OSP Module User Guide

The Open Simulation Platform (OSP), is an open-source industry initiative for co-simulation of maritime equipment, systems and entire ships. Building on the Functional Mockup Interface (FMI) standard, the OSP can build simulations with FMUs exported from different simulation tools.

The Shipware prototype have three sample modules built with FMU: Ship Dynamics and Steering Controller, and Thrust Controller. This guidance will cover the following topics: Requirements and General Information, Ship Dynamics with FMU, Thrust Control with FMU, and Steering control with FMU.

1. Requirements and General Information

(1) System Requirements

The updated Shipware is developed on Ubuntu 20.04 and ROS noetic. For installation guide of both Shipware and the OSP, please refer to the GitHub repository: https://github.com/Autonoship/Autonoship_simulation

The FMUs can be built with software which supports FMU export, such as MATLAB Simulink, OpenModelica, etc. MATLAB Simulink is used in this demo. A version later than R2020a is recommended. For details of constructing your own FMUs with MATLAB, please refer to our guide for FMI modules: https://github.com/Autonoship/Autonoship_simulation/blob/master/FMU_module_guidance.pdf.

The OSP alone can run simulation with Co-Simulation FMUs. An independent demo using the three FMUs for OSP-only simulation can be found here: https://github.com/Autonoship/OSP_demo. This guide will give a detailed introduction of OSP simulation and ROS-OSP connection.

(2) Introduction to OSP

The OSP aims to create a maritime industry ecosystem for co-simulation of "black-box" simulation models and "plug and play" configuration of systems. OSP relies on the FMI standard and the new OSP interface specification (OSP-IS: https://opensimulationplatform.com/specification/) for the simulation models interfaces.

OSP provide a C++/C library called libcosim/libcosimc to construct simulations with FMUs. They also provide a command line interface (CLI: https://open-simulation-platform.github.io/cosim) built with libcosim to run simulations defined with OSP-IS.

By defining the connection among FMUs with an OspSystemStructure file, one can run the simulation with the CLI. Additional features include testing scenario and logging definition. By providing the required configuration files to OSP, OSP will run your testing scenarios and log the outputs to the target directory.

2. Simulation with OSP and CLI

(1) Introduction to sample FMUs

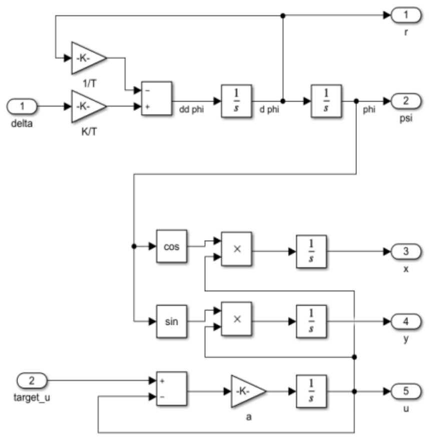
The sample simulation contains three FMUs: Ship Dynamics and Steering Controller, and Thrust Controller. These three FMUs are all built with and exported by MATLAB Simulink. The FMUs and their description files are in the folder *"/OSP_simulation_demo/fmu"*.

1) Ship Dynamics

The ship dynamics model used in the simulation is the following:

$$
\begin{aligned}
x_1 &= \dot{\psi} & \dot{x}_1 &= -\frac{x_1}{T} + \frac{K}{T}\boxed{u_1} \\
x_2 &= \psi & \dot{x}_2 &= x_1 \\
x_3 &= x & \dot{x}_3 &= x_5 \cos(x_2) \\
x_4 &= y & \dot{x}_4 &= x_5 \sin(x_2) \\
x_5 &= v & \dot{x}_5 &= a\left(\boxed{u_2} - x_5\right)
\end{aligned}
$$

The FMU is exported by MATLAB Simulink as the following. You can also find the Simulink file in the folder *"OSP_simulation_demo /simulink_model"*.



2) Steering Controller

The steering controller is designed as following:

$$
u_1 = \begin{cases} -k \cdot (x_2 + sign(x_2) * 70), when\ |x2| > 10deg \\ -k \cdot x_2, \qquad\qquad\qquad\quad when\ |x2| \le 10deg \end{cases}
$$

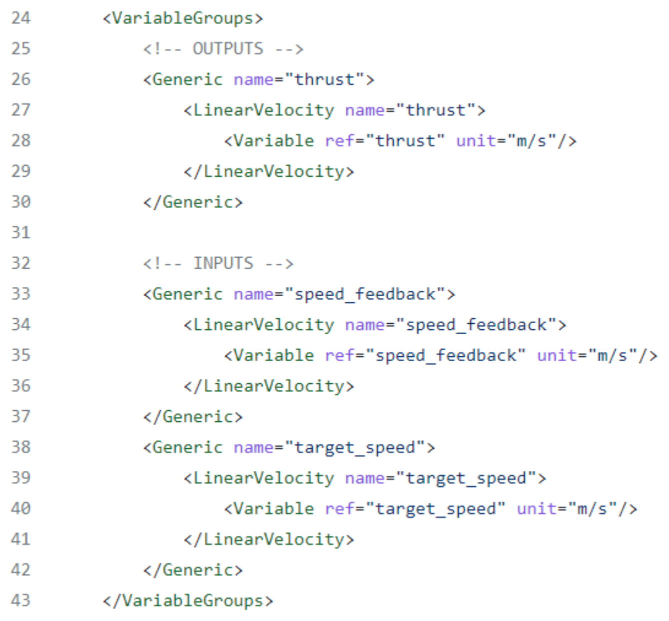
The name of the FMU is *nonlinear_controller.fmu*. The FMU is exported by MATLAB Simulink as the following. You can also find the Simulink file in the folder *"OSP_simulation_demo /simulink_model"*.

3) Thrust Controller

The steering controller is a regular PID controller. The FMU is exported by MATLAB Simulink as the following. You can also find the Simulink file in the folder "*OSP_simulation_demo /simulink_model*".

(2) Introduction to OSP interface specification

For each FMU, you need to provide a model description file, as illustrated in the OSP-IS. Sample model description files can be found in the folder *"OSP_simulation_demo/fmu"*.



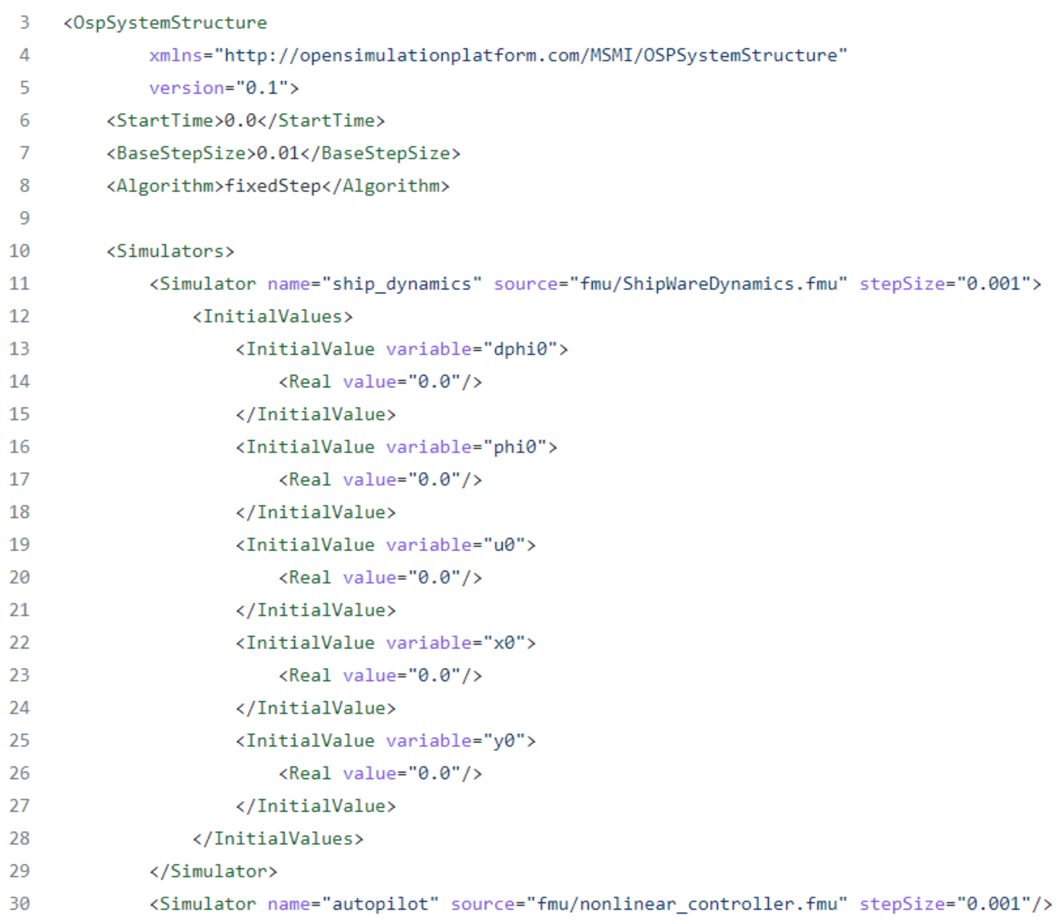
```
    <VariableGroups>
        <!-- OUTPUTS -->
        <Generic name="thrust">
            <LinearVelocity name="thrust">
                <Variable ref="thrust" unit="m/s"/>
            </LinearVelocity>
        </Generic>

        <!-- INPUTS -->
        <Generic name="speed_feedback">
            <LinearVelocity name="speed_feedback">
                <Variable ref="speed_feedback" unit="m/s"/>
            </LinearVelocity>
        </Generic>
        <Generic name="target_speed">
            <LinearVelocity name="target_speed">
                <Variable ref="target_speed" unit="m/s"/>
            </LinearVelocity>
        </Generic>
    </VariableGroups>
```

To build a simulation in OSP, you need to provide a configuration file as instructed on https://open-simulation-platform.github.io/libcosim/configuration. In this file, the following conditions will be defined:

1) StartTime: Simulation starting time, unit in seconds.
2) BaseStepSize: Base step size of co-simulation, aka, macro time step size, unit in seconds. Can be larger than the step size of the Simulink simulation (in the sample FMU, the step size is 0.001s).
3) Algorithm: Co-simulation master algorithm, currently a fixedStep algorithm is supported.
4) Simulators: Contains all sub-simulators in the system specified as <simulator> elements. The FMUs will be declared here.
5) Functions: Contains all functions to be applied on variables. Currently supported functions include LinearTransformation, Sum, VectorSum.
6) Connections: Contains all scalar and variableGroup connections between simulators, or between simulators and functions. The connection between FMU input/output pairs is defined here.

A sample configuration file can be found at *"/OSP_simulation_demo/ OspSystemStructure.xml"*



```xml
<OspSystemStructure
        xmlns="http://opensimulationplatform.com/MSMI/OSPSystemStructure"
        version="0.1">
    <StartTime>0.0</StartTime>
    <BaseStepSize>0.01</BaseStepSize>
    <Algorithm>fixedStep</Algorithm>

    <Simulators>
        <Simulator name="ship_dynamics" source="fmu/ShipWareDynamics.fmu" stepSize="0.001">
            <InitialValues>
                <InitialValue variable="dphi0">
                    <Real value="0.0"/>
                </InitialValue>
                <InitialValue variable="phi0">
                    <Real value="0.0"/>
                </InitialValue>
                <InitialValue variable="u0">
                    <Real value="0.0"/>
                </InitialValue>
                <InitialValue variable="x0">
                    <Real value="0.0"/>
                </InitialValue>
                <InitialValue variable="y0">
                    <Real value="0.0"/>
                </InitialValue>
            </InitialValues>
        </Simulator>
        <Simulator name="autopilot" source="fmu/nonlinear_controller.fmu" stepSize="0.001"/>
```

In addition to system configuration file, you can also provide Scenario file and Logging file.
The instruction of scenario definition can be found here: https://open-simulation-platform.github.io/libcosim/scenario. If loaded with a scenario file, the OSP can override the inputs or parameters at the time steps defined in the scenario file, and run the simulation according to the starting and ending time.
The sample scenario file can be found at *"/OSP_simulation_demo/scenarios/autopilot_test_scenario.json"*.

```
{ "description": "turn right, then turn left, then turn back to the initial course",
  "defaults": {
    "model": "autopilot",
    "variable": "target_course",
    "action": "override"
  },
  "events": [
    {
      "time": 0.0,
      "model": "thrust_PID",
      "variable": "target_speed",
      "action": "override",
      "value": 7.0
    },
    {
      "time": 60.0,
      "model": "autopilot",
      "variable": "target_course",
      "action": "override",
      "value": 1.0
    },
    {
      "time": 360.0,
      "value": -1.0
    },
    {
      "time": 660.0,
      "value": 0.0
    },
```

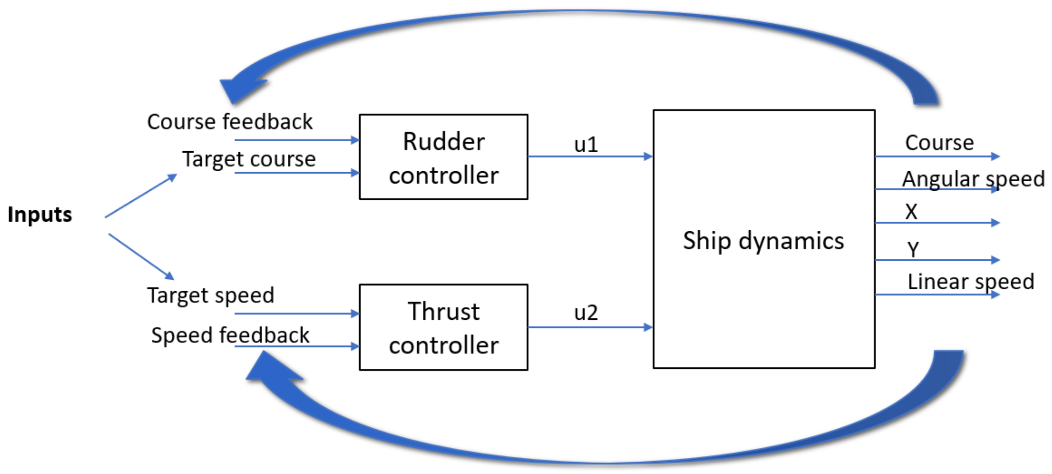
The instruction of logging file can be found here: https://open-simulation-platform.github.io/libcosim/logging. With a logging file, the OSP will only record the variables declared.
The sample logging can be found at *"/OSP_simulation_demo/LogConfig.xml"*

```xml
<?xml version="1.0" encoding="utf-8" ?>

<simulators>
    <simulator name="ship_dynamics" decimationFactor="100">
        <variable name="x"/>
        <variable name="y"/>
    </simulator>
    <simulator name="autopilot">
        <variable name="delta"/>
    </simulator>
</simulators>
```

(3) Structure of demo simulation

The demo simulation accepts two inputs: target course and target speed. The structure is defined in the *"OspSystemStructure.xml"* and is shown as following:



The *rudder_controller (nonlinear controller)* takes the *course* output from the *ship_dynamics* and uses it as the input *course_feedback*. The *thrust_controller (PID controller)* takes the *linear_speed* output from the *ship_dynamics* and uses it as the input *speed_feedback.*

(4) Usage of sample OSP simulation

With all the files above defined, you can use the CLI to run the simulation now. A instruction of using the simulation can be found here: https://github.com/Autonoship/OSP_demo/tree/main/OSP_simulation_demo.

With CLI, you can go to *"/OSP_simulation_demo"* in the terminal and run this command to run the simulation:

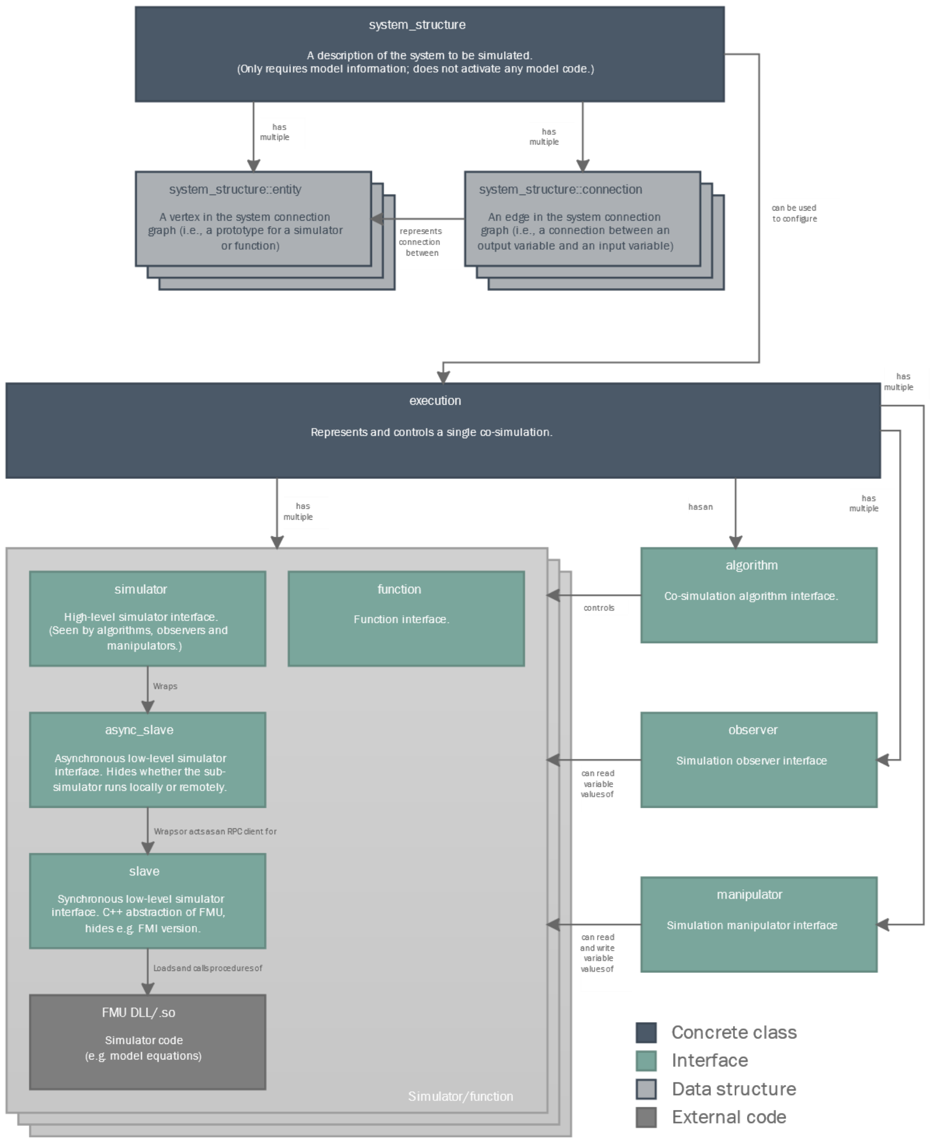
cosim run OspSystemStructure.xml -b 0.0 -e 1200.0 --scenario scenarios/autopilot_test_scenario.json -v

This command will run the simulation from t=0.0s to t=1200.0s, and set the inputs as defined in the scenario file.

3. Simulation with libcosim

(1) Architecture of libcosim based simulation

The architecture of libcosim can be seen here: https://open-simulation-platform.github.io/libcosim.


system_structure
A description of the system to be simulated.
(Only requires model information; does not activate any model code.)
has multiple
has multiple
system_structure::entity
A vertex in the system connection graph (i.e., a prototype for a simulator or function)
represents connection between
system_structure::connection
An edge in the system connection graph (i.e., a connection between an output variable and an input variable)
can be used to configure
execution
Represents and controls a single co-simulation.
has multiple
has multiple
hasan
has multiple
simulator
High-level simulator interface.
(Seen by algorithms, observers and manipulators.)
function
Function interface.
algorithm
Co-simulation algorithm interface.
controls
Wraps
async_slave
Asynchronous low-level simulator interface. Hides whether the sub-simulator runs locally or remotely.
observer
Simulation observer interface
can read variable valuesof
Wrapsor actsasan RPC client for
slave
Synchronous low-level simulator interface. C++ abstraction of FMU, hides e.g. FMI version.
manipulator
Simulation manipulator interface
can read and write variable valuesof
Loadsand callsproceduresof
FMU DLL/.so
Simulator code
(e.g. model equations)
Simulator/function
Concrete class
Interface
Data structure
External code

The libcosim will construct an instance of the execution class, and add simulators and functions to it according to the system structure configuration file. The simulators are the interfaces to the FMUs, and the functions can provide some basic operations on the variables

Currently libcosim can only run a fixed-step algorithm. An observer is used to read or record values from the OSP simulation, and a manipulator is used to set variable values.

(2) libcosim based simulation

In the demo at https://github.com/Autonoship/OSP_demo/tree/main/cpp_demo, we are using a *file_observer* to record variables according to the logging file, and a *scenario_manager* as the manipulator to set input variables according to the scenario file.

This demo shows the usage of libcosim in C++ code. The code follows the procedure:

1) load configuration
2) construct an instance of execution()
3) initialize with fixed step algorithm
4) build the simulation according to the configuration
5) add observer
6) load scenario
7) run the simulation

The compiling instruction can be found at: https://github.com/Autonoship/OSP_demo. After compiling, you can run the demo code by:

```
cd cpp_demo/build
./cosim_demo
```

4. ROS-OSP connection

(1) Introduction to ROS service

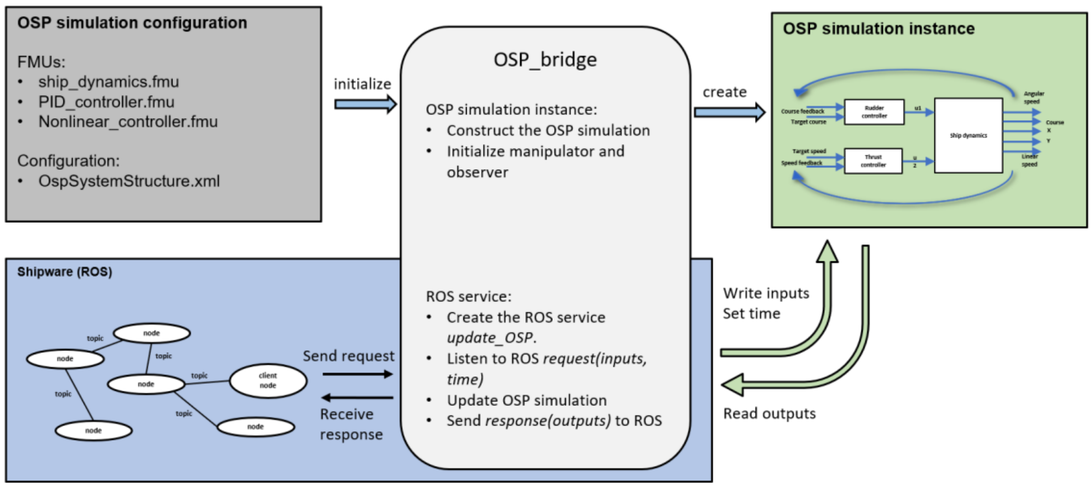
Unlike *Publisher / Subscriber* structure, *ROS Service* is designed for RPC request / reply interactions. *Request / Response* is done via a *Service*, which is defined by a pair of messages: one for the request and one for the response.

Services are defined using *srv* files, which consists of the variable names and types for the request and response. You can define your own *srv* file to work with your own request / response messages.

Once a *Service* is called, a callback function will be run. The callback function is defined in the code, with the request and response as inputs.

(2) Introduction to OSP-bridge

The OSP-bridge is a ROS node which construct an OSP simulation and provides a service for other nodes to interact with the OSP simulation.



At the initialization stage, OSP_bridge will read the OSP configuration file and create an instance of the OSP simulation. An override_manipulator is used here to modify values of variables in the OSP simulation, and a last_value_observer is used to observe the latest output of the OSP simulation.

After the OSP simulation is ready, OSP_bridge will initialize a ROS Service. When other ROS nodes want to interact with the OSP simulation, they should make a request to OSP_bridge and indicate the input values and simulation time in the request. OSP_bridge will modify the input values and run the OSP simulation according to the given time. Then the output of the OSP simulation will be sent back to the nodes as the response.

In the demo OSP_bridge class, the OSP simulation will be handled by a pointer pointing to an instance of execution(). If the path to OSP simulation configuration is not specified, it will read the default OSP simulation of with the three FMUs we have discussed.

When using the override_*manipulator* to set inputs, you need to provide the simulator index and the variable index. The simulator index can be found in the index_maps, and OSP_bridge provides function to print out the index for each simulator. The variable index can be found in the “modelDescription.xml” in the FMU. The following figure shows the variable delta in an FMU. The valueReference=”0” is its index.

```
<ScalarVariable description="Input Port: delta" name="delta" variability="continuous" valueReference="0" causality="input">
      <!--Index = 1-->
   <Real start="0"/>
 - <Annotations>
   - <Tool name="Simulink">
         <Data tag="cosimTransformedInput_1" elementAccess=""/>
      </Tool>
   </Annotations>
</ScalarVariable>
```

(3) Sample simulation with OSP-bridge
The ROS package of the sample simulation can be found at https://github.com/Autonoship/Autonoship_simulation/tree/master/osp_ros_demo.

In this package, an *updateState* service is defined in *srv/updateState.srv*.

```
std_msgs/Float64 time
std_msgs/Float64 target_speed
std_msgs/Float64 target_course

---

std_msgs/Float64 phi_dot
std_msgs/Float64 phi
std_msgs/Float64 x
std_msgs/Float64 y
std_msgs/Float64 v
```

The upper part defines the message types and the variable names of the request, and the lower part defines the message types and the variable names of the response. If you are building your own OSP simulation and want to modify the service, you can just add variables here.

The *OSP_bridge* node is defined in "*src/osp_ros_demo.cpp"*, The service is defined as following:

```
    ros::ServiceServer service = nh.advertiseService("update_OSP", &OSP_Bridge::updateOSP, &bridge);
```

The callback function *OSP_Bridge::updateOSP()* will set the *target_speed* and *target_course* of the OSP simulation, and simulate till the given time. Then the simulation results will be sent back in the response.

OSP_Bridge ->pExe is the pointer to the instance of execution(), and is constructed with the default path *"osp_ros_demo/osp_simulation_model/"*, *override_manipulator* and *last_value_observer*. You can write your own callback function, and play with the OSP simulation with the new service. For information about how to write a ROS service, please look at http://wiki.ros.org/Services.

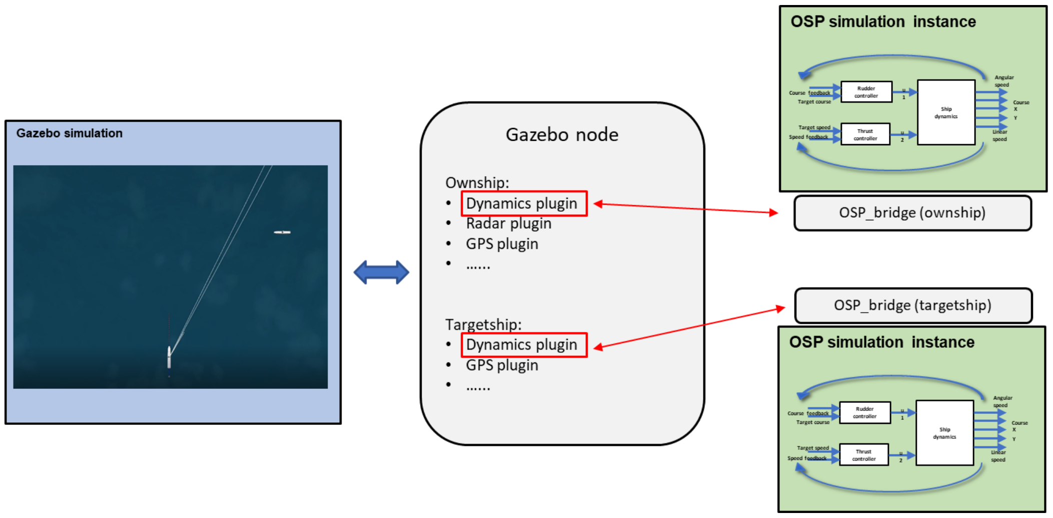
To change the simulation speed, you can modify the *<BaseStepSize>* in "*osp_ros_demo/osp_simulation_model/OspSystemStructure.xml"* and the *<max_step_size>* of *<physics>* in *osp_ros_demo/*worlds/autonoship.world. The *<max_step_size>* of *<physics>* has to be an integer multiple of the *<BaseStepSize>.*

Be careful with the step size. The Gazebo simulation is based on physics engine (we are using the ODE engine), and the stability might be ruined by a large step size.

To run the sample ROS-OSP simulation, use the command:

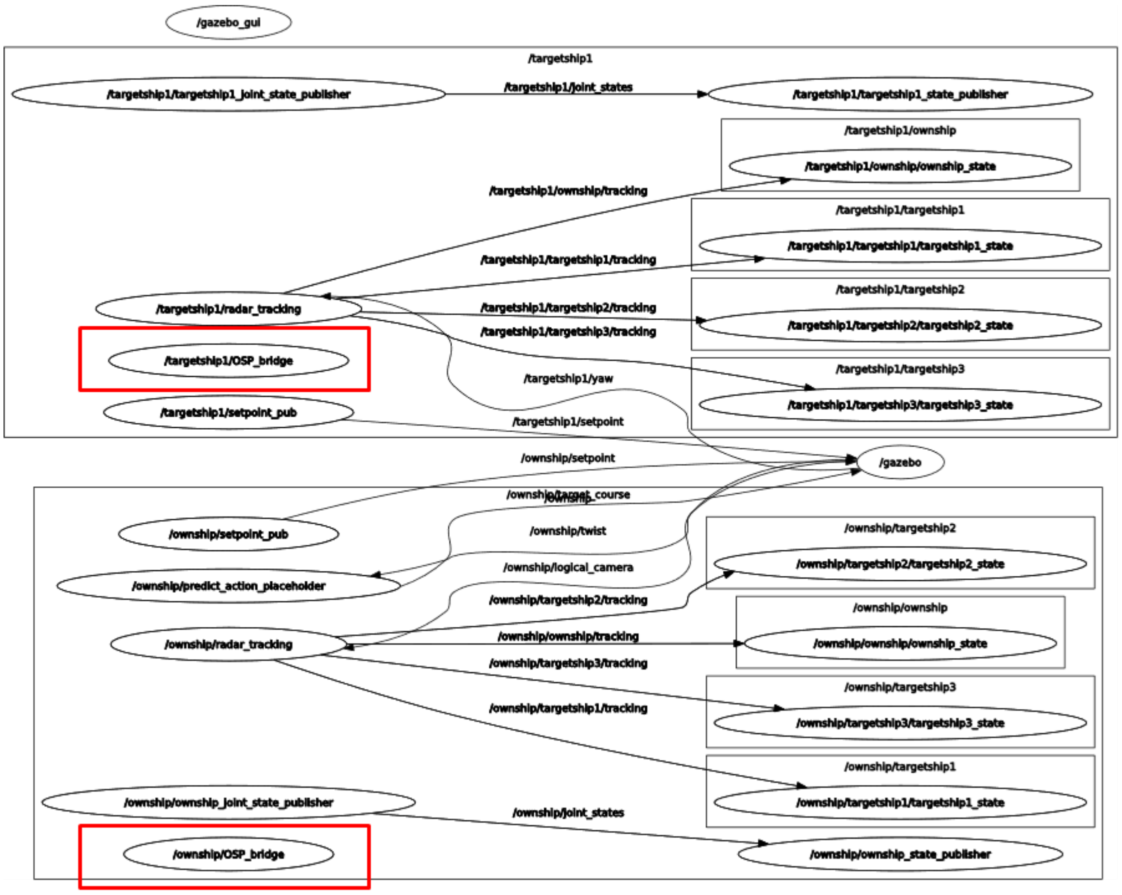
roslaunch osp_ros_demo osp_autonoship_gazebo.launch scenario:=scenario1

This will run the scenario 1 where the ownship is equipped with the placeholder for the collision avoidance module. Both the ownship and targetship are simulating the ship dynamics with OSP, and the structure is as follows:



Each ship has its own OSP_bridge, which is operating its own OSP simulation. The two OSP_bridge has different namespace, and the targetship is not equipped with a collision avoidance module. Thus, the targetship will keep moving straight to the left, while the ownship will steer to avoid collision.

The ROS graph of this simulation is shown in the following. The red boxes are highlighting the OSP_bridge nodes for each ship.


/gazebo_gui
/targetship1
/targetship1/targetship1_joint_state_publisher
/targetship1/joint_states
/targetship1/targetship1_state_publisher
/targetship1/ownship
/targetship1/ownship/ownship_state
/targetship1/ownship/tracking
/targetship1/targetship1
/targetship1/targetship1/targetship1_state
/targetship1/targetship1/tracking
/targetship1/radar_tracking
/targetship1/targetship2
/targetship1/targetship2/tracking
/targetship1/targetship3/tracking
/targetship1/targetship2/targetship2_state
/targetship1/OSP_bridge
/targetship1/yaw
/targetship1/targetship3
/targetship1/targetship3/targetship3_state
/targetship1/setpoint_pub
/targetship1/setpoint
/ownship/setpoint
/gazebo
/ownship/target_course
/ownship
/ownship/setpoint_pub
/ownship/twist
/ownship/targetship2
/ownship/targetship2/targetship2_state
/ownship/predict_action_placeholder
/ownship/logical_camera
/ownship/targetship2/tracking
/ownship/ownship
/ownship/radar_tracking
/ownship/ownship/tracking
/ownship/ownship/ownship_state
/ownship/targetship3/tracking
/ownship/targetship3
/ownship/targetship1/tracking
/ownship/targetship3/targetship3_state
/ownship/targetship1
/ownship/ownship_joint_state_publisher
/ownship/targetship1/targetship1_state
/ownship/joint_states
/ownship/OSP_bridge
/ownship/ownship_state_publisher