before the LEC is ready to predict, the car will drive forward some time before any steering angle information is provided. This is not a preferred mode of operation.

9.3.2 LEC Model Evaluation With Additional Data

A new dataset could be used to test data points not included in the original training dataset. This is done by creating a Jupyter notebook that has access to the LEC model (i.e. weights and metrics), and any new evaluation or validation datasets. Fig. 9.29 shows an example where evaluation data is used to test the LEC model and the predicted steering

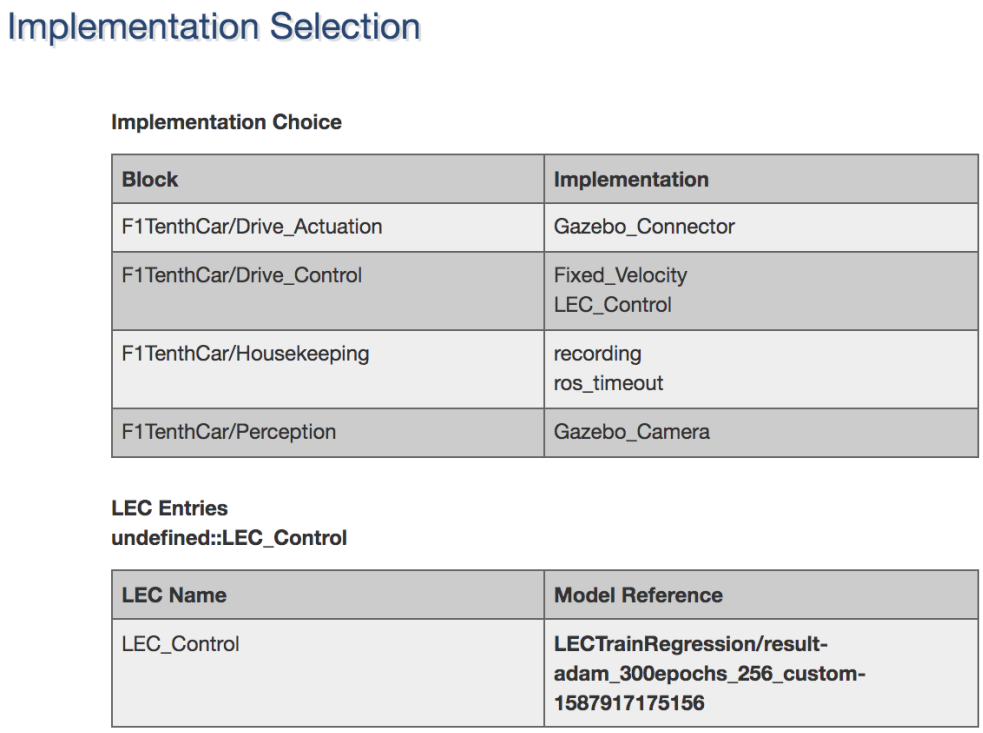


Fig. 9.28: Designer View for Adding Code Files

angle is plotted against the actual steering angle (shown in Fig. 9.30). The data used in the evaluation can be either evaluation data (**EvalData**) or **ValidationData**. Go into these blocks to choose the desired dataset. The **LEC_Model** allows the selection of the trained LEC to be evaluated (inside the block). The **PostProcess** will provide a place to place code that can be reused for evaluation of different trained LECs.

To run this evaluation, the user can launch an experiment by selecting the **Launch Experiment** plugin under the top left play button, see Fig. 9.31 for the form. This time, the `Setup Jupyter Notebook` option should be selected so that a notebook will be created. This notebook (shown in Fig. 9.32) will appear on the model. This notebook is also referenced in the **Result** block in the `Notebook` column. It can be executed (and viewed) by either double clicking on the new notebook element on the model or clicking the link in the results list. If desired, it can be removed from the model without loosing a reference to this run.

The initial Jupyter notebook, shown in Fig. 9.33, will establish links to the evaluation data (`eval_data`), validation data (`val_data`) and model data (`lec_data`) selected. As with the notebooks in the experiment results, the user will `Run` the first 6 cells of the notebook. The last cell ([6]) will load in the code from the **PostProcess** and should be rerun to execute the evaluation code. The evaluation code in the **PostProcess** block should be written Python. The following code blocks show how to access the available data.

- To access the evaluation or validation datasets, the data directory location can be found in the provided JSON data information. An example for the `eval_data` is shown below.

```
path = eval_data['EvalData'][0]['directory']
```

- To access the model data, the following code snippet shows how to get the directory location from `lec_data` and use it to load the model data for use in the evaluation code.

```
dir = lec_data['LEC_Model']['directory']
dir = dir + '/model.h5'
```

(continues on next page)

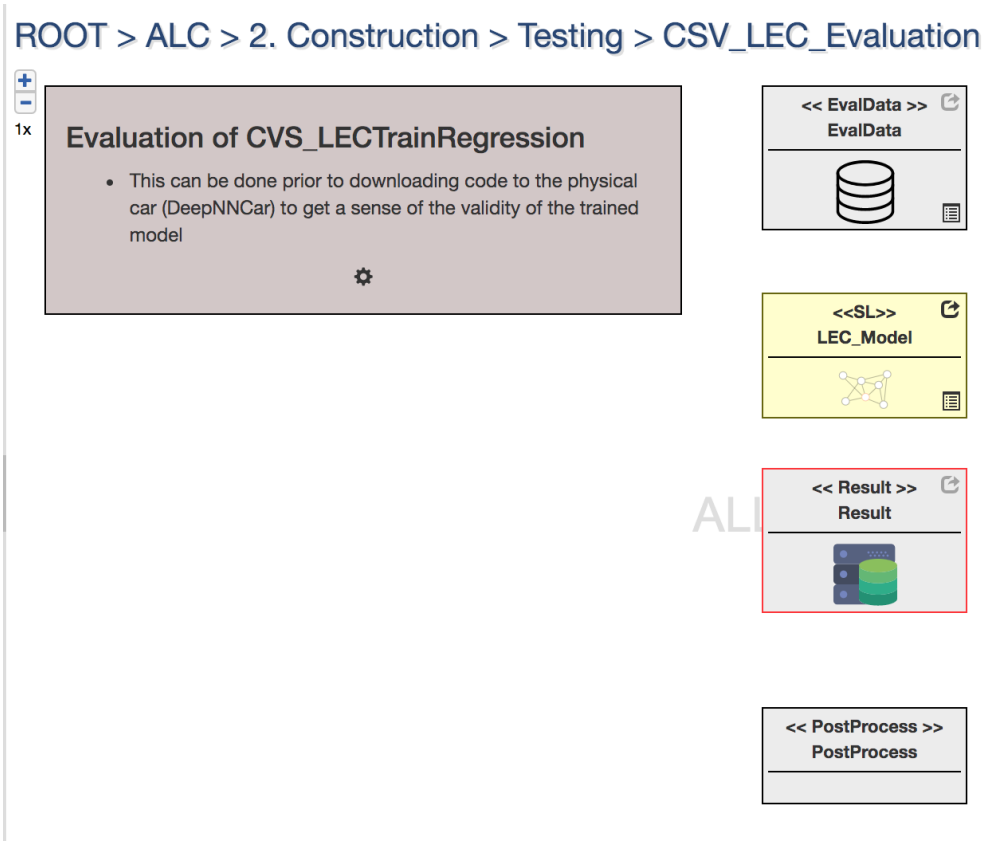


Fig. 9.29: LEC Model Evaluation Setup

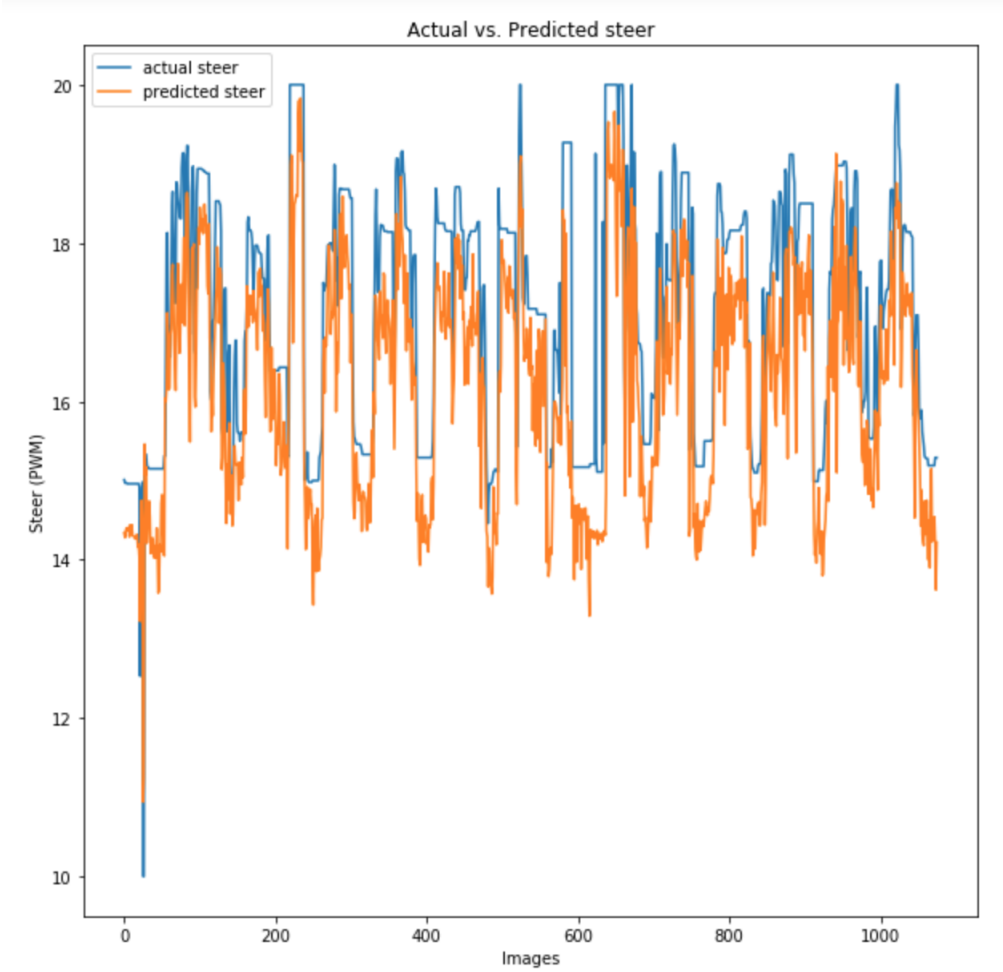


Fig. 9.30: LEC Model Evaluation Results

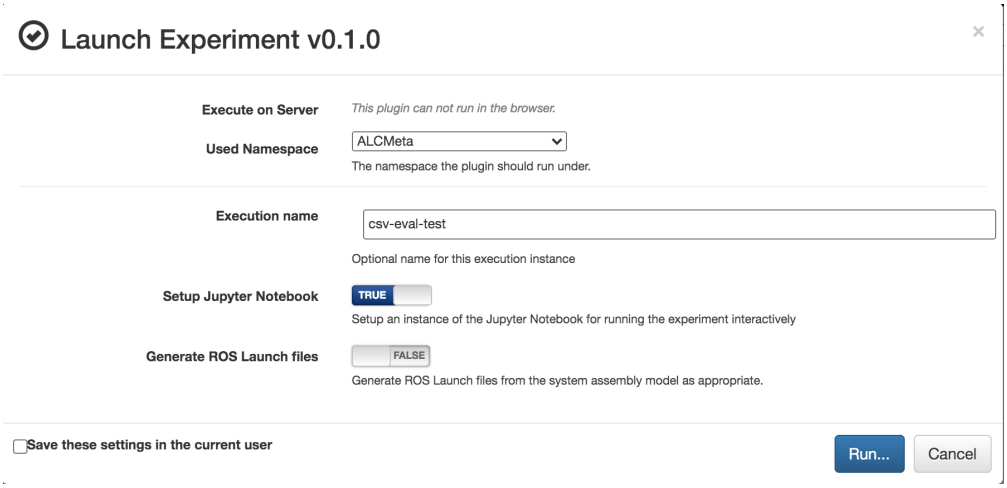


Fig. 9.31: Launch Experiment for LEC Evaluation

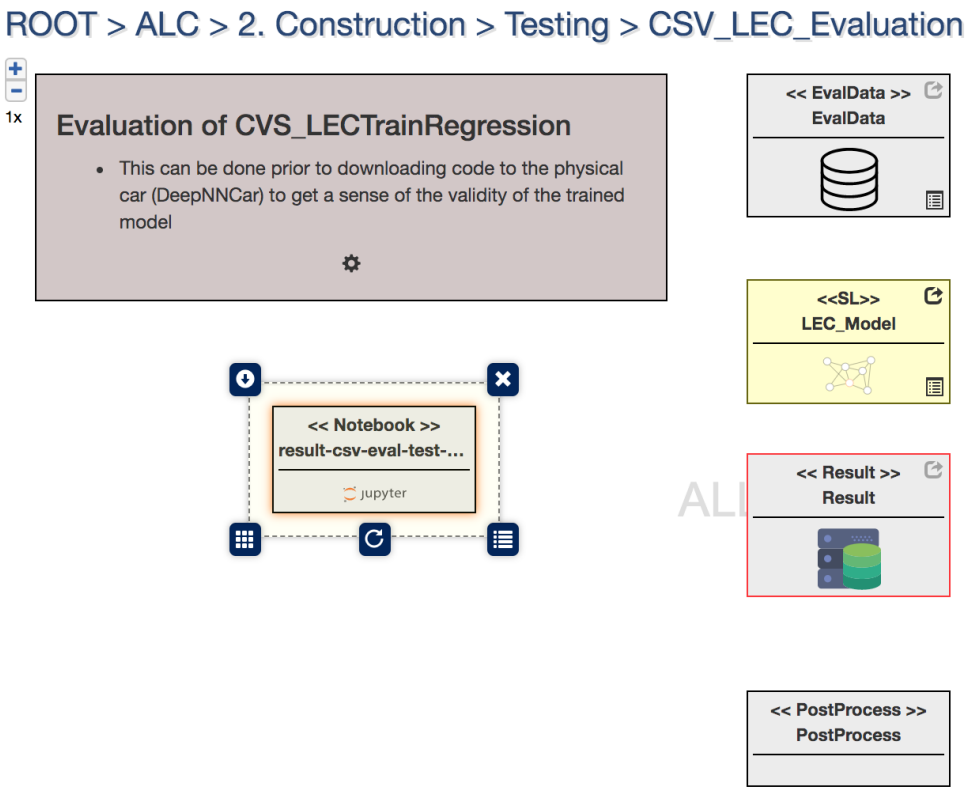


Fig. 9.32: Jupyter Notebook from Launched Experiment

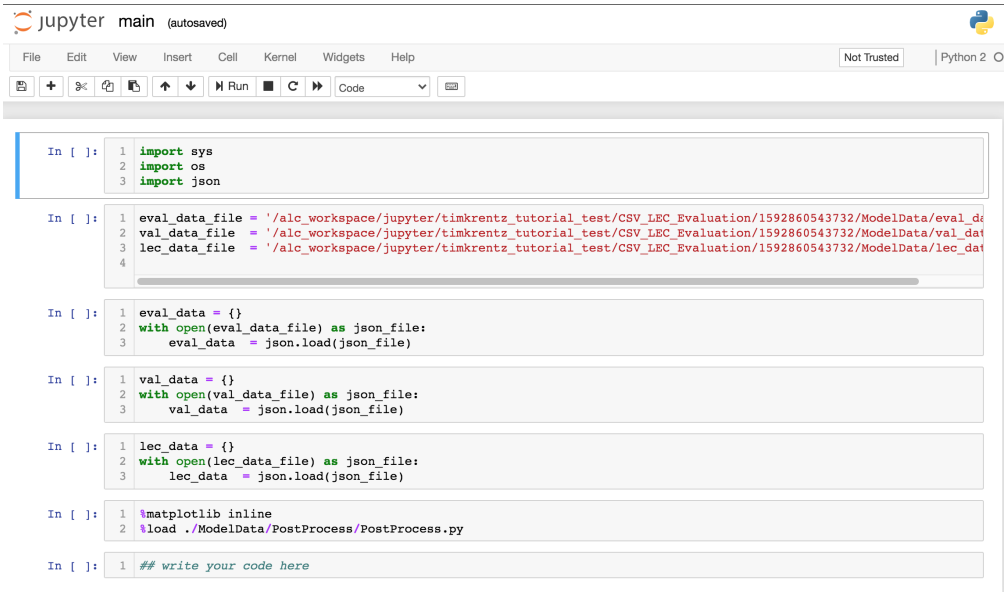


Fig. 9.33: Initial LEC Evaluation Jupyter Notebook

(continued from previous page)

```
model = load_model(str(dir))
```

CHAPTER

TEN

ASSURANCE MONITORING

The training of the assurance monitors (AMs) are based on concepts of Variational Auto Encoders (VAE), Siamese Network (Selective classification), Support Vector Data Description (SVDD), Inductive Conformal Anomaly Detector (ICAD), Saliency Mapping and k-nearest neighbor classifier (kNN). Each assurance monitor is trained to monitor a specific task and is paired with a single trained LEC model. In a supervised learning setup, an LEC model is trained against the collected dataset, with the option to train runtime assurance monitors as well. In other words, the assurance monitor is trained at the same time as the LEC model. For reinforcement learning, the assurance monitors are trained using the generated data when the associated trained LEC is deployed in the system.

The trained LEC association happens using the **Selector** visualizer inside the LEC block defined in the assurance monitoring training model. Assurance monitors are setup similar to the LECs, with a specified training dataset and model definition. The DataFormatter attribute of LEC Model must be defined for the AM. Post processing in the LEC training and evaluation step should include the assurance monitor predictions.

Specific hyper-parameters can be made available in a separate parameter block for assurance monitor, including turning on and off the training for this block (lec_assurance_monitor).

The toolchain currently supports the following assurance monitoring technologies.

1) Selective Classicitaion:

   Assurance monitor for Classfication models based on an underlying Siamese network. The monitor computes the LEC output as well as a confidence measure on whether the LEC output can be trusted. This approach has two variants - a snapshot based approach that outputs the assurance monitor decision based on a snapshot input, and a sequence or window based approach that outputs the assurance monitor decision based on a sequence of time-stamps.

2) VAE:

   Assurance monitor based on Variational Auto Encoder (VAE) network for regression models.

3) VAE Regressor:

   Assurance monitor uses VAE and a regressor network (internally) to predict the LEC outputs.

4) VAE Classification:

   Assurance monitor based on Variational Auto Encoder (VAE) network for classification models.

5) LRP:

   Assurance monitor uses VAE and Layer-wise Relevance Propagation.

6) Saliency Map:

   Assurance monitor uses VAE and a Saliency map based on Visual back propagation.

7) SVDD:

   Assurance monitor based on Support Vector Data Description (SVDD).

8) KNN:

   Assurance monitor based on k-Nearest Neighbour(KNN). This is currently not being used because of the large memory foot-print.

CHAPTER

ELEVEN

VERIFICATION AND VALIDATION

Real-time reachability analysis

1) The real-time reachability analysis examples work with ALC_BlueROV2 model.
2) This involves running the reachability analysis code at run-time (with the simulation) and and compute the reachability of uuv to check if there will be a collision in projected window.

Other Examples

Other Verification and Validation examples (based on the NNV toolset) include

1) System-ID and validation: Learn the System ID model and validate it against data sets for UUV.
2) Single-Step reachability: Gets sensor data from the simulation data set and predicts the position of the uuv taking into account model uncertainity and controller uncertainity
3) Multi-Step reachability: Repeats the single-step reachability over all time steps in the simulation to project the possible locations of UUV and analyse possibilities of collision with static obstacles.
4) Robustness analysis: Analyze robustness of LEC to classification/ segmentation when the input image is pertubed by different attacks.

CHAPTER

TWELVE

ASSURANCE CASE

The toolchain supports construction of static assurance cases using an extended version of Goal Structuring Notation.

More recent updates to the assurance case support in the toolchain are in line with the paper

```
Hartsell, C., Mahadevan, N., Dubey, A., & Karsai, G, "Automated Method for Assurance␣
↪Case Construction from System Design Models", in 5th International Conference on␣
↪System Reliability and Safety (ICSRS), 2021.
```

These updates include

1) **Model Cross-referencing and Assurance Case Construction**

   Construction of Assurance case based on cross-referencing the models in the toolchain.

   These models and cross-references are as follows:

   Models

   i) Requirements model

   ii) Architecture (System) model

   iii) Functional Decomposition model,

   iv) Hazards model

   v) Dynamic Risk Mitigation (DRM) model.

   Cross-references in the Models

   Cross-references details in each of the individual models listed above are as follows.

   i) The lowest-level functions in the Functional decomposition model are cross-referenced to the components or subsystems in the architecture model that implement the specific function. The cross-referenced component could be a

      - Hardware component,
      - Simulation of a Hardware component,
      - Software component such as ROS nodes that implement perception, control, mapping with (or without) Learning-Enabled components
      - Software components related to contingency management implemented with (or without) Behavioral Trees.

   ii) System model: The underlying fault propagation graph in the System model includes degradation *Effects* of the failure as well as mitigation *Response* to the failure. These two nodes (*Effect* and *Response*) are cross-referenced to the appropriate function that degrades in the presence of failure or mitigates the failure.

iii) Dynamic Response model: The *Barrier* nodes in this threat propagation model are cross-referenced to the appropriate

functions in the functional decomposition model that implement the barrier or mitigation.

The DRM model is also cross-referenced to the Hazards that are associated with the threat propagation model.

Assurance Case model

Each GSN node in the assurance case can be cross-referenced to the relevant ALC model. These include

i) Requirement model relevant to a *Goal*, *Context* or *Assumption* node in the GSN argument.

ii) Function model that is the subject of a *Goal* or *Assumption* node in the GSN argument.

iii) Hazard and/or DRM model that is the subject of discussion in a *Goal* or *Context* node.

iv) Evidence references to *Solution* nodes include *Testing*, *Evaluation*, *Workflow* and *Verification* models.

Note: As an example, please refer to the assurance case model "ALC/ 3. V&V/ Assurance/ BlueROV AC" in the BlueROV2 project.

2) **Automated generation of Assurance-Case argument**

The toolchain includes the implementation of the automated assurance case generation discussed in the paper (link).

This tool currently has been implemented as an interactive session with user inputs.

In order to try this, the user needs to open an ALC/IDE session as discussed in the initial part of *Working with IDE*

1) Once the IDE is opened in a browser, open a terminal and type the following command to get started

```
/run_assurance.sh <ALC_Model_Name>  [model-owner default=admin]

e.g.

/run_assurance.sh ALC_BlueROV2 admin
```

It is possible that initially the user is asked to accept an ssh connection. Please type *yes*.

Thereafter, the cross-referenced model is loaded in memory for generation of the assurance- case.

2) The next questions relates to the initial pattern to choose from. Choose the number related to the *Risk reduction pattern*.

3) The user is prompted to choose the initial system model. Please choose the system model with the shortest path as the other model relates to an instance of the system model inside the assembly model.

4) Thereafter, the tool explores the cross-references in the model and asks relevant question. With regards to the admin_ALC_BlueROV2 model, the questions include

i) The Hazard to choose from for Risk reduction. Since the Obstacle Encounter Hazard is developed, we will choose this.

ii) The DRM model related to the Hazard choosen above - we will choose Obstacle Encounter DRM.

iii) The pattern to apply for further analyzing the DRM - choose the only avaliable option of *Resonate pattern*.

iv) The next set of questions deal with the Barrier functions that one needs to consider as part of the argument. We will choose *all*.

v) Finally, the tool asks for a name for the generated assurance case model. Type in a name.

vi) Go to the model and open the ALC/3. V&V&A/ Assurance Model and refresh the browser. An assurance model of the name that you had typed should appear.

vii) Open the model. The nodes might need to be shuffled around a bit.

viii) The tool generates the high level assurance case arguments based on the pattern (as discussed in the paper mentioned at the beginning of this section).

ix) The user can improve upon the undeveloped goals in the model by using their arugment scheme.

CHAPTER

THIRTEEN

RESONATE

The toolchain supports resonate statistics computation for static and dynamic-assurance cases as described in the paper

```
Hartsell, C., Ramakrishna, S., Dubey, A., Stojcsics, D., Mahadevan, N. & Karsai, G,
↪"Resonate: A runtime risk assessment framework for autonomous systems", 16th␣
↪International Symposium on Software Engineering for Adaptive and Self-Managing␣
↪Systems, SEAMS 2021, 2021.
```

The statistics computation is based on the threat propagation path described in each Dynamic Risk Model (Bow-Tie Diagrams) Based on the data collected from running multiple scenarios, ReSonAte pre-computes the priors and conditional probabilities for the events described in the dynamic risk models. It also pre-computes the parameters for the runtime hazard rate estimation function.

While the prior probabilities are used in the static assurance arguments, the hazard rate estimation function is used as part of for dynamic-assurance monitor to compute at runtime the likelihood of occurence of the consequence event.

Statistics Computation

Resonate statistics computation is setup in the toolchain as an *Evaluation Setup* model.

An example of this can be found in the BlueROV2 model (ALC/2.Construction/Testing/Resonate_Stats_Computation).

- In this model, the "EvalData" block is populated with data collected from multiple scenarios.
- The model is executed by starting the "LaunchExpt" plugin a Jupyter Notebook is created.
- The plugin creates a Jupyter Notebook model for the user to step through and execute.
- The user can then click on the "<<Notebook>" model which brings up the Jupyter Notebook for the user to execute.
- An example run of the statistics computation can be seen by click on the Jupyter Notebook already present in this model.

Note: This is an initial version.

CHAPTER

FOURTEEN

UTILIZING WORKFLOWS

Workflow models define a sequence of actions to perform in order to automate tasks, such as data generation, LEC training, LEC performance evaluation and system verification. Typically operations in a workflow are dependent on results from one or more of the preceding steps in a prescribed order. Operations which share the same set of dependencies may be performed in parallel. Additionally, many workflows are iterative processes which are repeated until a predefined threshold is met. To capture these dependencies between operations, workflows are often represented as directed graphs, as shown in Fig. 14.1.

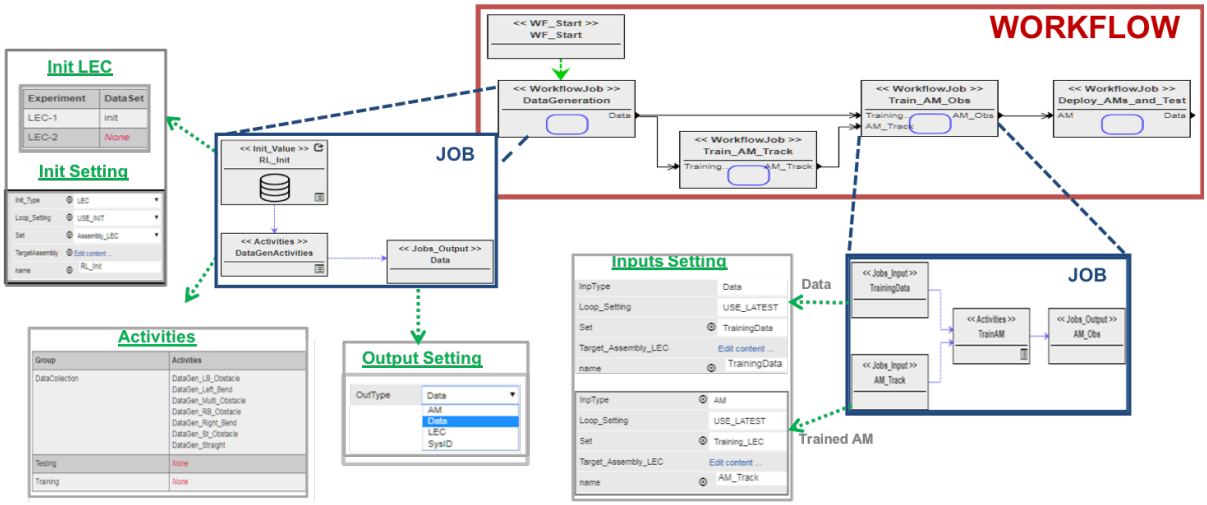


Fig. 14.1: Toolchain Workflow Overview

14.1 Workflow Jobs

Construction of a workflow model starts with the creation of job blocks, which describes the steps in the workflow process shown in Fig. 14.2.

These blocks can be:

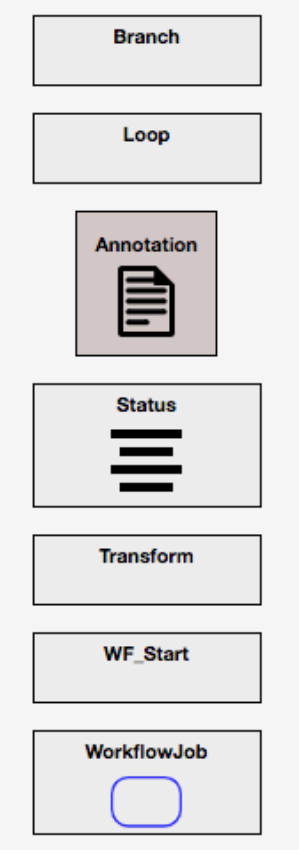


Fig. 14.2: Available Workflow Blocks

Blocks Types	ALC Name	Description
Job	WorkflowJob	define a single activity to perform
Iteration	Loop	specifies repetitive execution desired over a specific sequence of jobs in the work flow and interruption criteria
Decision	Branch	allow decision making during the workflow progression using inputs from a previous job and a user defined script
Result Processing	Transform	filter results from previous jobs using a user defined algorithm and pass the results to the next job, this can be to down-select or join results from multiple jobs

The tutorial **Train_Eval** workflow model (shown in Fig. 14.3) utilizes an **Loop** block to perform the hyperparameter search to train the LEC using the uploaded training dataset. The next step looks at the loss value of each of the resulting LECs to determine the one with the lowest loss using a **Transform** block and then uses the LEC in a system test for evaluation of the trained LEC's performance using a **WorkflowJob** block that runs a LEC testing experiment.

Status block is a tool specific block that must be available to allow the collection and creation of status information that will be available in the **Status** Visualizer (in the left panel at the top of the workflow model definition). No configuration is necessary for this block. For each execution of a workflow, a **WFExecStatus** block is added inside the status block and is populated with information from the workflow executor. This information is utilized by the status visualizer tool to provide information about the run. The user can choose to remove a **WFExecStatus** block when desiring to cleanup the status visualizer information, i.e. for old runs no longer relevant to the present work.

As each job block is defined (in particular input and output ports), they can be connected using arrows to indicate the next job to run and how the generated artifacts flow (represented with a solid black line directed from an output port to an input port). The work flow starts with a **WF_Start** block to show where the processing begins. When this block is connected to the first workflow block, the arrow will be represented as a dashed green line.

Note: The block to block connections should be made from the output port of one block to the input port of the next block. If these ports are not yet defined, a connection will be made from block to block, but it will not be attached to the desired output and input ports. So, define the blocks before connecting them together.

A job block or **WorkFlowJob** contains one activity block, any job interface information (inputs and outputs) and initialization values (if needed). Fig. 14.4 shows a workflow name `Deploy_and_Test` which brings in a LEC from

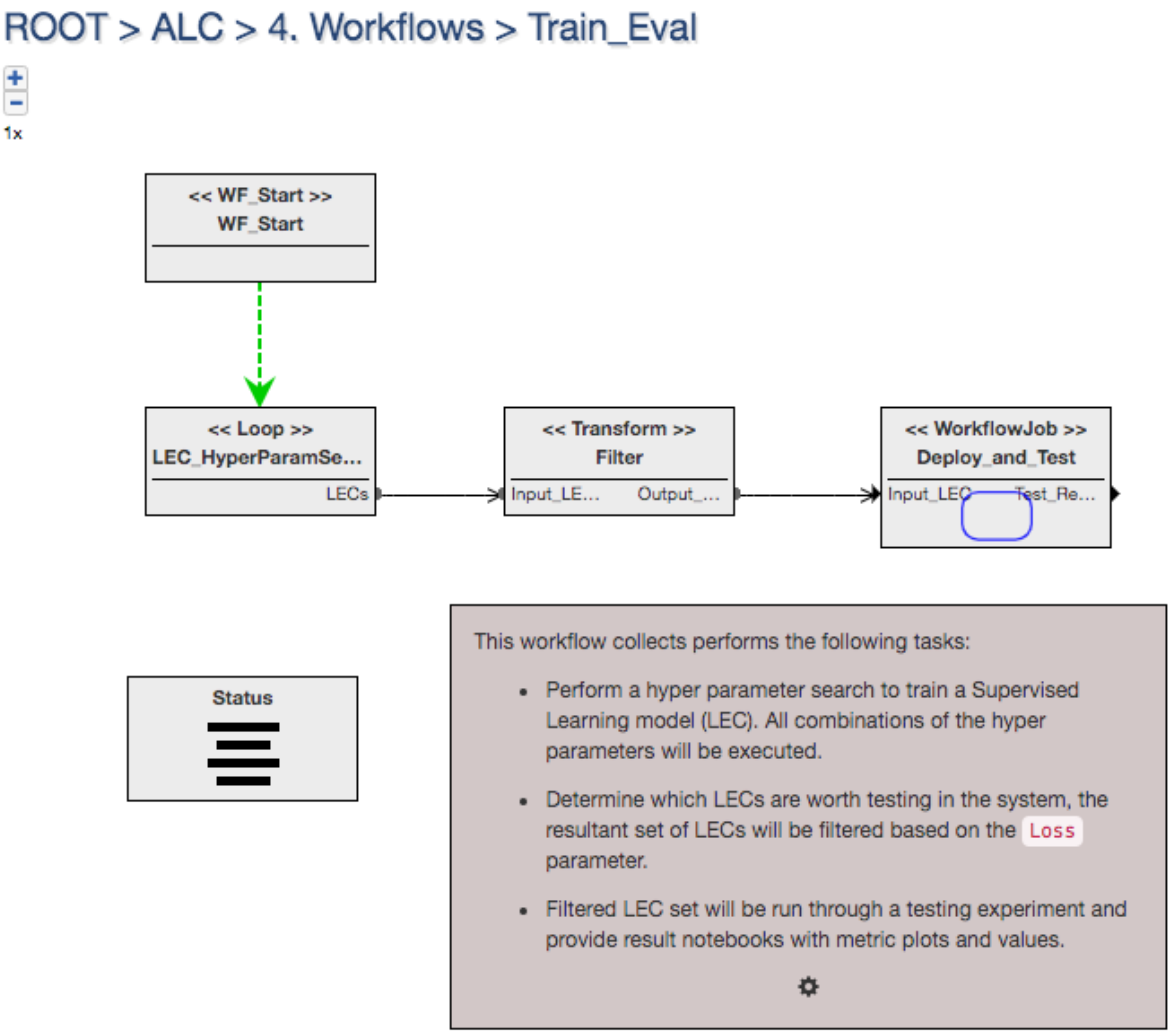


Fig. 14.3: Tutorial Workflow Example

a previous job, performs an activity and outputs the results.

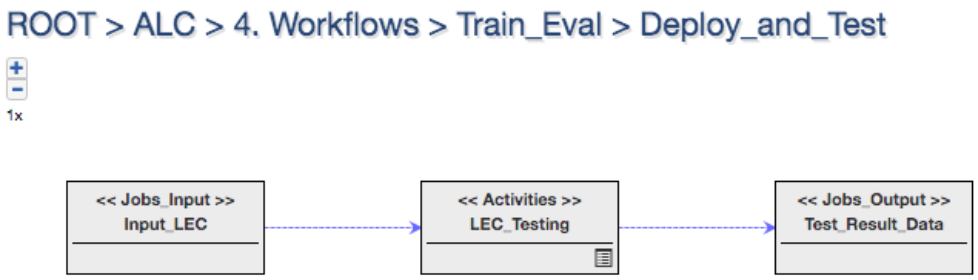


Fig. 14.4: Workflow Example

A job activity (**Activities**) can be any executable model such as experiment setups, LEC or assurance monitor (AMs) training, and performance evaluation of deployed LECs and AMs. Execution of an activity typically involves the use of domain-specific tools which operate on the input model and produce a result. The user will choose the activity using the selection tool available by pushing down into the activity block, shown in Fig. 14.5. All executable models in the user's project will be available for selection. Only one category of execution model should be selected for an activity, where categories are Data Collection, Testing and Training. For a data collection activity, one or more existing execution models can be selected.

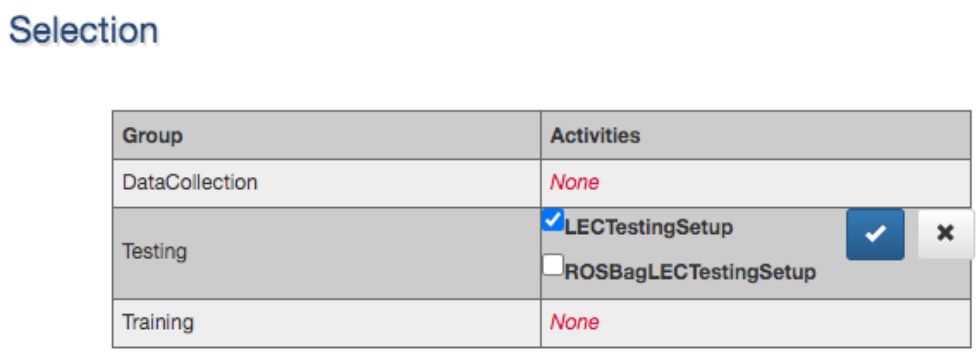


Fig. 14.5: Activity Selection Example

Note: In the near future, verification activities will be available in the workflow.

Each individual activity model usually needs to be initialized before execution, which can be done with an appropriately configured initializer block. Activity initialization is often dependent on the results of previous jobs in the workflow, which can be accessed through one or more input ports (**Jobs_Input**). The results of each activity contained within a job can be made available to later jobs through output ports (**Jobs_Output**). Some jobs in the workflow may be started using existing system information, such as uploaded datasets, pre-trained LECs or AMs and SysIDs defined in the verification model. This information is input into the activity by using the **Init_Value** block and can be the only input or combined with Jobs_Inputs from preceding jobs. The signal flow at this level is represented by dashed blue lines directed from an output port to an input port.

The **Jobs_Input** block allows configuration of how the input trained data or models from other jobs should be used in the activity for this job. The input types (**InpType**) attribute helps to indicate the form of the incoming data and **Set** shows what it will be utilized for in the activity. For example, LECs can be used in a training activity as the parent (or base) LEC to be trained or it can be deployed in the assembly for testing. The table below shows the possible combinations of input types and set values. If the input type is LEC or AM, a target LEC component from

the assembly model (**TargetAssembly** in the attribute list) should be specified. In the car tutorial, the LEC component was called `LEC_Control`). The **Init_Value** block has the same set of attributes as the **Jobs_Input**.

InpType	Set
data	TrainingData, EvalData, ValidationData
LEC	Assembly_LEC, Training_LEC
AM	Assembly_LEC, Training_LEC
SysID	PlantModel

The **Jobs_Output** block will specify the type of data generated or any trained models created by the job's activity. Like the input types for the Jobs_Input block, the possible options are data, LEC, AM or SysID.

14.2 Workflow Execution

Workflows can be as simple as a sequence of workflow jobs or can utilize advance work flow elements like *Loops*, *Transforms* and *Branching* (which are discussed below). Once a workflow is designed and ready, it can be executed by running the **Workflow Executor** plugin in the upper left under the play button, shown in Fig. 14.6.

Fig. 14.6: Workflow Executor Plugin

The workflow executor handles deployment, execution, and data management for workflow models. Parallelism is supported; for example, multiple parameters in hyperparameter search can be run jobs simultaneously, based on hardware resources that are available. Once a workflow execution begins, the executor tracks the progress of each job in the workflow and provides this information back to the user with a status table (available in the **Status** visualizer at the workflow model level and shown in Fig. 14.7). As each **WorkflowJob** starts, an entry will be created and will turn green when completed. Since the second workflow job input depends on the output of the previous workflow job, it is not provided in the status until it is started. If a workflow job encounters an error during execution, the status table will be updated to reflect this. Once the error has been identified and fixed, then the workflow can be resumed from the point where the failure occurred, i.e. none of the preceding jobs which finished successfully before the error will be executed. As with the model results blocks, a link is provided to the resulting Jupyter notebook and a log of information about the run.

Results of each run can be seen by clicking the `Notebook` link next to each result to pull up the Jupyter notebook associated with that job result. The `Log` link will provide the logs from the job run and will help to identify errors that may happen during a particular run. To see information about the workflow execution, refer to the `Workflow` line and see the `Results` column where there is a `Log` and `Error` link. The `Script` link shows how the workflow is utilized and where any user defined scripts are inserted. User errors in the scripts will appear in the `Error` information and it will reference the line numbers in the `Script /path/to/file`.

1. Initial Status

WF_example

Job	Activity	Status	Results
Workflow		Started	Script Log Error

2. First Job Partially Complete

WF_example

Job	Activity	Status	Results
Workflow		Started	Script Log Error
LEC_HyperParamSearch, SLModelTraining iteration: 2, -	LECTrainRegression	Completed	Data Log Notebook
LEC_HyperParamSearch, SLModelTraining iteration: 1, -	LECTrainRegression	Completed	Data Log Notebook
LEC_HyperParamSearch, SLModelTraining iteration: 3, -	LECTrainRegression	Started	
LEC_HyperParamSearch, SLModelTraining iteration: 0, -	LECTrainRegression	Started	

3. First Job Complete, Second Started

WF_example

Job	Activity	Status	Results
Workflow		Started	Script Log Error
LEC_HyperParamSearch, SLModelTraining iteration: 2, -	LECTrainRegression	Completed	Data Log Notebook
LEC_HyperParamSearch, SLModelTraining iteration: 1, -	LECTrainRegression	Completed	Data Log Notebook
LEC_HyperParamSearch, SLModelTraining iteration: 3, -	LECTrainRegression	Completed	Data Log Notebook
LEC_HyperParamSearch, SLModelTraining iteration: 0, -	LECTrainRegression	Completed	Data Log Notebook
Deploy_and_Test iteration: -	LECTestingSetup	Started	

4. All Jobs Complete

WF_example

Job	Activity	Status	Results
Workflow		Completed	Script Log Error
LEC_HyperParamSearch, SLModelTraining iteration: 2, -	LECTrainRegression	Completed	Data Log Notebook
LEC_HyperParamSearch, SLModelTraining iteration: 1, -	LECTrainRegression	Completed	Data Log Notebook
LEC_HyperParamSearch, SLModelTraining iteration: 3, -	LECTrainRegression	Completed	Data Log Notebook
LEC_HyperParamSearch, SLModelTraining iteration: 0, -	LECTrainRegression	Completed	Data Log Notebook
Deploy_and_Test iteration: -	LECTestingSetup	Completed	Data Log Notebook

Fig. 14.7: Workflow Status Example

Individual results can be seen for each job run by returning to the model where the activities are defined and looking at the result block or selecting `Data` link under the "Results" column on the workflow status view (shown in Fig. 14.7). For example in the Fig. 14.4, the `LEC_Testing` activity will be setup to use the `LECTestingSetup` (see Fig. 14.5). If you go to that activity in the model and look at the results block, you will see the following runs in Fig. 14.8 and can view the same results information. The same information will be available in the **DataSet** information for easy referencing in the future.

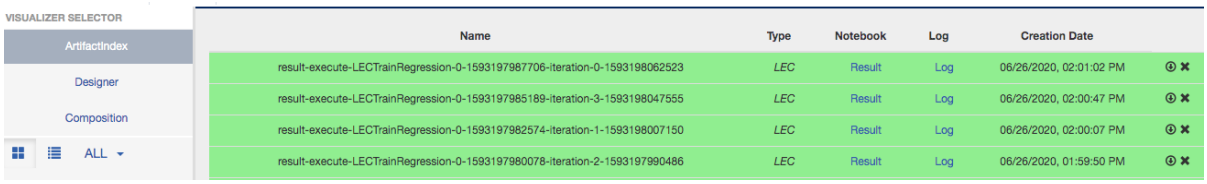


Fig. 14.8: Workflow Status Example

14.3 Workflow API for Scripts

For the advanced workflow elements, the developer can create scripts to control looping conditions, change parameter values that may be used by jobs within loops, filter results to select subsets of the results to move forward, and logic to select the next job. The scripts are written using Python 3.6. Examples of scripts will be provided in each of the appropriate sections below.

This section will outline the API available to access the job setup and result information. Each of the completed work flow jobs will have an associated JSON file that provides experiment setup parameters (`exptParams`) and result information (`results`). An example JSON result file can be found below:


```
{
    "info": {
        "description": "No Upload",
        "directory": "/alc_workspace/jupyter/alc_wk_models/LEC-2/1590712494538/...",
        "exptParams": {
            "BATCH_SIZE": 4,
            "DATASET_NAME": "rosbag_hdf5",
            "EPOCHS": 2,
            "LOSS": "mse",
            "METRICS": [
                "accuracy"
            ],
            "OPTIMIZER": "adam",
            "TRAINING_DATA_FRACTION": 0.85,
            "USEFUL_DATA_FRACTION": 0.2,
            "assurance_monitor_type": "SVDD",
            "epsilon": 0.75,
            "lec_assurance_monitor": false,
            "ml_library": "pytorch-semseg",
            "path_prefix": "MODEL_LEC-2",
            "runSLTrainingSetup": 1,
            "upload": false,
            "use_vae": true,
            "window_size": 5
        },
        "hash": "dd6648c8d247435b023528c35de666fa150f3536",
        "result_url": "alc_wk_models/LEC-2/1590712494538/...",
        "results": {
```


(continues on next page)

(continued from previous page)


```
            "model_evaluation": {
                "0": 0.9545281438463257,
                "1": 0.0076326002587322125,
                "2": 0.0037994327607427623,
                "3": 0.0,
                "FreqW Acc : \t": 0.9533210134575314,
                "Mean Acc : \t": 0.3196469889233613,
                "Mean IoU : \t": 0.24149004421645018,
                "Overall Acc: \t": 0.9540669642857142,
                "loss": 1.0003615220387776
            },
            "training_history": {
                "0": [
                    0.9165251573906917,
                    0.9545281438463257
                ],
                "1": [
                    0.004606702752504895,
                    0.0076326002587322125
                ],
                "2": [
                    0.0006646971935007386,
                    0.0037994327607427623
                ],
                "3": [
                    2.3353936695262263e-05,
                    0.0
                ],
                "FreqW Acc : \t": [
                    0.9153626547510644,
                    0.9533210134575314
                ],
                "Mean Acc : \t": [
                    0.24787725317771886,
                    0.3196469889233613
                ],
                "Mean IoU : \t": [
                    0.23045497781834814,
                    0.24149004421645018
                ],
                "Overall Acc: \t": [
                    0.9162483258928571,
                    0.9540669642857142
                ],
                "loss": [
                    1.026919510629442,
                    1.0003615220387776
                ]
            }
        },
        "upload_prefix": null
    },
    "name": "result-execute-LEC-2-0-1590712492524-1590712494538",
    "path": "/y/V/z/Y/7/4",
    "time": 1590712494538
}
```

To access an element of a job result, use `result.<key-name>` for the element you want to reach. For instance to get the list of loss values over the training of the LEC whose results are shown above, use `result.info.results.training_history.loss`. There are also shortcuts for results that are expected to be commonly used, these are shown in the table below along with the full command to allow referencing back to the original result JSON location.

Result Information	Full Command	Shortcut
Experiment Parameters	result.info.exptParams	result.parameters
Model Loss Value	result.info.results.model_evaluation.loss	result.loss

14.4 Transforms

Workflow supports **Transform** blocks to filter or join results produced in one or more jobs and feed it to the next job. An example could be to create a script that selects the best LEC for the situation and prune away the other LECs. Fig. 14.9 shows the transform block named `Filter` (on the left side) and the definition of this block (on the right side). The transformation type (**Transform** attribute) can be a filter, join or custom (note: custom is not currently utilized and will be available in the future).

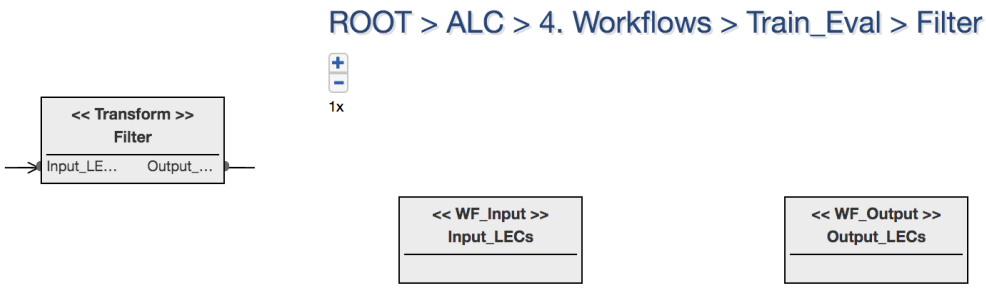


Fig. 14.9: Transform Example

Inside the transform block, the user defines the number of **WF_Input** ports desired to provide a path for the previous job's data to be passed in and places a single **WF_Output** block to feed data to the next job in the workflow. The transform block has a script attributes (located in the bottom right property editor) to allow the user to provide the code to perform this transformation. The WF_Input and WF_Output block names provide the expected variable names to use within the script. The script code can access the results from a previous job to utilize in the user generated logic as described in the *Workflow API for Scripts* section. Below is the script used in the car tutorial to take a set of LEC models (`Input_LECs`) and select the one with the lowest loss metric value for output to the next job (`Output_LEC`). This script should always return a list for the `Output_LEC`, shown by `return [ret_lec]` below.

```
def fun1(Input_LECs):
  if (len(Input_LECs)<=1):
      return Input_LECs

  loss_val = 100000
  ret_lec = None
  for lec in Input_LECs:
      lec_loss = lec.loss
      if (lec_loss < loss_val):
          loss_val = lec_loss
          ret_lec =  lec

  return [ret_lec]
```

The example above passes the input LEC list forward if only one LEC exists. Another option would have been to make

sure that an input exists, and if it does not throw an exception (shown below). The user will be provided feedback from the script execution.

14.5 Branching

For situations where a system parameter or job result can indicate a desired execution path, the **Branch** block can be used to allow the user to define a decision criteria for a True or False result using a Python3 script. Fig. 14.10 shows an example where the resultant LEC from the `Filter_LECs` block could be either the result of a trained model using an uploaded dataset (jpeg files) or a model trained using a ROS bag recording file from a system data collection run. Each of these resultant LECs utilize different formatted data for predictions (either jpeg files or ROS bag files). Therefore, the system testing workflow job to evaluate the LEC is different (`Deploy_Test` for jpeg files and `ROSBag_Deploy_Test` for ROS bag files). In this example, the resultant LEC will be fed to both of the system test options, but only one of the system tests will be executed based on whether the **Branch** block script indicates a `True` or `False` answer. The branch block will show the data flow path for the True result in a solid blue line, while the False path is in a solid red line.

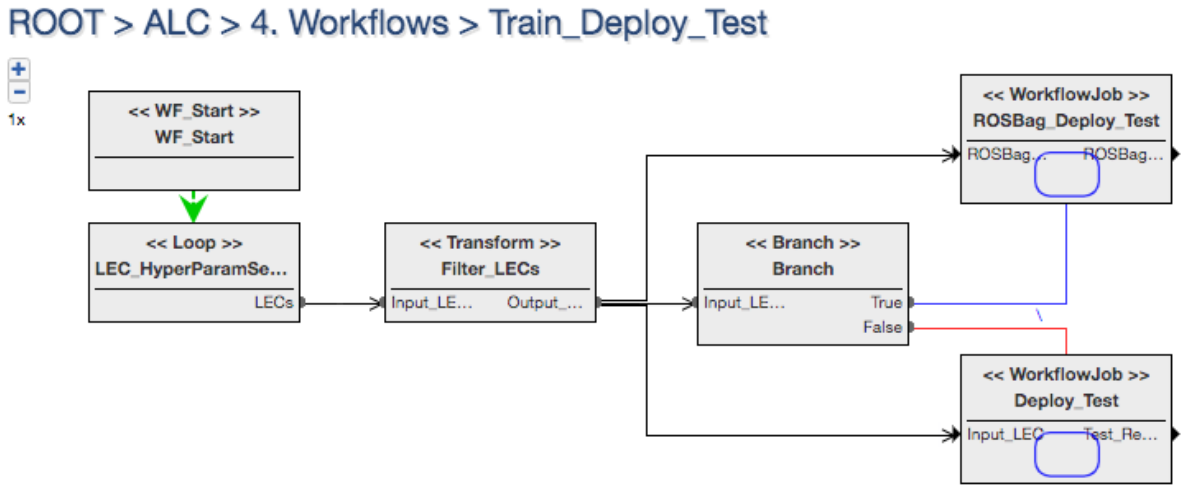


Fig. 14.10: Branching Example

The `script` attribute (bottom right) provides an `Edit content ...` to let the user create the needed code for the script. Inside the branch block, there should be a **WF_Input**, **True** and **False** blocks to allow identification of the variables available to the script as shown in Fig. 14.11.

Below is an example script that will take the input to the branch block (the **WF_Input** block named `Input_LEC_b`), check the LEC training setup parameter `DATASET_NAME` to determine if it used an uploaded dataset of jpeg images (`CUSTOM`) or ROS bag files (`rosbag_hdf5`). For the later, the branch result will be `True` and the `ROSBag_Deploy_Test` workflow job will be run. Otherwise, the branch will be `False` and the `Deploy_Test` workflow job will be run.

```
def my_branch(Input_LEC_b):

    if len(Input_LEC_b) == 0:
        raise Exception("No lec provided to \"my_branch\" function")

    if Input_LEC_b.parameters.DATASET_NAME == "rosbag_hdf5":
        return True
    else:
        return False
```

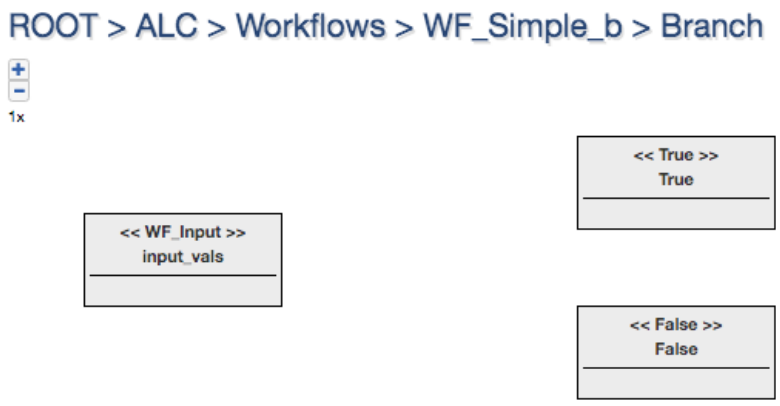


Fig. 14.11: Branching Block Example

14.6 Loops

Loop blocks can be used to allow repetitive execution over a contained sequence of jobs. For example, loops are useful in performing hyperparameter searches on a LEC training model as shown in Fig. 14.12. Loops can iterate over a single or set of WorkFlowJobs. Loops may also be placed within a loop. `For`, `While` and `Do...While` looping mechanisms are available in the **Loop_Type** attribute. The **FOR** looping mechanism will iterate over all combinations of the loop variables defined in the model. This mechanism supports both parallel and sequential execution. The **WHILE** and **DO...WHILE** looping supports sequential execution and provides termination of loops based on user defined conditions in a script found in the property editor's attributes. More will be provided about **WHILE** and **DO...WHILE** loops in *While Looping*.

Note: All workflow scripts are defined using Python3.

The **WF_Start** block indicates the start of the loop. As in the work flow, the connection with the first job is represented by a dashed green line. The single activity (job) or set of jobs for the loop should be defined first to configure the input and output ports for each job. One or more loop variables can be defined by adding them as **Loop_Var** blocks within the loop block definition. These variables will be directed into the appropriate WorkFlowJob block. If the loop is following another WorkflowJob, a **WF_Input** will provide a path for the previous job's data to be passed to the loop's WorkFlowJob. If the loop is the first element of the work flow, an **Init_Value** block can be used in the **WorkflowJob** block that contains the first activity. The loop output port is represented by a **WF_Output** block.

Loop variables can be one of the parameters setup in the execution model for a job, a local named variable or it could be a list of LECs or AMs or SysIDs from previous jobs or loops in the workflow, as identified in the variable type attribute (**VarType**). For parameters, the **Values** attribute provides a list of values or range that will be iterated on within the `For` loop. Specific values can be provided in a bracketed comma separated list, i.e. `[2,4,6,8,10]`. Ranges can be defined using Python 3 `range` for lists functions, i.e. `list(range(1,10,2))`. A list of string values must include double quotes around the strings, i.e. `["Hello","World"]`. All possible combinations of loop variable values will be used in the iteration process by using a Cartesian product to traverse the loop variable. For `While` and `Do...While` loops, the loop block **script** attribute is used to check if additional iterations of the workflow are desired and update the activity models as needed for the next iteration. This script should be developed by the user using Python3 code. All loop variables are accessible to this script.

To utilized LECs, AMs or SysIDs from a previous job as loop variables, the **Loop_Var** block needs to be connected to an input port that brings the information into the loop block. For example, if the previous job is a filter that supplies multiple results and these result are then fed into a **WF_Input** block within a loop block. The workflow executor will then iterate over each of these results, as it would a parameter.

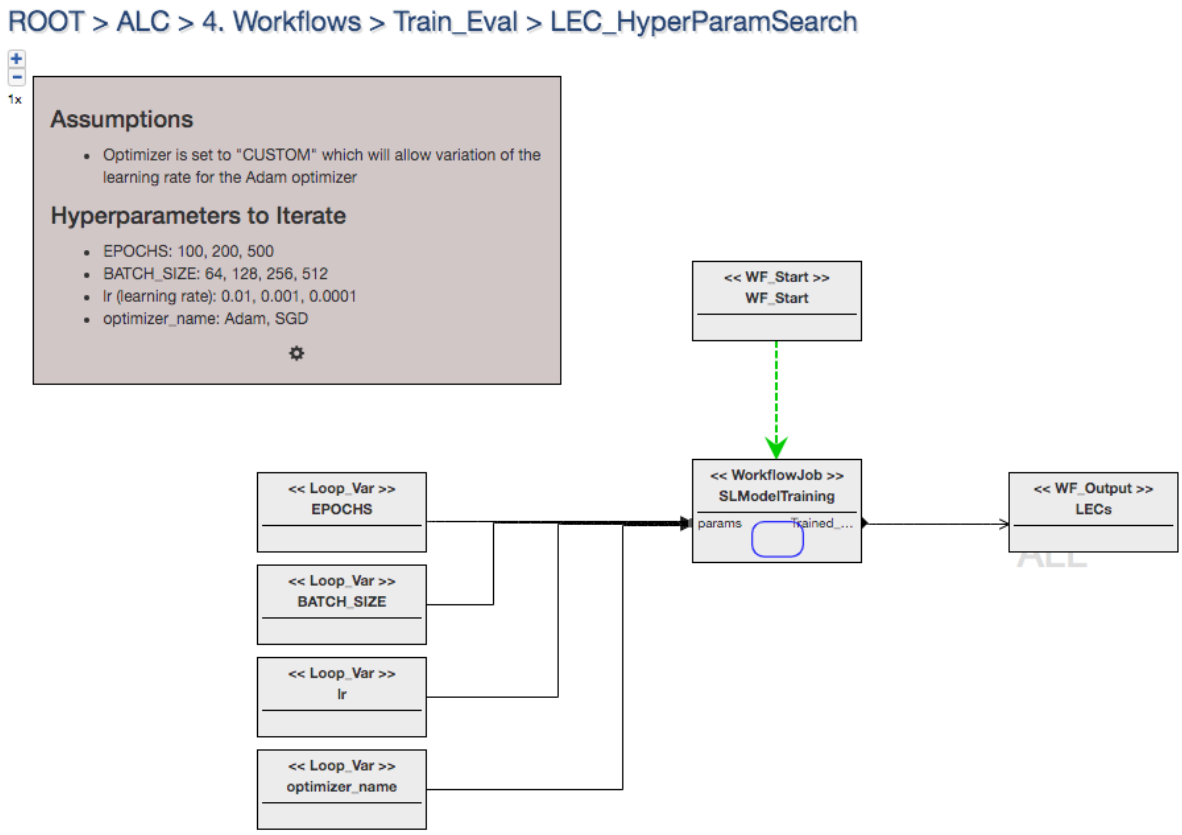


Fig. 14.12: Looping Example

Jobs which utilize the loop variables must include a **WF_Property** block to accept the loop variables into the job, as shown in Fig. 14.13. The loop variable value is passed by the Workflow Executor during the looping process to the job's WF_Property block. This block handles providing the information to the job's activity. The only configuration needed for the WF_Property block is to name it for reference as an input port to the job activity. Once the block is available, it will be reflected as an input to the job at the loop definition level. Then the **Loop_Var** blocks can be connected to this block input using the data flow lines.

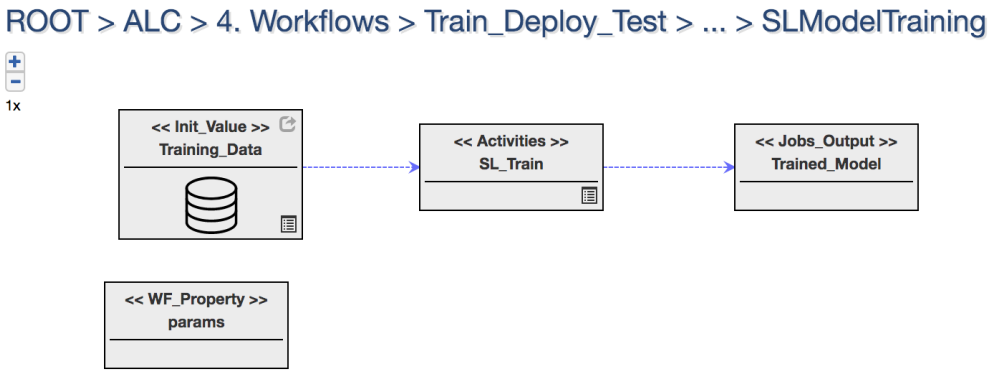


Fig. 14.13: Job Utilizing Loop Variables

14.6.1 While Looping

A while-loop in the workflow-engine works similarly to a while-loop in any programming language, it iterates through the provided script until the while-loop condition becomes false. This method could be used to define criteria for adaptive looping or early stop methods. Select `Edit content ...` from the script attribute (on the bottom right) to bring up a full screen editor that can be used to create the desired script. There are two types of user-written functions that can be associated with while loops:

1) Implements the loop's condition
2) Controls the values of the parameters for jobs within the loop body. These parameters may, for example, be the hyperparameters for an LEC training job.

Unlike the scripts associated with filters and branches, the scripts associated with while-loops do not receive a list of results as their input parameter. Instead, the script receives a **WorkflowData Object**, called `workflow_data`, which gives complete access to all of the results generated by the jobs in the workflow as it executes the loop. Another thing to note is that transform and branching scripts are not considered jobs, just scripts run to direct the next workflow job activity (either input information or which jobs to run). So, this data is only the results of WorkflowJobs.

Let's take the workflow example shown in Fig. 14.3 to explore how to access the WorkflowData object information within a script. The loop is named `LEC_HyperParamSearch`. From Fig. 14.12, the WorkFlowJob that will be looped on is called `SLModelTraining`. The WorkflowData Object name for this example is `Train_Eval`. So, the path to the SLModelTraining job is `["LEC_HyperParamSearch","SLModelTraining"]`. Once the loop starts execution, each iteration of the loop creates a results (numbered sequentially starting from 0). To specify a particular SLModelTraining job, specify the iteration of the LEC2_HyperParamSearch job of which the SLModel-Training job of interest is part. This is done by specifying the iteration's index. Both of the following paths specify the SLModelTraining job of the last iteration of LEC2_HyperParamSearch that has 4 iteration passes. The –1 is the index of the last item of a collection in python, and so is equivalent to "3" in this case.

The `get_results` method is used to return a list of the requested results given a job path as indicate above. The code below shows how to retrieve the results from the SLModelTraining job of next-to-last iteration of the LEC_HyperParamSearch While-Loop. Note that, if there are no jobs that match the path, an empty list is returned.

Since the script is written in the loop block element (`LEC_HyperParamSearch`), the `get_results_relative` method can be used and will pull a list of results relative to the current loop block. An example is shown below where the current iteration results are retrieved. Given that the returned results are provided in as a list, the elements of the results can be retrieved as in the other scripts by adding an index of 0 (`[0]`).

```
current_results = workflow_data.get_results_relative(-1, ["SLModelTraining"])
current_loss = current_results[0].loss
```

Below is an example of a While-Loop condition function. This while loop will keep iterating until the current iteration's (-1) loss value is either below 0.1 or less than the previous iteration's (-2) loss value.

```
def adapt1(workflow_data):
    current_results = workflow_data.get_results_relative(-1, ["SLModelTraining"])

    if not current_results:
        return True

    current_loss = current_results[0].loss

    if current_loss < 0.1:
        return False

    previous_results = workflow_data.get_results_relative(-2, ["SLModelTraining"])

    if not previous_results:
        return True

    previous_loss = previous_results[0].loss

    return current_loss < previous_loss
```

To controls the values of the parameters for jobs within the loop body using the script, there is a **ParameterUpdates object** that can be retrieved using the code below. This object can be used to set the value for a particular job parameter, either system parameters or training/experiment parameters. These parameters does not have to be loop variable.

```
parameter_updates = ParameterUpdates()
parameter_updates.my_parameter = value
```

Here is an example of a While-Loop controlling the values of the parameters for jobs within the loop. The example script below sets the number of epochs for training in the current iteration based on the loss value of the previous iteration and returns the information for use in the next running job.

When setting up a WorkflowJob activity, it was possible to select multiple activities to run. An example in Fig. 14.14 shows where multiple `DataCollection` activities are chosen. To customize parameters for each of these activities, use the `for_activity` call shown in the code segment below.

Note: If the same parameter value is set both with and without the `for_activity` call, the value specified with `for_activity` overrides the value specified without this call.

Selection

Group	Activities
DataCollection	**GazeboCampaignExperiment** **GazeboExperiment**
Testing	*None*
Training	*None*

Fig. 14.14: Multiple Activities Selected for Job

Part II

Additional Information

CHAPTER

FIFTEEN

DATA SET VIEW

The DataSet view keeps track of the record for each execution of activities associated with data collection, training, testing, verification and validation. Each activity gets its own table and each row in the table corresponds to an execution of the activity. See *ALC Toolchain Overview* for more information.

Currently the data set view, captures the location of the activity, the parameters used during its execution, and links to the input data sets (for training or evaluation) and the input models (for re-training or testing). This allows the user to trace through the evolution of the execution records in each activity and the relationship between the records across all activities.

The view also provides links to the activity model and to a IPython Jupyter Notebook with results from a user configured post-processing script. Since the results produced during the exuection are in the same folder as the notebook, the users may write additional scripts in the notebook to further evaluate the execution results. The figure below shows a screenshot of the data set view with a table for each activity. The rows in each table correspond to the individual runs. The "Result" link may be clicked to open the Jupyter notebook for the corresponding run. The parameters used for each run are displayed when

The figure also shows the tracking of the artifacts ("Trained LEC") used in a run of *cp2_00*. The LEC/ Assurance Monitor used is highlighed in the activity LEC_DD_VAE.

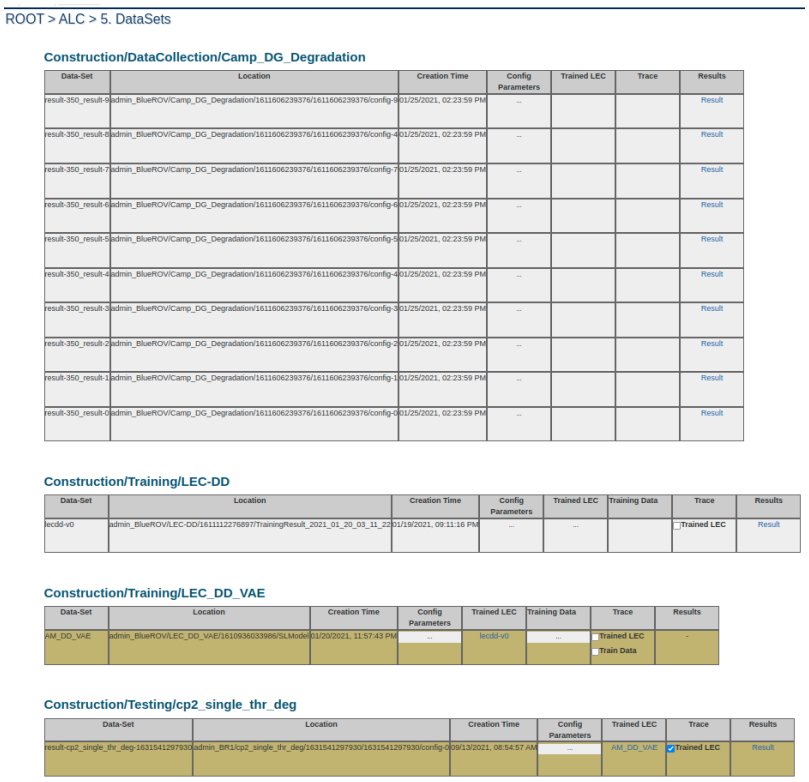


Fig. 15.1: Data Set View

CHAPTER

SIXTEEN

GIT AND REGISTRY SUPPORT

The ALC toolchain runs a local git repository server for the code-base associated with the projects. For instance in the BlueROV project, the "ALC" includes an attribute "repo" that is set to bluerov.

When the IDE is launched (refer to section *Working with IDE*), the *LaunchIDE* plugin clones the git repository and makes it available in the IDE.

ALC IDE

In the ALC IDE environment:

- Users can build, run, check the results and debug the code.
- Like any other local git copy, users can commit and push the code to the "remote" git repository.
- The users can use the git commands to tag the repository or create branches etc.
- The IDE environment is set up with the correct credentials and git config for the user.
- Environment variable *$GIT_SERVER_URL* provides the git server ip-address and port number.

ALC Webgme

In the ALC webgme environment:

- The updated code can be used using the "LaunchExpt" plugin.
- When the plugin is started, it allows the user to set a boolean flag - "Setup and Build Repo".
- In case the user has pushed any changes to the git repository in the IDE environment, the user can set the this flag to true and start the execution. The plugin will pull the updated code from the git repository and build it prior to execution.
- In case there are no changes to the repository code, the user can leave this option as false and the existing code is used to run the simulation.
- The top level "ALC" model in any project includes an attribute where the user can specify the branch and/or tag to use. The user can set this attribute and the LaunchExpt will try to checkout the appropriate branch or tag prior to execution. When these options are left empty (default), the code in the "master" branch is executed.

Docker registry

The ALC toolchain also run a local docker registry server. This is because the docker environment of the host machine is separated from the docker environment in the IDE.

Apart from updating the code in the IDE, the users can also update the docker image locally within the IDE space and test it out. If they want to update the docker image for future executions, they can push the docker image to the registry.

```
docker tag $image_name $REGISTRY_ADDR/$image_name
docker push $REGISTRY_ADDR/$image_name
```

The above command will ensure that the image ($image_name) is pushed to the local docker registry. The registry address is available in the environment variable *$REGISTRY_ADDR*. The user can update the top-level build.sh script to ensure that the updated docker is pulled in the host machine docker space when the LaunchExpt plugin is started with “Setup and Build Repo” option set to true.

CHAPTER

SEVENTEEN

EXPORTING ARTIFACTS

The “ExportArtfact” plugin exports as artifact the code in the repository as well as the trained artifacts (LEC and Assurance models) that are used in specific execution instances in the “Experiment Setup” models found in “ALC/ 2. Construction/ (Data Collection/ Testing)”

Upon execution the plugin creates:

- a tar file for the sources (if the repository is set up)
- a “TrainedModel_<ModelName>.tar.gz” file for each LEC/ Assurance monitor. The “ModelName” refers to the “Deployment Key” attribute for the LEC in the Assembly model.
- links to download the above files to the client machine. To view these links click on “Show Details” in the plugin results.

Once downloaded to the client machine, the artifacts can be transferred to the target board for setup and execution.

CHAPTER

EIGHTEEN

UPLOAD TO SERVER

This is a plugin available in the ALC toolset to import data file (training data, weights, etc.) that were generated outside of the ALC environment. This plugin should be run in the context of the activity to which the data belongs - Experiment Setup (for data collection, testing), SLTainingSetup (for supervised learning training), RLTrainingSetup (for reinforcement learning setup).

This plugin uploads data/ model folders to the ALC fileserver and creates a record of the data/ model in the context of the activity they were uploaded from. This allows the record to be referenced and used for future jobs/ activities run with the ALC tool set.

Note: The plugin name is "Import Data" and it is available in the "Experiment Setup" and "Training" models that are contained in "ALC/2. Construction/ (Data Collection/ Training/ Testing)"

CHAPTER

NINETEEN

MODELING ERRORS

It is common to make mistakes while modeling and writing the code associated with the LECs. The Execute Experiments plugin in the ALC toolset is used to launch the jobs for data collection, training and testing. It parses through the model and catches some of the errors such as missing training data sets, missing LEC links, missing model elements and errors in the parameter dictionary and reports them back to the user.

There are other errors which are embedded in the code and are reported by the execution backend in the server. The errors from the backend are reported in the log files that the plugin provides as *log* links in the ArtifactIndex view. If the error/ exception stack is reported to the plugin, it catches these errors reports the messages in the plugin results (show details link).

The ALC toolset uses a generic parameter dictionary to set the parameters in most contexts. There could be human errors associated with editing and setting these dictionaries. While there isn't a mechanism to report the exact error, the visual rendering of the parameter table fails when there is an error.

CHAPTER

TWENTY

PARAMETERS

Parameters are used in many models in the ALC toolchain. The `Param` element is used for this. This includes a `Param` view where the users add/ delete parameters from a table and edit the parameter name and value in each row.

Use in ALC Models

The `Param` element is used in various contexts,

Implementation Block

- In the Blocks that are marked as Implementation blocks ('IsImplementation' attribute is set to true), the Param elements are used to indicate any ros-launch file specific arguments and their value.

Environment Model - In the Environment model defined in Modeling/WorldModels and used in Construction activities (ExperimentSetup, RLTrainingSetup), the parameter dictionary is used to indicate any ros-launch file specific arguments and their value.

Execution Params

- Data Collection, Training and Testing activities under Construction provide a set of 'Execution' parameters. While some of these might be relevant to ros-launch files, they mostly are for controlling the simulation and their results.
- upload : Indicates if the result should be uploaded to the fileserver. (boolean)
- path_prefix : If upload is true, this specifies any prefix to the upload path in the file server directory structure.
- timeout : If this value is set, then the simulation will be killed by the execution runner when this time elapses. Since this parameter is used by some ros-launch files to terminate themselves, a buffer time is added by the execution runner before the dockers are stopped.
- unpause_time: time to wait before starting the simulation (this commonly used in ros-launch files to define a waiting period while all resources finish loading).
- record : boolean value to indicate if results need to be recorded.
- gui : set to false to run without gui.
- headless : set to true to run in headless mode.
- num_episodes : number of episodes to run (often used in RL-Training simulations)
- testing : used in RL Training Setup 1 to indicate testing mode, 0 to indicate training mode
- termination_topic : a topic that execution runner can listen on and terminate the dockers once a message is received. Allows for more flexible termination timing/triggering than can be achieved with a simple timeout.

Params used to specify Campaign

The `Campaign` element uses the same syntax as the `Param` element. However, in this case it is possible that some of the keys have a list of values over which the activity needs to be performed. So entries with more than one value are set as a list or an array in the JSON dictionary.

Example :

- {"pipe_roll":[0.0, 3.14159]}

Params in Supervised Learning

Parameters specified in supervised learning (SL) activity include

- BATCH_SIZE : size of each batch
- EPOCHS - number of epochs for training
- USEFUL_FRACTION : fraction of the data to be used
- TRAIN_FRACTION : fraction of the used data to be considered for training
- LOSS - keras parameter to set the loss type
- OPTIMIZER - keras parameter to set the optimizer
- lec_assurance_monitor - boolean value to indicate if the assurance monitor should be trained

Params in Assurance Monitor Training

These parameters are specified for training assurance monitors. They are specified in AssuranceMonitorSetup and SLTrainingSetup.

- type : currently set to "Assurance_KNN",
- sub-type : includes "Classification" or "Regression"
- num_neighbors : number of neigbours in the k-nearest neighbours classification defaults to 5)
- test_fraction : fraction of the data to be used for testing (not training), defaults to 0.1)
- random_seed : random value seed (defaults to 42),
- confidence_values : a list of values of confidence bound ranges to be predicted by the trained assurance monitor, defaults to [0.9, 0.95, 0.99]

CHAPTER

TWENTYONE

SETUP WITH MATLAB

1. Install MATLAB (>=R2019a) on the host machine. (MATLAB does not support installation within docker).
   - Toolboxes needed:
     - Control Systems
     - Optimization
     - Parallel Processing
     - Deep Learning
     - System Identification
   - Additional Neural Network model optional examples:
     - VGG-16
     - VGG-19
     - To install these, open Matlab and type "vgg16". Click on the Addon-Explorer link and install. Repeat with the command "vgg19".

   Note: Matlab installation with the Deep Learning library can corrupt your nvidia driver installation. If Matlab has corrupted your nvidia driver installation, you will see errors with cuda when starting the ALC Toolchain services. To remedy this, uninstall and reinstall your nvidia drivers.

2. Follow the instructions in the *Standard Setup* to setup the environment variables, paths etc.
3. Follow instructions in ${ALC_HOME}/docker/alc/jupyter_matlab/readme.txt copied below:
   A. update *setup_env.sh* as per your setup
   B. change the username 'matlabuser' in the Dockerfile based on the name of the user who is registed to run matlab

      Note: If all users can run Matlab, then comment the lines (10-13,18) corresponding to 'matlabuser' in ${ALC_HOME}/docker/alc/jupyter_matlab/Dockerfile.

   C. update the path to MATLAB_SUPPORT_PACKAGE_ROOT based on where the matlab support packages are installed. This can be obtained by typing the following command in the matlab shell 'matlabshared.supportpkg.getSupportPackageRoot'
   D. run the following commands in order to build alc_jupyter_matlab

```
source ./setup_env.sh
./build.sh
./run_setup.sh
```

Note: "./run_setup.sh" is interactive. One of the matlab dependencies used in verivital requires a user confirmation.

At this stage you should have a complete alc_jupyter_matlab image

4. Run ALC Toolchain services

```
cd ${ALC_HOME}/docker/alc
./run_services.sh
```

Part III

BlueROV2

CHAPTER

TWENTYTWO

BLUEROV2 ACTIVITY DEFINITION

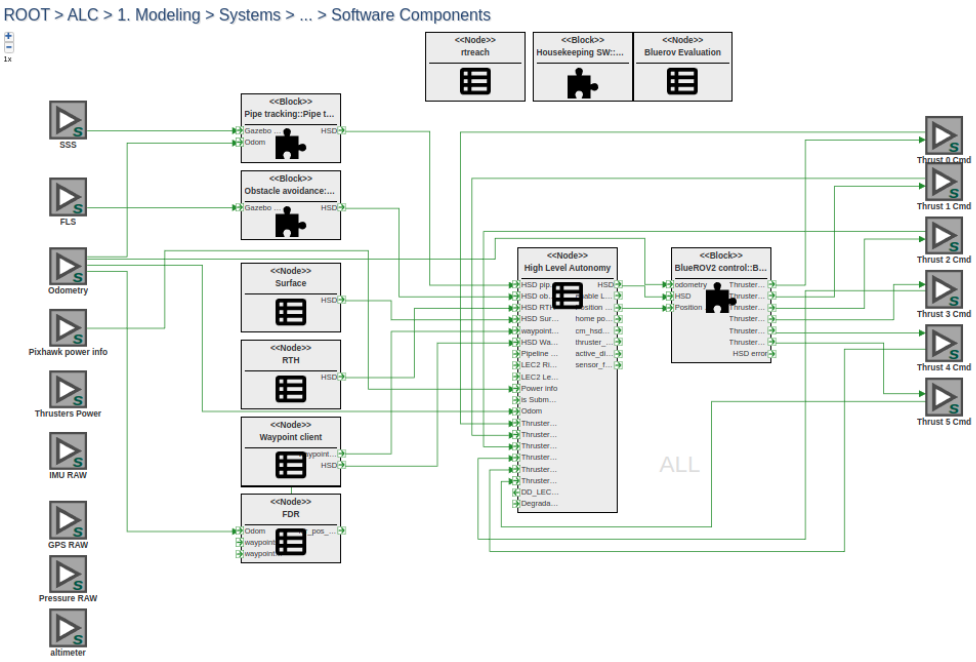


Fig. 22.1: System architecture of BlueROV2 UUV

22.1 BlueROVSim Activity

- To create a new instance of the BlueROVSim activity, open a BlueROVActivity model and switch to ALC/Construction/ and click on Testing or DataCollection model.
- Then Drag and drop the “Activity” model from the Part Browser on the left.
- Inside the “Activity” model, launch the “ActivityInit” plugin (Sometimes this plugin throws an error with a red-box message. Please ignore this and re-run this plugin.)
- Once this plugin is running, in the initial dialog box choose the “bluerov_sim” activity.
- In the subsequent dialog box, choose one or more options that can be configured as part of the BlueROV simulation. Options include
  - Waypoint: Setting up random waypoint generation.
  - Pipeline: Setting up pipeline generation for tracking pipeline.

  - Obstacle: Static and dynamic obstacle generation during simulation.
  - Degradation: Configuring the thruster degradation and FDIR
  - AIS: This is for setting up the AIS simulation. This feature will be supported in the next release which will include a planner as part of the BlueROV autonomy stack.
  - NoGoZone: Configure the no-go zone.
- Once the options are choosen and submitted, this generates an activity model for simulation.
- The user may change the defaut parameters associated with each of the options above and run the simulation by using the launch activity plugin (refer to headless video).
- When the user hovers over a parameter, its description is shown.
- A list of important parameters to tweak for the CP1, CP2, and CP4 scenarios is presented below.

22.2 Common parameters across all scenarios

- **The parameters in "Execution" table are common across all scenarios. The user may** tweak the following parameters
  - timeout (simulation execution time)
  - random_val (random seed for simulation)

22.3 CP1_00 (Pipe tracking)

This is a general scenario involves pipe tracking and obstacle avoidance. The UUV attempts to avoid static and dynamic obstacles while tracking the pipe. The user may tweak the following tables related to CP1

- Pipe : For pipeline generation details
- Obstacles: For static and dynamic obstacles. By default only static obstacle generation is enabled.

22.3.1 CP1_01_geofence_rth

This simulation presents a pipe tracking scenario with Geofence failsafe and Return To Home (RTH).

When the UUV reaches maximum distance from Home Position (Geofence = 50m), the UUV activates Geofence failsafe and command Return To Home (RTH). When UUV reaches home position, it emerges to the surface in a helix.

Parameters of interest include:

- Autonomy/ batt_charge [0.0 to 1.0]
- Autonomy/ geofence_threshold
- Autonomy/ failsafe_rth_enable [true/false]
- Obstacles/ enable_obstacles [true/false]

22.3.2 CP1_02_obstacles

This simulation presents a pipe tracking scenario, with random obstacles.

Parameters of interest include:

- Autonomy/ batt_charge [0.0 to 1.0]
- Autonomy/ failsafe_rth_enable [true/false]
- Obstacles/ enable_obstacles [true/false]

22.3.3 CP1_03_batt_low

This simulation presents a pipe tracking scenario, with low (20%) battery level at the beginning, without Return to Home function. This means UUV is executing the mission as long as it can, before battery gets critically low (15% for this example).

When the battery level reaches critically low threshold, the UUV is commanded to surface. The UUV emerges to the surface in a helix.

Parameters of interest include:

- Autonomy/ batt_charge [0.0 to 1.0]
- Autonomy/ failsafe_rth_enable [true/false]
- Obstacles/ enable_obstacles [true/false]
- Autonomy/failsafe_battery_low_threshold [0.1 to 0.2]

22.3.4 CP2 (Thruster degradation)

This simulation presents a pipe tracking scenario, with thruster degradation with or without static obstacles. By default it is #1 thruster, efficiency drops to 79% at t=50s.

Fault Detection, Isolation, Reallocation (FDIR - DD_LEC) system is active, detects degradation under 1-5 seconds. After detection FDIR reallocates thrusters to balance torque loss and maintain control for pipe tracking mission.

Parameters of interest include:

- Degradation/ thruster_motor_failure [true/false]
- Degradation/ enable_fault_detection [true/false] (this controls reallocation - detection itself is actice all the time)
- thruster_id [0 to 5] : ID of the degraded thruster
  - thruster_thrust_force_efficiency [0.0 to 1.0]
  - thurster_motor_fail_starting_time [1 to end of simulation]
  - enable_obstacles [true/false]

22.4 CP4 (Waypoint Following)

CP4 includes scenarios with waypoint following and obstacle avoidance.

22.4.1 CP4_00

This simulation presents a waypoint following scenario. Waypoints are defined in the mission file:

- mission_file [mission_04.yaml]

22.4.2 CP4_01 (obstacle on waypoint)

This simulation presents a waypoint following scenario, where a static obstacle is detected close to waypoint #3. UUV alters the waypoint - moves it forward towards the next waypoint. When it is closer to a given threshold to the next waypoint, UUV skips the already altered one.

The Mission/ mission_file parameter controls the mission file used in the simulation. This file includes the waypoints for the UUV to reach.

- enable_obstacles [true/false]

22.5 Real-Time Reachability

The RTReach node is always running in the BlueROV simulation. It warns the behavior true if the commanded heading leads to an unsafe situation for the uuv (collision with obstacle or nogo zone). When RTReach reports UNSAFE condition based on the obstacles, the UUV is commanded to “emergency stop”. Thereafter, the UUV moves straight up to the surface.

- use_rtreach [true/false]

CHAPTER

TWENTYTHREE

EVALUATION & METRICS

The *bluerov_eval.py* node does some basic realtime (*1Hz* update) evaluation, saved as *results/bluerov_evaluation.txt*. *(note: pipeline related nodes are still running in other, eg. waypoint missions also)*

```
[bluerov_eval.py]

Evaluation:
=========
* Simulation time total:    : 151 [sec]
------ [ Pipe tracking metrics] ------
* Pipeline detection ratio  : 1
* Average pipeline distace  : 22.2451621597 [meter]
* Tracking error ratio      : 0.0219278140944
* Semseg bad, not used      : 3 [sec]
* Pipeline not in view      : 2 [sec]
* LEC2 AM not triggered     : 0 [sec]
-------- [ Waypoint metrics] --------
* Average cross track error : 0 [m]
* Time to complete          : -1 [sec]
-------- [ Degradation GT info ] --------
* Degradation starting time : -1 [sec]
* Degraded thruster id      : -1
* Degraded efficiency       : -1
-------- [ FDI LEC info ] --------
* Thruster Reallocation     : 53.67 [sec]
* FDI Degraded thruster id  : 1.0
* FDI Degraded efficiency   : 75.0
```

CHAPTER

TWENTYFOUR

BEHAVIOUR TREE BASED CONTINGENCY MANAGER

Behavior Tree (update rate: `1 Hz`)

This node uses py_trees_ros v0.5.14 package for ROS Kinetic (`https://github.com/splintered-reality/py_trees_ros`, `http://docs.ros.org/en/kinetic/api/py_trees_ros/html/about.html`) to implement higher level autonomy for the BlueROV2. The package is included in the `bluerov_sim` docker.

The autonomy is using the `bluerov_bt.launch` file with the generated files in `$ALC_HOME/bluerov2_standalone/catkin_ws/src/vandy_bluerov/behaviour_tree_gen/` folder. For more information about the generator, please check `alc/btree` repo.

In the logs there will be lines starting with `[BT] task_name`.

For detailed information about the BlueROV2 Behaviour Tree based autonomy and the fault detection topic please check: https://www.mdpi.com/1424-8220/21/18/6089

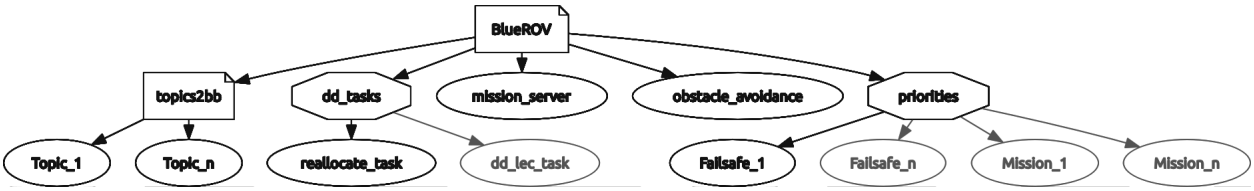


Fig. 1.: General layout of BlueROV BTree

For the autonomy manager check `4.2.1. Autonomy Manager` in the paper. The top level BlueROV root node has parallel children:

- topics2bb: Reads ROS topics, and update the input for the tree
- mission_server: Reads mission file, and controls the mission subtree
- FDIR: Fault detection (Degradation detector LEC + selective classfication AM + thruster control reallocation) subsystem
- Obstacle Avoidance: This node creates the final HSD command for the UUV based on the available HSD commands from mission and obstacle ROS nodes - always active
- Priorities
  - Failsafes: Battery low, RTH, Pipe lost, obstacle standoff, emergency brake, geofence etc.
  - Mission: Pipe Tracking, Waypoint following (FDR also), Loiter subtrees

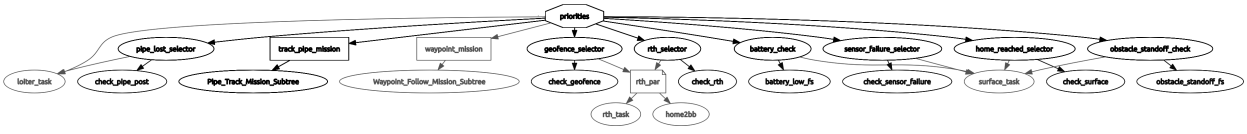


Fig. 2.: (Simplified) BlueROV BTree

Mission server:

In the BTree based mission server, when the simulation is started, the mission file is loaded from `vandy_bluerov/missions`. The parameter is controlled by `mission_file:=` definition in the launch script (launched from IDE) or environment variable (headless mode). This file can contain one or multiple missions (waypoint following with local, global waypoints or with headings; or pipe tracking)

Standard waypoint mission file is `mission_04.yaml`

Standard pipe tracking mission file is `mission_pipe_track.yaml`

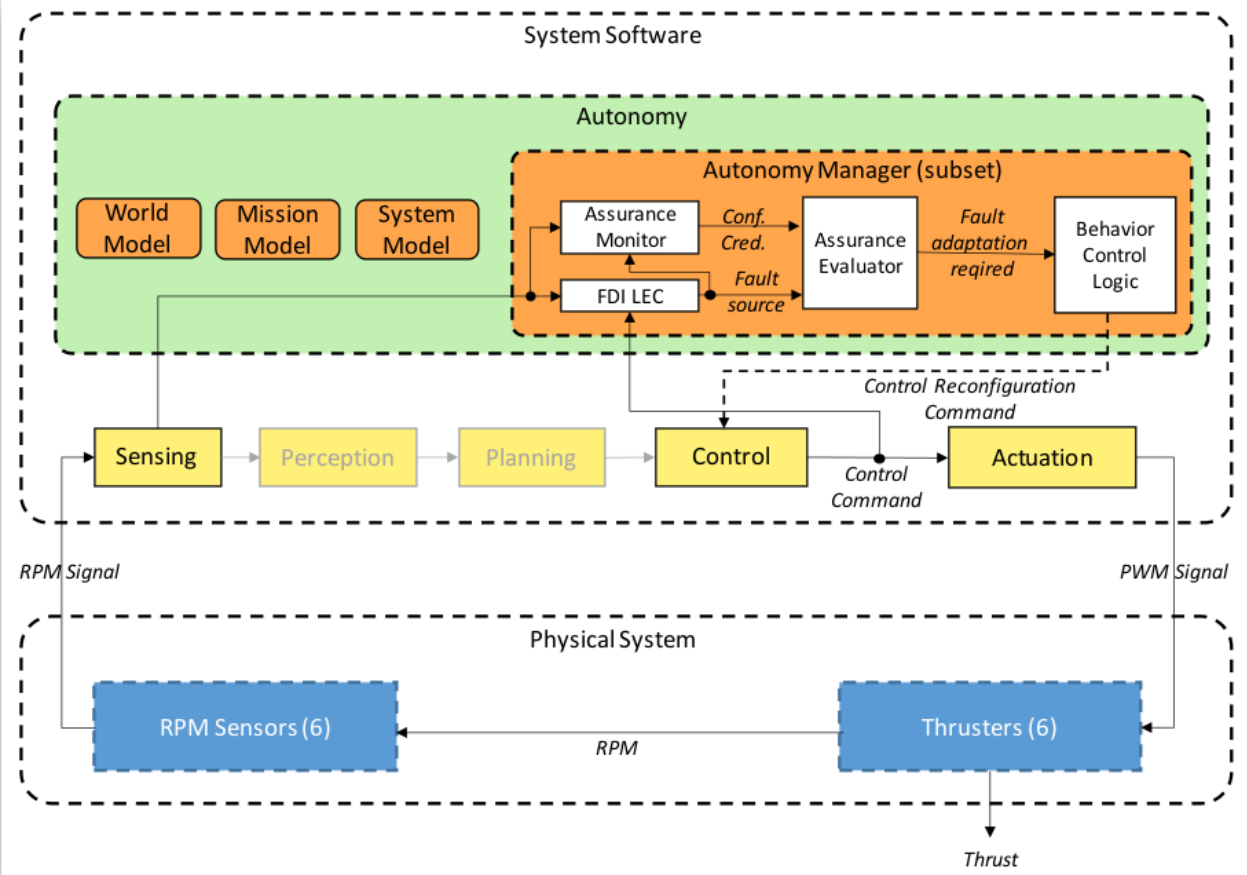
The FDIR system is using the DDLEC to detect thruster degradation and command thruster reallocation. It contains an LEC-based fault detection and isolation (FDI) module along with an AM. If the LEC with the AM detects a thruster failure and the Assurance Evaluator verifies this fault, then the information is passed to the BT control logic. Under autonomous operation, the FDI subsystem is always running. It receives thruster RPM signals from the sensors and control commands from the Control node. If the LEC and AM are indicating that the AUV is in nominal state, and the Assurance Evaluator confirms this output, there is no need for reconfiguration.

If the LEC detects a degradation (Fault source), the AM calculates the Credibility and Confidence metrics for that output. Using these values, the Assurance Evaluator marks the output as reliable with a Fault Adaptation Required signal, and fault adaptation is initiated. The LEC output class determines the degraded thruster (Actuator) and the approximate degradation level. Based on this information, the Control node can perform a control reallocation using a Thruster Allocation Matrix (TAM) which allows the AUV to continue the mission.

The degradation detector LEC + selective classification AM runs at `1 Hz` and checks the input data for degradation. The input data are:

- the 6 thruster commands,
- the 6 thruster RPMs (absolute value)
- heading command (change)

The `enable_fault_detection:=true` parameter controls the fault detection and control reallocation. Note, with the Behaviour Tree based autonomy, the fault detection is always running, only the reallocation part can be turned off with it.



LECs and AMs in BlueROV

In this release there are 3 LECs with AMs:

LEC2Lite with VAE AM:

- This is a segmentation LEC for the VU SSS images (pipe perception).
- AM is a VAE. LEC and AM were trained on SIM and HW collected data. (HW version is using a ground robot with an RpLidar as the scan source) When log(Martingale) value is high, pipe estimation is disabledIf the mission is pipe mapping, BT commands UUV for minimum speed.
- LEC is implemented in TF2.6.2 Lite, can run on CUDA GPU, CPU or Google Coral accelerator.
- Sources: catkin_ws/src/lec2lite/

LEC3Lite

- This is a signal processing NN. Input is the 252 bin sensor raw data (digitalized analog signal from sonar, with multiple contact echos)
- The lec3_lite_node.py will return with an array of contacts in [m]
- tflite inference takes around 2ms in average on x64 CPU (i7-9900)
- Trained with SIM and HW data (MaxSonar MB7070 using Analog Envelope output + RPi Pico)
- LEC is implemented in TF2.6.2 Lite, can run on CUDA GPU, CPU or Google Coral accelerator.
- Sources: catkin_ws/src/lec3lite/

FDIR

- Degradation Detection LEC is a fault detector combined with Selective Classification AM. More details can be found in the MDPI paper linked above.

CHAPTER

TWENTYFIVE

MAP-BASED PIPE TRACKING

Pipe mapping (update rate: `1 Hz`)

Pipe mapping function is always running, if LEC2Lite segmentation output is available - even in waypoint follow missions. (User can combine waypoint follow with pipe track missions in the mission file definition)

Map is a `1000x1000` array (Occupancy map). Resolution is `1m`, value `0` represents empty space, `50` is pipe. Update rate: `1Hz`. Map is fixed to World in TF tree.

Processing of Semseg messages:

Mapping description:

If there is a valid reading from perception LEC2Lite (`cm_use_lec2` enabled and message is not older than `1.5` sec:

- calculate pipe planar distance from UUV based on SSS beam angle parameters and UUV altitude from seabed
- calculate rotation matrix for pipe position using UUV position and rotation
- mark pipe estimated positon on Pipe map
- add pipe estimated position to `pipe_data` (max. `50` estimation) list
- estimate pipe heading using `pipe_data` with 1st order polynomial fitting. *Note: Last pipe heading information is available even when no pipe detected on LEC*
- publish output

Map based pipe tracking (update rate: `1 Hz`)

Tracking description:

If the mission type is set to "pipe tracking" this node will produce HSD commands for the BlueROV2 to follow the infrastucture, reconstructed using pipe mapping.

From `pose_gt_noisy_ned` reads UUV heading, from `pipeline_heading_from_mapping` reads the last pipe heading

Update rate `1Hz`:

- Get UUV and pipe heading difference as desired heading. *Note: Since this method is using pipe heading output from Mapping, so this works even when no Pipe is detected in LEC2 Semseg*
- Slight modification of heading cmd (when UUV and pipe heading close to parallel: differece < 5 deg.) to bring pipe back to center of scan. The `K_p_pipe_in_sls` controls the strenth of this behaviour.

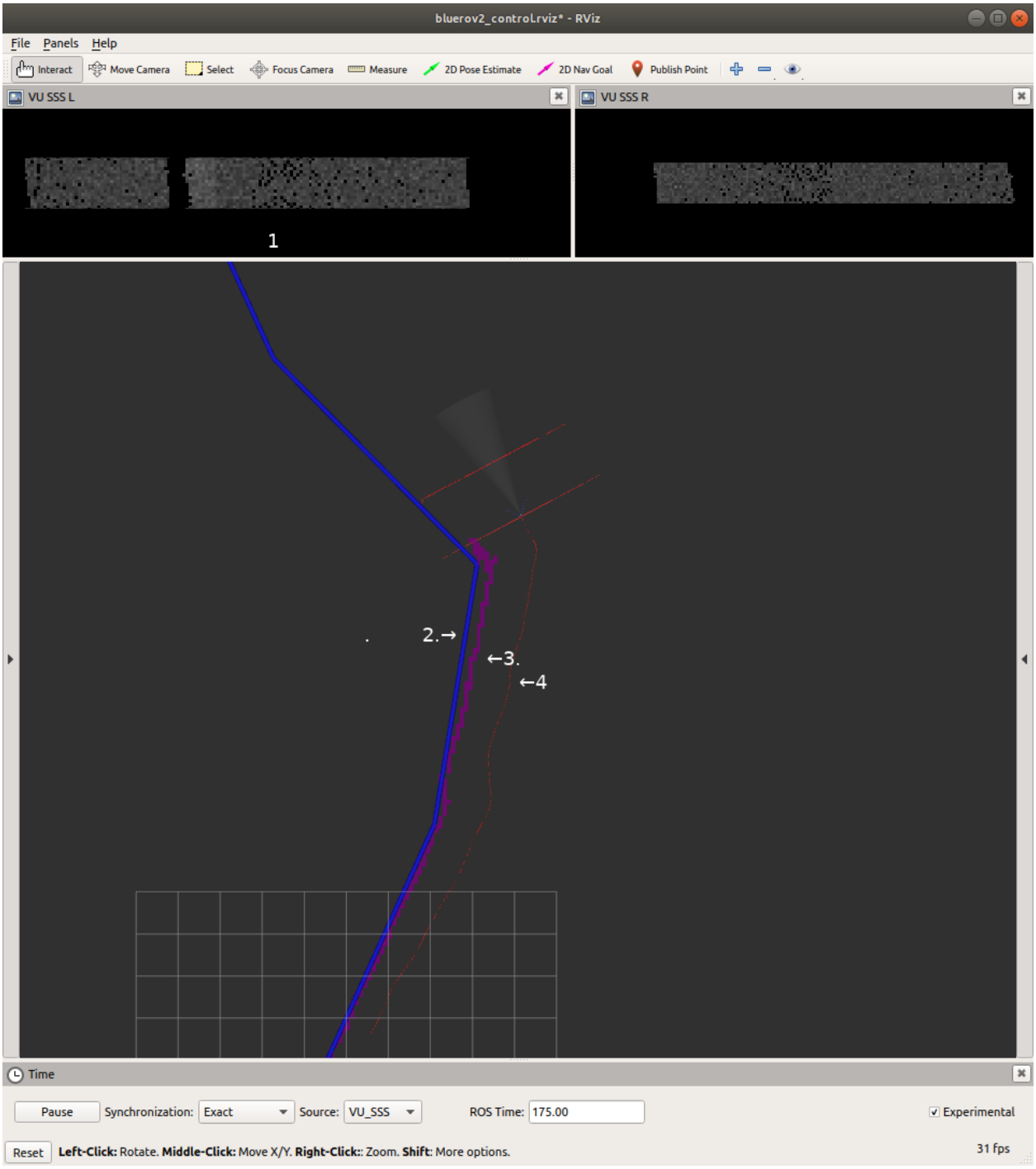


Fig. 1.: Bring pipe to center of scan in RVIZ

1. VU SSS raw output
2. Pipe estimation
3. Pipe
4. UUV Path

CHAPTER

TWENTYSIX

OBSTACLE AVOIDANCE

Obstacle mapping (update rate: `1 Hz`)

Inputs:

- BlueROV wide angle ultrasonic sonar (single gazebo distance reading)
- VU FLS based LEC3Lite perception
  - simulated multi-range FLS with 252 bins
  - constructed from 9 narrow beam ultrasonic gazebo sensor
  - LEC3Lite input: bins, output: array of ranges + bindary classification bins
  - waterfall image representation available in RViZ as a human 'readable' form - for both RAW FLS and LEC3Lite output)

Outputs:

- obstacle world map
- obstacle local map
- /uuv0/fls_echosunder

Update rate: `1Hz`. Map is fixed to World in TF tree

Static obstacles:

Processing of FLS messages:

The FLS echosounder (acoustic rangefinder) on the BlueROV2 has a 30 deg beamwidth with 30m range. The sensor gives only a range, so in the given distance the whole beamwidth is mapped as an obstacle. The surrounding 2m radius is represented as danger value. Valid ranges are between 0.5 to 30m. Other readings are neglected (as noise).

If the FLS detects no obstacles ahead, the previously detected obstacle values are decreased in every reading (until it disappear from map or appear again as obstacle).

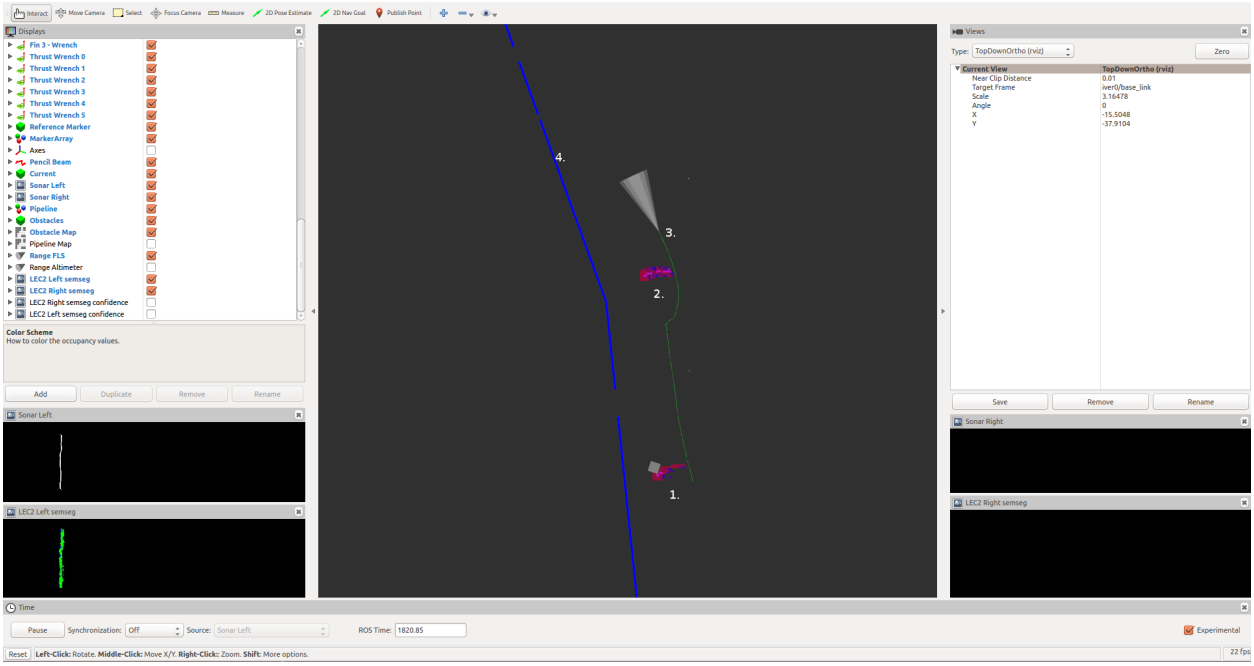


Fig. 1.: Pipe map in RVIZ

1. Large obstacle with obstacle representation on map
2. Small obstacle with obstacle representation on map
3. UUV and FLS beam
4. Pipeline

Local obstacle map

Local map is a small size obstacle map, represents the change in the large map. Size is `60x60` for BlueROV sonar (size depends on the FLS range). Local map is fixed to UUV in TF tree

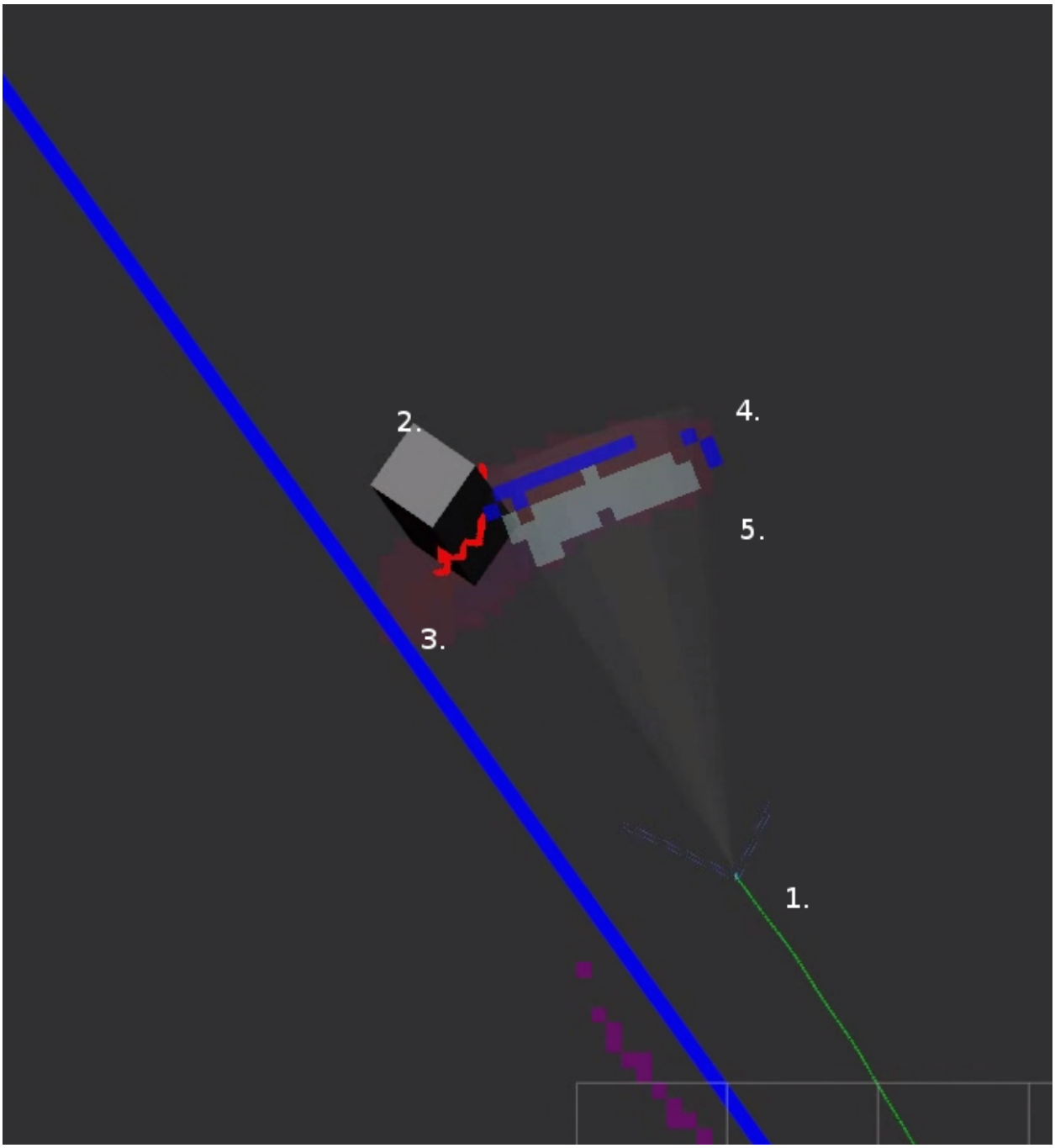


Fig. 2.: Local obstacle map in RVIZ

1. UUV
2. Obstacle
3. Transparent blob: World oriented obstacle map (for avoidance)
4. Local obstacle map: Blue - added value
5. Local obstacle map: Grey - removed value

Static obstacles:

Static obstacles always spawned directly ahead of UUV in a fixed distance. Can be turned on/off with `enable_obstacles=True/False` parameter. If `enable_debris` also `True`, the static obstacles will be generated on the seafloor (multiple at once, representing debris for SSS)

Dynamic obstacles:

Dynamic obstacles will spawned on a perpendicular path ahead of the UUV, in the FLS range (~25m), moving with a constant speed and heading. Can be turned on/off with `enable_dynamic_obstacles=True/False` parameter.

Picked up static and dynamic obstacles are represented also in a local map (60x60 occupancy grid for BlueROV, 100x100 occupancy grid for LEC3 FLS). Local map is represented in a local UUV's coordinate system.

AIS obstacles:

The information about surface traffic is provided by AIS, while the UUV is at the surface. AIS provides ship position, heading and speed etc. These objects are represented as 'dynamic obstacles'. At initialization, the node generates a list of a given number (`dynamic_obstacles_count`) obstacles with random position, heading and speed around the UUV (`1km^2^` area). The objects are `20x20m` in size in the obstacle map. The positions are updated `1 Hz`.

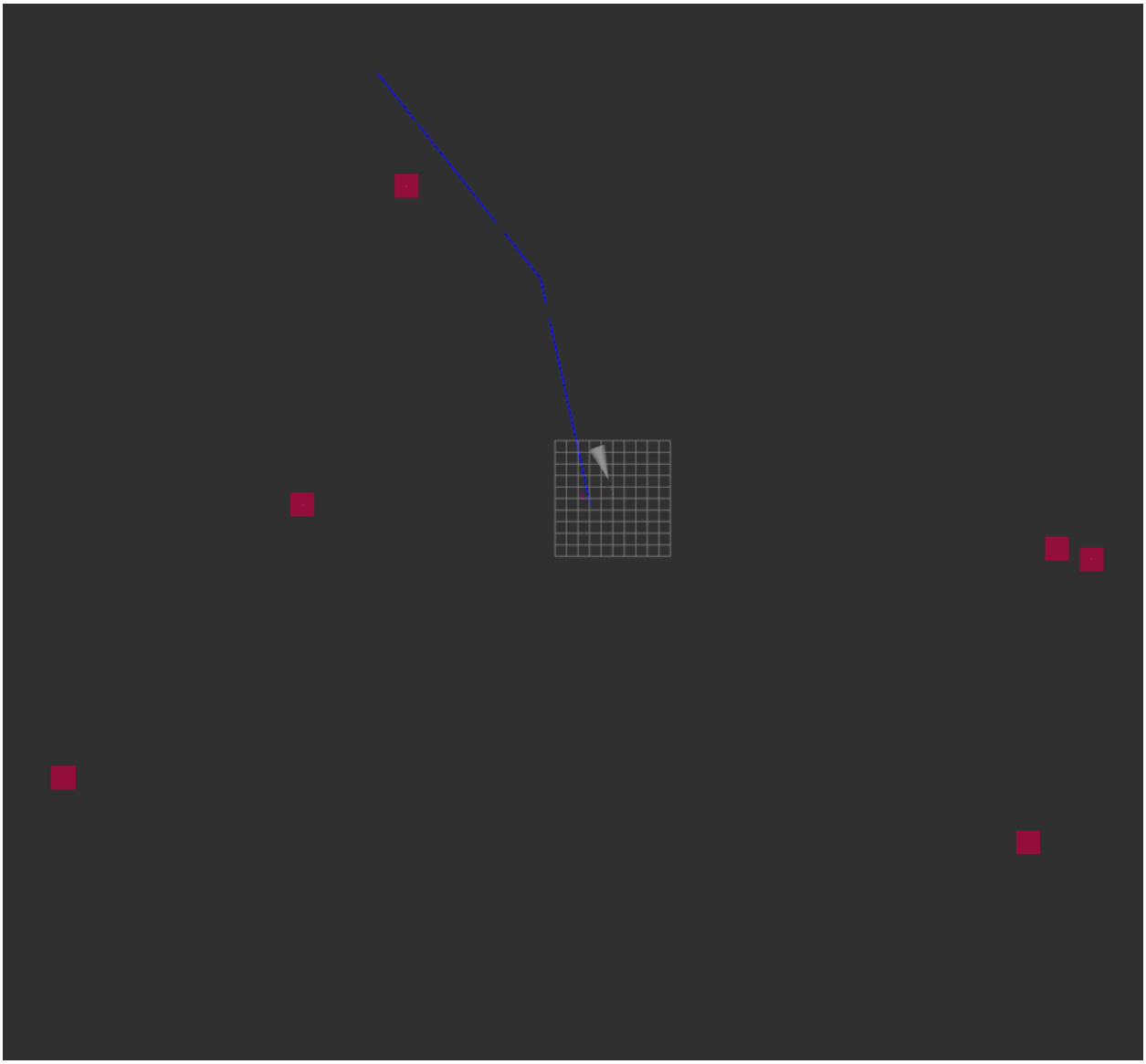


Fig. 3.: AIS obstacles on obstacle map in RVIZ

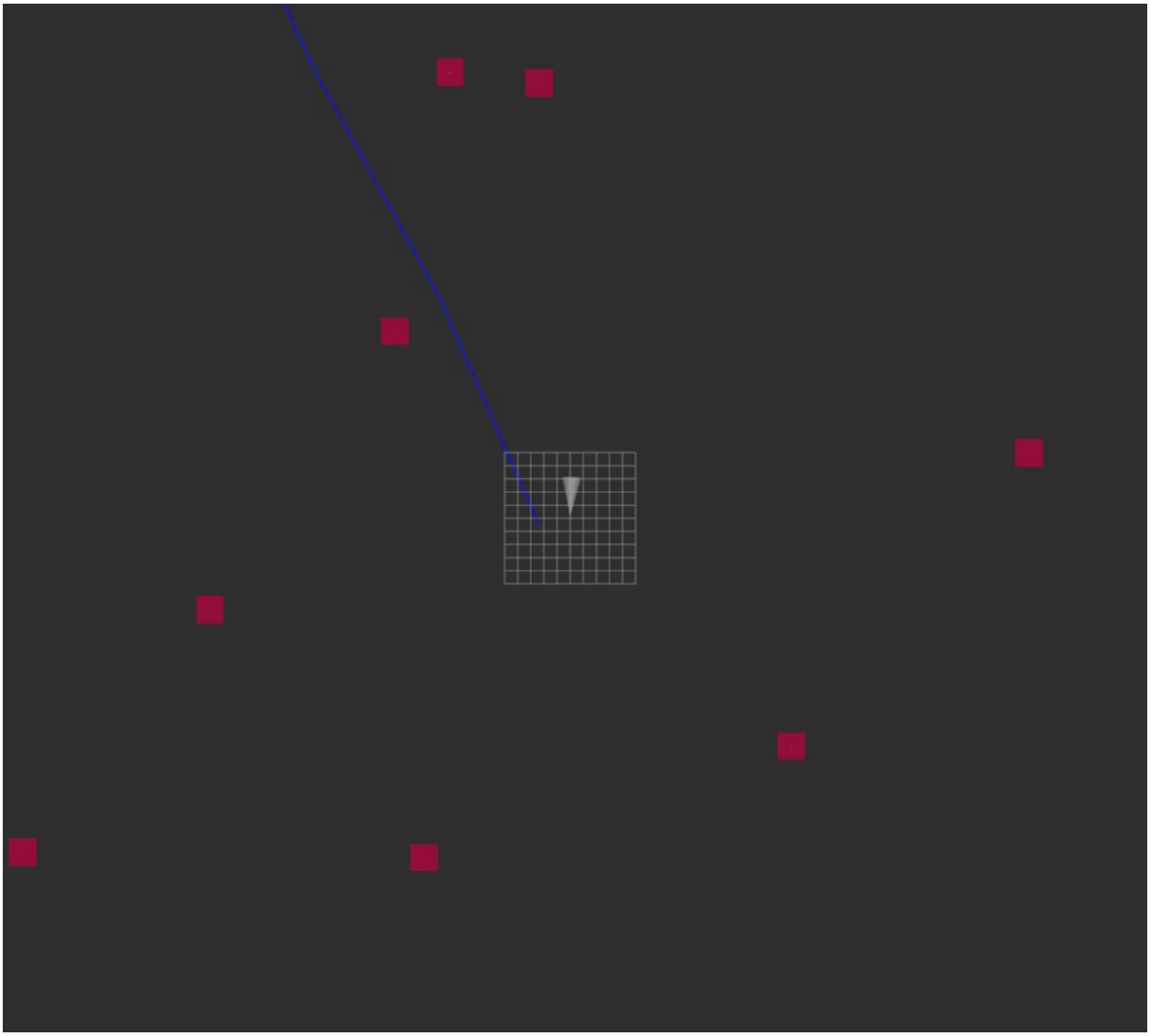


Fig. 4.: AIS obstacles GIF on obstacle map in RVIZ

Map based obstacle avoidance (update rate: `1 Hz`)

The input is the desired heading from the active mission task (eg. pipe tracking). If there is an obstacle ahead, or in the pipe heading direction, the UUV avoids them using the FLS + obstacle map output. After the obstacle cleared, mission task drives back UUV to the desired path.

Avoidance algorithm:

- If there is no obstacle ahead the UUV (no obstacle in FLS view):
  - if there is no obstacle in the desired heading:
    - Output is mission HSD
  - else:
    - Select the first safe heading using obstacle map. Angle steps are provided by `avoidance_angle_step`value, default: 20 deg.
- else:

– Select the first safe heading. Angle steps are provided by `avoidance_angle_step`value, default: 20 deg.

Note: UUV always avoids the obstacle on the far side of the pipe to avoid crossing over the pipe.

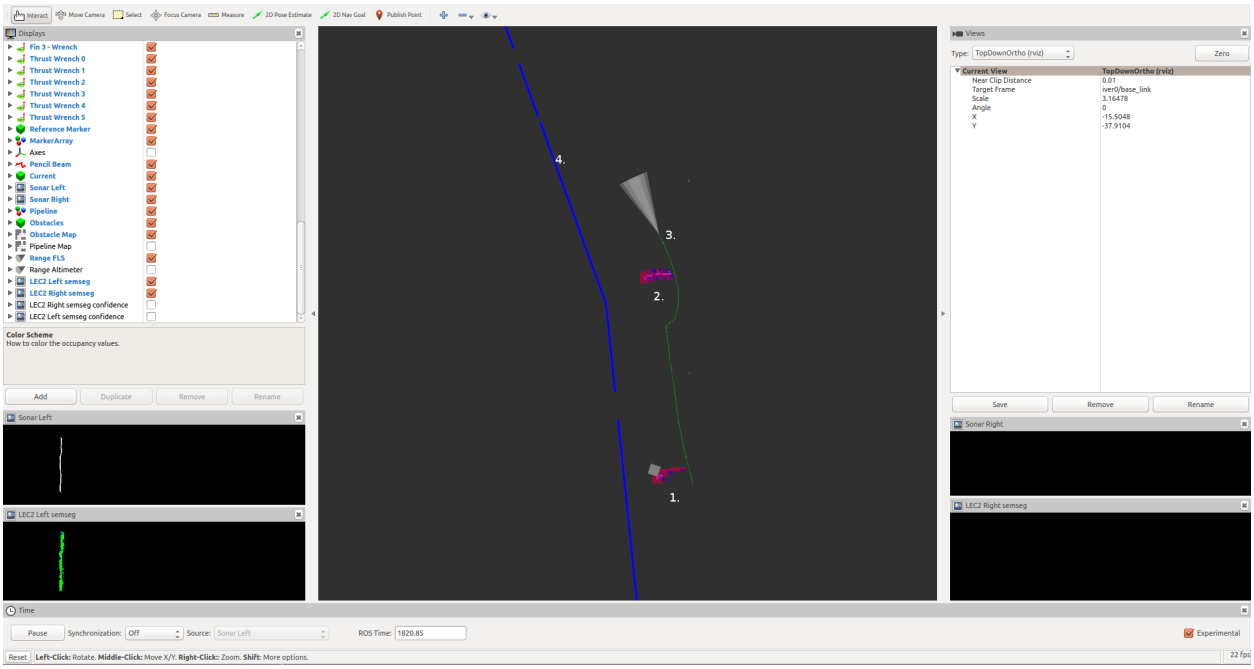


Fig. 1.: Obstacle avoidance and obstacle map in RVIZ

1. Large obstacle with obstacle map representation
2. Small obstacle with obstacle map representation
3. UUV and FLS beam
4. Pipeline

Part IV

Troubleshooting

CHAPTER

TWENTYSEVEN

COMMON ISSUES

27.1 Slurm

Slurm is very particular about configuration and small errors may cause failure. The related error messages for these configuration errors are not always helpful. Some common configuration related symptoms and underlying issues:

- *Configured execution nodes are not available for job deployment*: Often caused by incorrect node HW configuration (eg. number of CPUs, Memory, GPUs). Specified node HW configuration must always be less than or equal to real HW available. Also, HW configuration specified in *slurm.conf* must agree with that specified in *gres.conf* (particularly GPU configuration).

CHAPTER

TWENTYEIGHT

ISSUE REPORTING

The ALC git repository at https://github.com/AbLECPS/alc includes an issue tracking system. If possible, please report any issues, comments, suggestions, etc. about the ALC Toolchain using this tracking system.

28.1 Known Issues

- Docker "ros" network used by the simulation environment is configured to use a particular IP range (172.18.X.X). If this IP range happens to already in use (eg. by the default "docker0" bridge), then the "setup.sh" build will fail.
  - As a workaround, it is recommended to change the IP range of the conflicting network. For conflicts with the docker0 bridge, see https://docs.docker.com/v17.09/engine/userguide/networking/default_network/custom-docker0/