LMS

University of Patras

DOHC Manual

LMS

Laboratory for
Manufacturing Systems
& Automation

LABORATORY FOR MANUFACTURING SYSTEMS & AUTOMATION

Deformable Object Handling Controller Manual

Laboratory For Manufacturing Systems & Automation
University of Patras

Table of Contents

Installation

Use the following steps to install the application. The steps have been tested on ubuntu 18.04 and ubuntu 20.04. It is recommended to use the DOHC image that uses ROS melodic with ubuntu 18.04 and ROS melodic and the image that uses ROS noetic with ubuntu 20.04 and ROS noetic.

1. Install Docker

2. Extract lms_DOHC.zip.

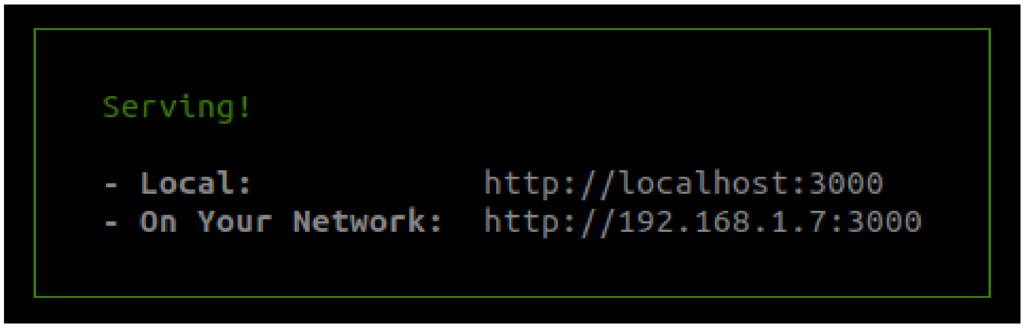
3. Find the dohcData folder in lms_DOHC.zip and place it anywhere in the pc on which the DOHC will run.

4. Open a terminal inside the directory where lms_WC.tar file is and run `sudo docker load -i lms_WC.tar`

5. Run `sudo docker run --rm -it --net=host --name wc-mongo mongo mongod --bind_ip_all`. In case a mongoDB server is running on the host machine, it should be closed using `sudo systemctl stop mongod` before step 5.

6. Open a new terminal in home directory and run `sudo docker run --rm -it --net=host -v<path-do-dohcData>/dohcData:/host-fs --name wc lms/wc:noetic`

After the successful completion of the above steps, an output similar to that shown in Figure 1 will appear on the terminal.



```
Serving!

- Local:            http://localhost:3000
- On Your Network:  http://192.168.1.7:3000
```

Figure 1 The DOHC address.

To use the DOHC, follow the following steps:

7. Open a browser (a chromium based browser is recommended) and navigate to the address shown in the terminal(192.168.1.7:3000 or localhost:3000 in Figure 1).

8. Click on the “load diagram to execute” button (Figure 8).

9. On the dialog that appears on the screen, click on Operations and execute the sample operation to confirm that the installation is successful.

After the first installation, only steps 5 and 6 are necessary to start the DOHC. To shut down DOHC it is important to close the container opened in step 6 first and then the container opened in step 5. Containers should be closed using ctrl + C. Closing the containers in a not appropriate way may result in lost data.

In case it is desired to use DOHC in a pc different than the one that holds the ROS master, replace the command in step 6 with `sudo docker run --rm -it --entrypoint /bin/bash --net=host -v<path-do-dohcData>/dohcData:/host-fs --name wc lms/wc:noetic -c "/entrypoint.sh -ros_master 168.192.1.23"`.

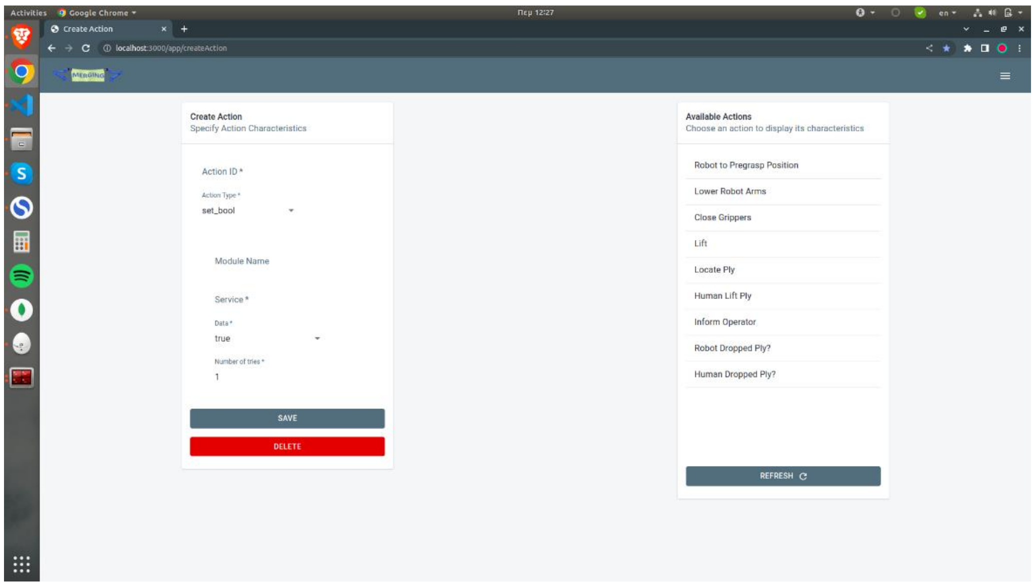
General

Deformable Object Handling Controller is a tool that provides the ability for high level orchestration of the shopfloor resources and monitoring the executions' process. The provided software has been tested using ROS noetic on ubuntu 20.04. DOHC follows a hierarchical approach for constructing the execution process based on 5 levels of activities: Orders, Jobs, Tasks, Operations and Actions. The highest level is the Order, which is a superset of activities that take place in a certain period in a shop floor. An order is decomposed in a set of Jobs which take place in the workstation and constitute a group of tasks. Tasks constitute a group of operations and each operation consists of a group of actions which can be found at the lowest level and contain information about how a module, or a resource can execute a specific low level activity. An activity can contain sub activities from any abstraction layer, as long as the sub activities do not belong to an abstraction layer higher than that of the activity that contains them. To execute actions, only the details regarding the actions' execution are required. The more complex information needed for executing activities that consist of a sequence of steps (Operations, Tasks, Jobs and Orders) is represented by diagrams.

The execution details can be given to the DOHC by inserting the corresponding JSON files into the database or by using the user interface. It is strongly encouraged to use the user interface, since in this way the possibility of error is minimized. An interface that aids monitoring and sending commands to the DOHC for controlling the execution flow also exists.

Create/Edit Actions

Actions constitute the lowest level of activities DOHC supports and by combining them it is possible to form diagrams that represent activities of the higher abstraction layers. Action creation can be done in Create/Edit Action layout where the actions' characteristics can be specified by inserting the appropriate values to the corresponding fields. All actions need to have a unique action ID and an action type. Once action type is chosen, the necessary fields for the corresponding action will appear. For some action types it is also possible to fill an optional field, "module name", which allows a more detailed feedback to be printed in some cases. In case an action's execution frequently fails after the first try, it is possible for some action types to define a number of tries. If the action's execution fails, DOHC will retry to execute the corresponding action for the specified number of tries. The available action types are set_bool, trigger, delay, store, setParam, compareParam, setMode, goToPose and loadYaml.



Create Action
Specify Action Characteristics

Action ID *

Action Type *
set_bool

Module Name

Service *

Data *
true

Number of tries *
1

Available Actions
Choose an action to display its characteristics

Robot to Pregrasp Position
Lower Robot Arms
Close Grippers
Lift
Locate Ply
Human Lift Ply
Inform Operator
Robot Dropped Ply?
Human Dropped Ply?

Figure 2 Create/Edit Action Layout

Set_bool

Actions of the set_bool action type make a setBool ROS standard service call to the specified service. Apart from the Action ID and the module name, to create a set_bool action using the graphical user interface, it is necessary to fill the service field and the data field.

To manually insert a set_bool action in the database open a terminal and run mongod. Then insert the document in the database:

```
db.Actions.insertOne({
 "ActionID": "Unique ID",
 "ActionType": "set_bool",
 "Data": "true",
 "ModuleName": "optionalModuleName",
 "Service": "/setboolsrv"
})
```

Data can have the value true or false.

Trigger

Actions of trigger action type make a trigger ROS standard service call to the specified service. The only field that needs to be specified apart from Action ID and Action type is the service field.

To manually insert a trigger action in the database, open a terminal and run mongod. Then insert the document in the database:

```
db.Actions.insertOne({
 "ActionID": "unique ID",
 "ActionType": "trigger",
 "ModuleName": "optionalModuleName",
 "Service": "/triggersrv"
})
```

Delay

Delay actions stay in "running" state for the specified time duration and then they complete, enabling the execution to continue. The only field needed for delay actions is the duration field.

To manually insert a delay action in the database, open a terminal and run mongod. Then insert the document in the database:

```
db.Actions.insertOne({{
 "ActionID": "unique ID",
 "ActionType": "delay",
 "Duration": "1"
})
```

Store

Store actions get the specified parameter's value and replace it with the following value: <parameter value>*multiplicationFactor + additionConstant. For example, in case the desirable outcome of a store action is to increase the value of a parameter by 3, multiplication factor should be set to 1 and addition constant should be set to 3. Apart from Action ID and actionType, to create a store action the AdditionConstant, MultiplicationFactor, Parameter and ParameterType fields must be filled. For the store action to be executed successfully, the specified parameter must have a compatible value (of type int or float).

To manually insert a store action in the database, open a terminal and run mongod. Then insert the document in the database:

```
db.Actions.insertOne({
 "ActionID": "unique ID",
 "ActionType": "store",
 "AdditionConstant": "1",
 "MultiplicationFactor": "1",
 "Parameter": "/parameter",
 "ParameterType": "int"
})
```

MultiplicationFactor and AdditionConstant should be integers. ParameterType can have the values “int” or “float”.

SetParam
Set parameter actions set the specified parameter to the given value. SetParam actions require parameter, parameterType and value fields to be filled. ParameterType can be int, float, bool or string.

To manually insert a setParam action in the database, open a terminal and run mongod. Then insert the document in the database:

```
db.Actions.insertOne({
 "ActionID": "unique ID",
 "ActionType": "setParam",
 "Parameter": "/parameter",
 "ParameterType": "int",
 "Value": "12"
})
```

CompareParam
Compare parameter actions compare the specified parameter’s value to a given value. The action completes successfully when the given statement is true or returns an error otherwise. The supported parameter types for this action are int, float, bool and string.

To manually insert a compareParam action in the database, open a terminal and run mongod. Then insert the document in the database:

```
db.Actions.insertOne({
 "ActionID": "unique ID",
 "ActionType": "compareParam",
 "Comparison": "equal",
 "Parameter": "/parameter",
 "ParameterType": "int",
 "Value": "0"
})
```

Comparison can take the values "greater", "less" and "equal". ParameterType can take values "int', "bool", "float" and "string".

SetMode

Setmode actions make a service call of the setMode type. The setMode service is defined in the wc_msgs package which is given along with the DOHC. This service can be used by a module if the module depends on the wc_msgs package.

To manually insert a setMode action in the database, open a terminal and run mongod. Then insert the document in the database:

```
db.Actions.insertOne({
 "ActionID": "unique ID",
 "ActionType": "set_mode",
 "ModuleName": "sedModeModuleName",
 "Service": "/setMode",
 "Mode": "3"
})
```

GoToPose

GoToPose actions send a pose to a ROS action server. GoToPose actions require ActionServerName, OrientationX, OrientationY, OrientationZ, OrientationW, PositionX, PositionY, PositionZ and WaitResultTimeout fields to be filled.

To manually insert a GoToPose action in the database, open a terminal and run mongod. Then insert the document in the database:

```
db.Actions.insertOne({
"ActionID": "gotopose ID",
"ActionServerName": "/actionserver",
"ActionType": "goToPose_action",
"ModuleName": "modul name ",
"NumOfTries": "1",
"OrientationW": "1",
"OrientationX": "0",
"OrientationY": "0",
"OrientationZ": "0",
"PositionX": "0",
"PositionY": "0",
"PositionZ": "0",
"WaitResultTimeout": "6"
})
```

LoadYaml

LoadYaml actions load the values from the specified yaml file to the parameter server. The yaml file needs to be in the dohcData folder. To create a loadYaml action, the path to the yaml file should be specified in the corresponding field. The path should not be the absolute path but the relative path, starting from the dohcData folder. For example, if the path to a yaml file is "/home/usr/Desktop/dohcData/folder/parameters.yaml", the corresponding field on layout should be filled with "myfolder/parameters.yaml".

To manually insert a loadYaml action in the database, open a terminal and run mongod. Then insert the document in the database:

```
db.Actions.insertOne({
 "ActionID": "unique ID",
 "ActionType": "loadYaml",
 "YamlPath": "/loadyaml"
})
```

PublishOnTopic
PublishOnTopic actions publish a message of type int32, int64, bool, float32, float64 or string on the given topic.

To manually insert a publishOnTopic action in the database, open a terminal and run mongod. Then insert the document in the database:

```
db.Actions.insertOne({
 "ActionID": "unique ID",
 "ActionType": "publishOnTopic",
 "MessageType": "int32",
 "Topic": "/topic",
 "Message": "1234"
})
```

WaitFor
WaitFor actions stay in "running" state until the specified target value is published in the specified topic. The target value can be an int32, bool or string.

To manually insert a waitFor action in the database, open a terminal and run mongod. Then insert the document in the database:

```
db.Actions.insertOne({
 "ActionID": "unique ID",
 "ActionType": "waitFor",
 "TargetValue": "false",
 "TargetValueType": "bool",
 "Topic": "/topic",
 "Timeout": "30"
})
```

Reusability

Actions created using Create/Edit Action layout are stored in database and can be used to form diagrams, making it possible to use the same action in many diagrams. In case a change in the actions details is necessary, only one action needs to be edited, it is not necessary to change the action's details for every activity that contains it.

Editing Actions

When in Create/Edit Action layout, the available actions appear on the right side of the page on a list that can be refreshed with the use of the corresponding button. An action can be selected with a click on the corresponding item on the list and the action's details will show up on the left side of the page. Then it is possible to change the action's details and save the updated action or to save the edited action as a new action while keeping the original action intact. To save as new, provide a new Action ID and save. To delete the selected action click on the delete button.

Local Actions

Some actions are not intended to be reused and will execute only during a specific activity. Those actions do not need to be stored separately in database and can be stored in the corresponding activity. It is possible to create a local action from the "Create/Edit Diagram" layout by dropping on the grid the "Local Action" element from the sidebar. Once dropped, a dialog appears that enables the user to insert the action details. Local actions can belong to any of the available action types.

Create/Edit Diagrams

To create a diagram navigate to Create/Edit Operation, Create/Edit Task, Create/Edit Job or Create/Edit Order layout. On the sidebar on the left lie the elements that can be dropped on the grid to form diagrams. The elements that can be dropped consist of "Manage Counters", "Add Local Action" and of activities belonging to the sublayers of the activity being created.

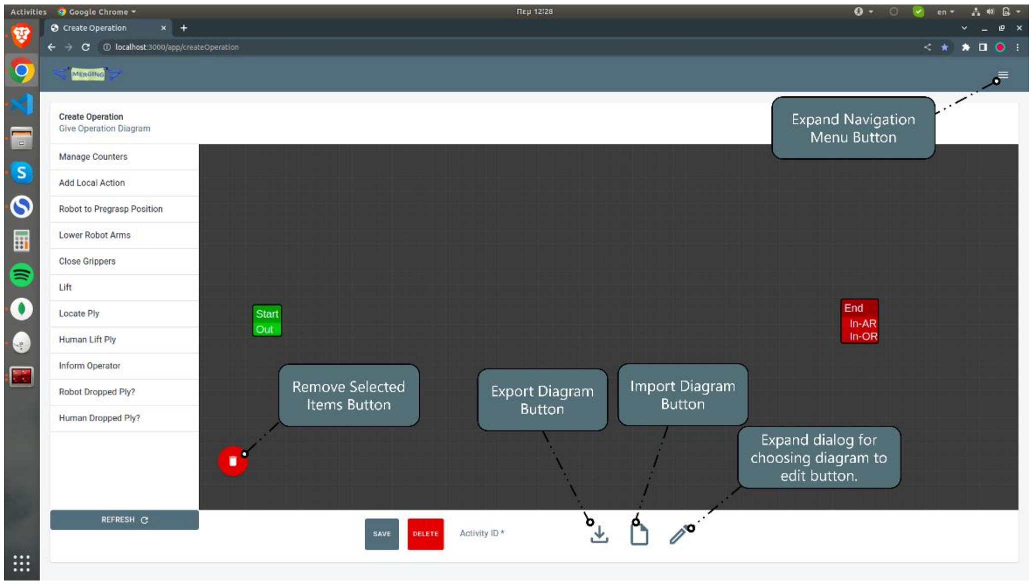
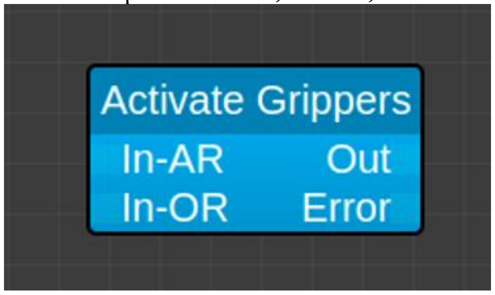


Figure 3 Create/Edit Diagram Layout

Dropping Activities On the Grid.

Activities can be dragged from the sidebar and dropped on the grid: expand the list with the available activities from the desired sublayer and drop any of the available activities on the grid. After an activity is dropped, a node appears on the grid that has different color, depending on the abstraction layer the represented activity belongs to. The nodes have four ports: In-AR, In-OR, Out and Error.



When connecting the Out port of an activity A to an In port of an activity B, the activity B depends on the activity A. When multiple out ports link to an activity's All Required – In (In-AR) port, all the activities from whose the out ports the links start should complete before the activity that carries the In port starts executing.

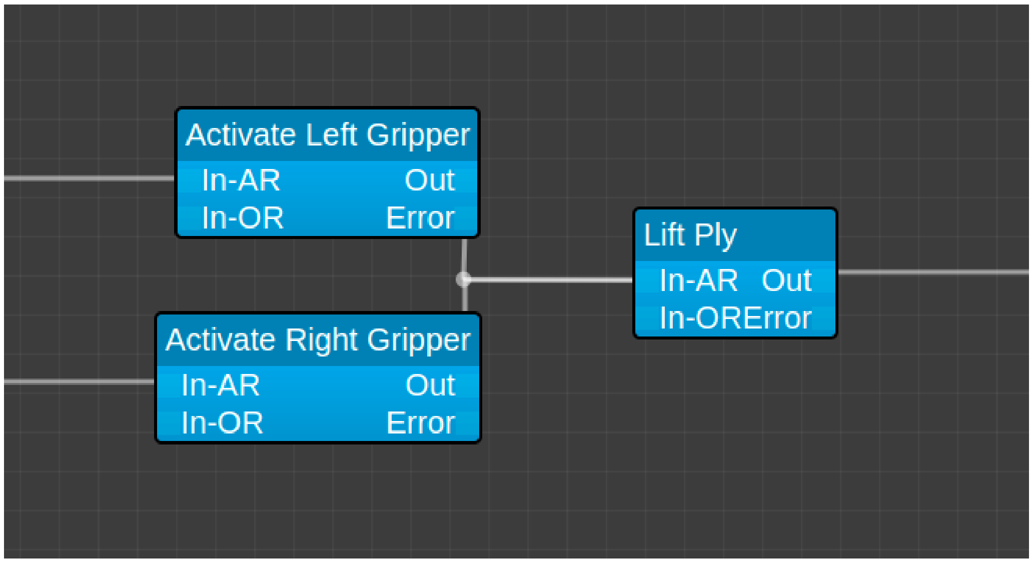


Figure 4 Lift ply will execute after "Activate Left Gripper" and "Activate Right Gripper"

When multiple out ports link to an activity's One Required-In (In-OR) port, the activity that carries the In-OR port will execute after at least one of the activities that are linked to the In-OR port is completed.

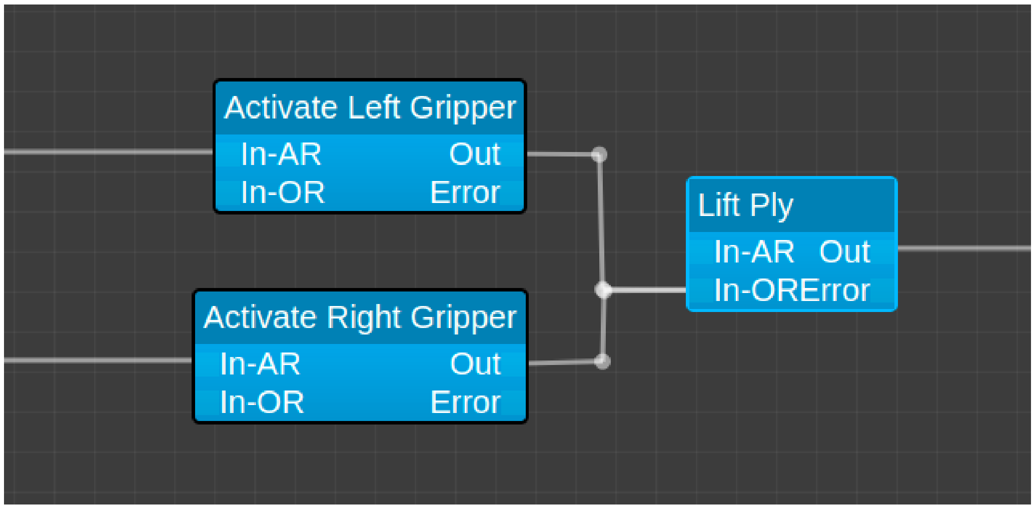


Figure 5 When one of the "Activate Left Gripper" and "Activate Right Gripper" is completed, lift ply will start executing.

It is important to link the activities using the correct port, otherwise issues can occur: The schedule part shown Figure 5 commands the controller to lift the ply when one gripper is activated without waiting for the second one before proceeding.

Manage Counters & Loops

After dropping the "Manage Counters" element on the grid, a layout shows up with which it is possible to create a Local Action that can initialize a counter, alter a counter's value or compare a counter to a value.

- Initialize Counter: Choose a value with which the counter specified in the corresponding field will be initialized.

- Set Counter Value: The specified counter's value will change after executing an activity of this type. If x is the counter's value, the value will be x = x*MultiplicationFactor + AdditionConstant.

- Compare Counter: The comparison specified in the comparison field will take place between the counter and the given value. If the comparison returns true, the action will be considered by DOHC as a successfully completed action. In case the comparison returns false, the action will be considered as an action in which an error occurred.

The counters are local: two different activities that execute in parallel can both have a counter i without causing any collisions. It gets easily understood that two instances of the same activity can also run in parallel without the need to have counters with different names.

Error Handling

Error handling is done by using the "error" port on diagram's nodes: a link can be created between this port and any activity that is desired to be executed in case of an error in the corresponding activity. After the execution that handles the error finishes, it is possible to continue with a different flow or to return to the activity that did not succeed and try to execute it again.

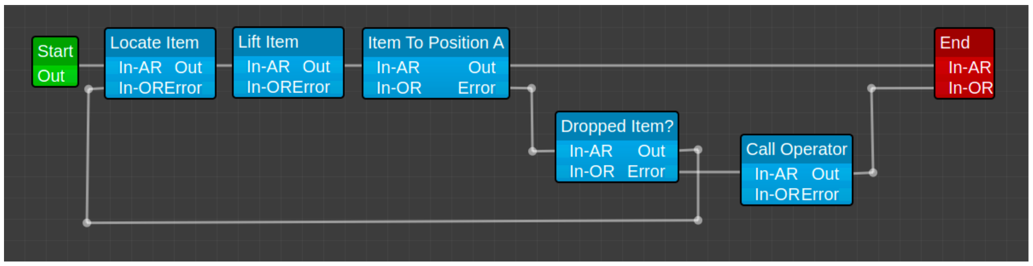


Figure 6 Error Handling Example

In the event of an error in "Item To Position A" in Figure 6, a check will take place. If the item is dropped, it will be located again and a new attempt to take it to position A will be made. In case the item is not dropped, the operator will be informed. Please notice how a loop is formed here: every time the item is dropped a new attempt will take place to bring it to position A for an infinite amount of times. A counter could be used here to set a number of attempts that will take place in case the item is dropped.

Editing Diagrams

When in Create/Edit Operation, Create/Edit Task, Create/Edit Job or Create/Edit Order layout, it is possible to choose a diagram to edit by pressing the pencil button. After a diagram is chosen it appears on the grid. Elements can be removed from the diagram by pressing dedicated button (see Figure 3) on the bottom left of the grid after selecting the corresponding elements. It is possible to select multiple elements with Shift + click. Multiple elements can be moved at once when they are selected and dragged on the grid. It is possible to save the updated diagram by pressing the save button or to keep the original diagram intact and save the edited one as new by providing a new activity ID and pressing the save button. Activities can be removed from database with the use of "Delete" button when an activity is loaded on the grid.

Importing/Exporting Activities

When a diagram is selected to be edited and loaded on the edit page, it is possible to export a json file that contains all the information required to execute the corresponding diagram by pressing the export button (Figure 3). The json file contains all the details required for executing the activity on the grid and all the information for executing sub-activities contained by the activity on the grid. Exporting is done by pressing the export button after a diagram is saved to the database and loaded on the grid. To import a diagram, click the import button and select a json file to import.

Because DOHC database is empty every time the docker image restarts, importing and exporting diagrams is an important element of the recommended workflow. The recommended way to handle schedules is to store the necessary activities by exporting them and importing them to the DOHC before execution or when they are needed to form other schedules.

Manually Insert Diagram JSON in Database

Diagrams are stored in the database in a format different from that in which they are created in the Create/Edit Diagram interface. To manually insert a diagram in database, a JSON file with the following format must be created:

```
db.Operations.insertOne({
"OperationID": "Locate & Lift Ply",
"States": [
  {
     //State 1…
  },
  {
     //State 2…
  },
  {
     //State 3…
  }
]
})
```

In "States" different states of the execution are described. For each state the following details have to be specified:

```
{
    "Prerequisites": {},
    "OrPrerequisites": {},
    "ErrorPrerequisites": {},
    "ErrorOrPrerequisites": {},
    "Outgoing": [
      {
        "ActivityID": "OutgoingActivityID1",
        "ActivityNum": "123",
        "AbstractionLayer": "Operation"
      },
      {
        "ActivityID": "OutgoingActivityID2",
        "ActivityNum": "234",
        "AbstractionLayer": "Task"
      }
    ]
  }
```

The activities listed on the "Outgoing" array are the activities that should execute when the described state is ready to run. Apart from ActivityID and AbstractionLayer, ActivityNum has to be specified. All the sub-activities contained in a certain activity must have a unique ActivityNum.

Prerequisites, OrPrerequisites, ErrorPrerequisites and ErrorOrPrerequisites are the arrays that contain the activities on which the current state depends. To add an activity in the state dependencies place the corresponding ActivityNum in the appropriate array. The state will be ready to run when one of the following happens:

- All the activities whose the ActivityNums are in the Prerequisites complete successfully AND all the activities whose the ActivityNums are in the ErrorPrerequisites result in an error.

- At least one activity whose the ActivityNum is in the OrPrerequisites completes successfully.

- At least one activity whose the ActivityNum is in the ErrorOrPrerequisites results in an error.

The ActivityNums 0 and 1 are reserved for the start and end nodes. To make a state start immediately after the activity starts, place "0" in state's prerequisites.

All activities need to have an end state. To mark a state as an end state, place an activity with ActivityNum "1" and ActivityID "End" in state's outgoing activities.

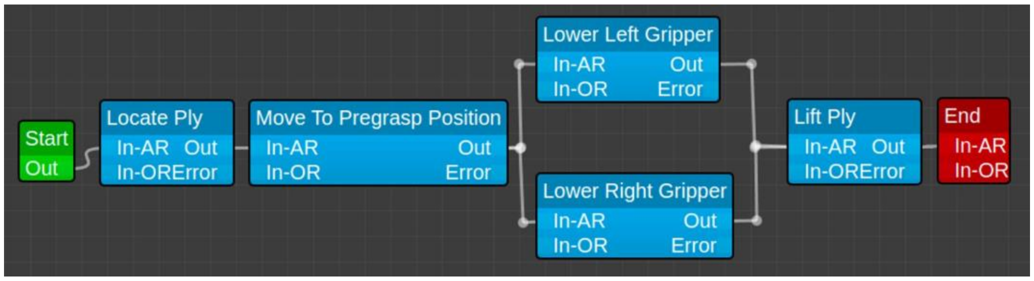


Figure 7 Locate & Lift Ply Example

For example, a JSON representation of the activity shown in Figure 7 is the following:

```
{
"OperationID": "Locate & Lift Ply",
"States": [
  {
    "Prerequisites": {
      "1": "6"
    },
    "OrPrerequisites": {},
    "ErrorPrerequisites": {},
    "ErrorOrPrerequisites": {},
    "Outgoing": [
      {
        "ActivityID": "End",
        "ActivityNum": "1"
      }
    ]
  },
  {
    "Prerequisites": {
      "1": "3"
    },
    "OrPrerequisites": {},
    "ErrorPrerequisites": {},
    "ErrorOrPrerequisites": {},
    "Outgoing": [
      {
        "ActivityID": "Move To Pregrasp Position",
        "ActivityNum": "2",
        "AbstractionLayer": "Action"
      }
    ]
  },
 {
     "Prerequisites": {
       "1": "0"
     },
     "OrPrerequisites": {},
     "ErrorPrerequisites": {},
     "ErrorOrPrerequisites": {},
     "Outgoing": [
       {
         "ActivityID": "Locate Ply",
         "ActivityNum": "3",
         "AbstractionLayer": "Action"
       }
     ]
   },
   {
     "Prerequisites": {
       "1": "2"
     },
     "OrPrerequisites": {},
     "ErrorPrerequisites": {},
     "ErrorOrPrerequisites": {},
     "Outgoing": [
       {
         "ActivityID": "Lower Left Gripper ",
         "ActivityNum": "4",
         "AbstractionLayer": "Action"
       },
       {
         "ActivityID": "Lower Right Gripper ",
         "ActivityNum": "5",
         "AbstractionLayer": "Action"
       }
     ]
   },
   {
     "Prerequisites": {
       "1": "4",
       "2": "5"
     },
     "OrPrerequisites": {},
     "ErrorPrerequisites": {},
     "ErrorOrPrerequisites": {},
     "Outgoing": [
       {
         "ActivityID": "Lift Ply",
         "ActivityNum": "6",
         "AbstractionLayer": "Action"
       }
     ]
   }
 ]
}
```

Execution Control

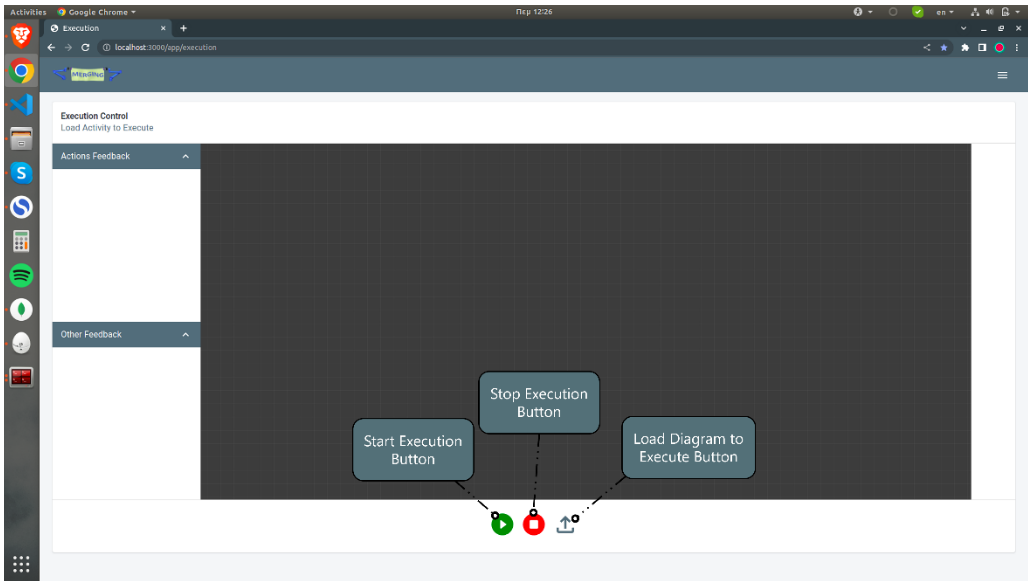


Figure 8 Execution Control Layout

Execution monitoring can be done in the "Execution Control" layout. By pressing the "Load Diagram to Execute" button, a layout appears from which it is possible to choose an activity to execute. After an activity is chosen, it is possible to execute it by pressing the play button. Nodes change color depending on the execution status: yellow are the nodes that are currently running, red are the nodes that returned an error and gray are the nodes that successfully finished execution. "Actions Feedback" on the left sidebar is low level feedback regarding the actions execution and whether execution completed successfully or not. "Other Feedback" contains feedback from the rest of the abstraction layers which is most useful when debugging schedules. The execution can be stopped by pressing the stop button: After pressing the stop button no new activities will be executed but the activities that have already started will continue running. In case it is important for an activity to stop immediately after the stop button is pressed, the activity must observe the /workcell_controller/stop_execution parameter of the parameter server. When this parameter has the value "true", the execution should stop.

Brokers

In case there is no way to control an entity by using the action types DOHC supports, a broker node should be implemented. Broker will map incompatible functionalities to workcell controller native functionalities. To create a broker follow the given template.

Common Mistakes

Error Handling

When executing a correcting routine after an error it is important to carefully choose the port from which the execution will return to the normal flow. In most of the cases that port is In-OR port.

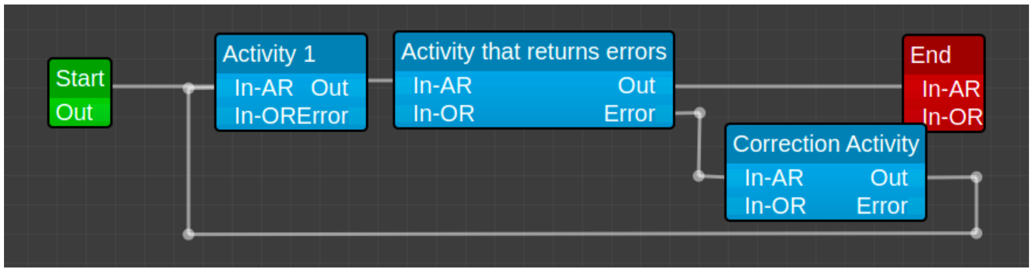


Figure 9 Common mistake while returning to normal flow after corrective activity.

The schedule in Figure 9 will not start running because Activity 1 has two prerequisites that must complete before it executes: The start button must be pressed and the Correction Activity must complete. Correction Activity is not possible to complete before Activity 1 completes.

Loops containing branches

Unwanted execution flows can happen with loops containing branches. The first time the loop in Figure 10 executes, " i++" will run only when "Activity B" and "Activity C" are completed. The second time, both prerequisites of activity "i++" will be considered completed and " i++" will execute when the first of the two prerequisites completes.

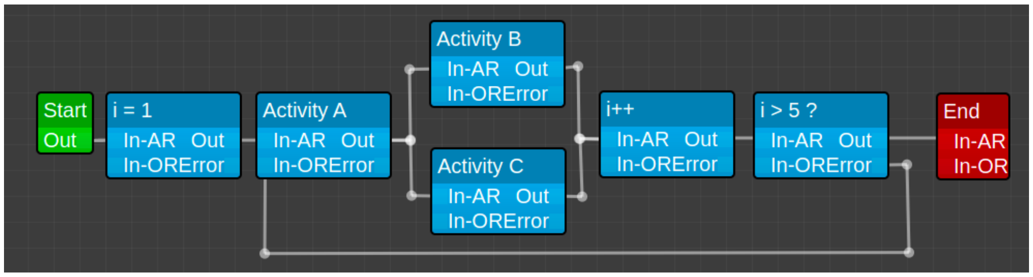


Figure 10 Loop containing branches common mistake.

In case this behavior is not wanted, " Activity B" and "Activity C" should be grouped together like shown in Figure 11.

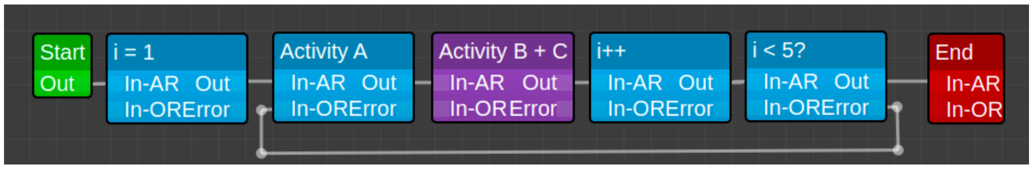


Figure 11 Loop containing branches correction

Loops and In-AR/In-OR

After a loop executes, it is important to return to the first activity of the loop using the correct port, which in most of the cases is the In-OR port.

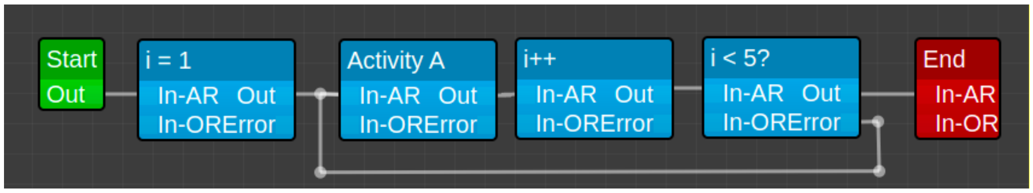


Figure 12 Common mistake with loops

The Activity A in Figure 12 will not start running because it has two dependencies: the activity " i=1" and the activity " i < 5 ?". Those activities are not complete the first time Activity A executes.