featX Guideline Documentation

This is the official guideline documentation for the featX language and framework. This guideline documentation assists end users to flexibly model and control features in robotic systems. It also assists in implementing modelled features. As a supporting artefact to the language, this document serves as a form of verification for the featX language and its capabilities.

Contents

1 Prerequisites and Dependencies

- Ubuntu 20.04.3 LTS and above
- ROS installation. This application was developed and tested with ROS Neotic 1.15.9.
- Python3 installation
- C++ 11 or higher
- ROS Pluginlib

2 Framework Components

- Domain Specific Language
- Modeller
- Configurator
- Model visualizer
- Syntax and Semantics Validator
- Feature Plugin Interface
- featX Shell

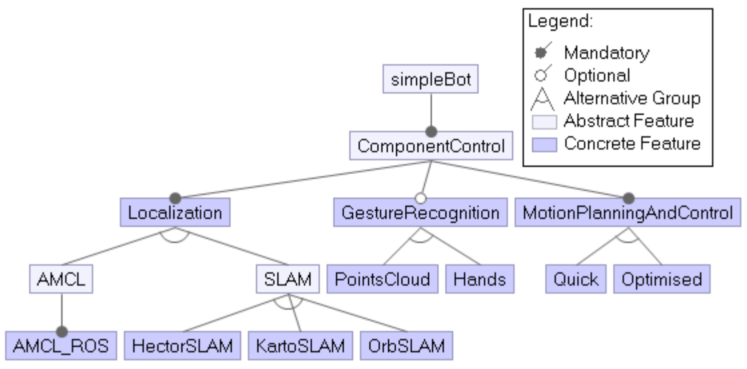
3 Setup and Installation of featX

- Clone featX from here into your catkin workspace.
- Go the featX workspace project directory and build the cloned project with the command catkin_make and press enter on your keyboard.

4 How To

This section gives a step-by-step walk-through of how featX can be used to instantiate a model or configuration, how it can be used to validate a model as well as the capability to link source code implementations with feature definitions. We also give a brief tutorial of our in-built featX console interface. Throughout our demonstrations in this guideline document, we use a sample model of a robotic system that is shown graphically in figure 1. This graphical representation of a simple robot model will also serve as our running example throughout this document.

Figure 1: Graphical Model of a Simple Robot



4.1 Instantiate a featX Model

- To create an instance of a model in featX, in the src folder of the cloned application, make a copy of the template folder provided and rename it to suit your product preference.
- In your new product instance folder, there is a model.json with an existing root feature, wrapped in a object.
- The model definition can be extended by appending sub features to the root feature. You can add nested feature objects according to your preferences.Note that each feature specification must have attributes such as id, name, constraints, group, isMandatory. The constraints attribute must contain sub attributes featureIncluded, featureExcluded, bindingTimeAllowed, bindingModeAllowed. Listing 1 shows a demonstrated example of an extended featX model instance, for the feature model captured in Figure 1.

- The value of each feature attribute in a model must conform to strict type systems defined within featX. A list of all the attributes together with their expected values is provided below in Table 1.

Table 1: Feature Attributes and Types

Attribute	Expected Value
id	Alphanumeric characters. No spaces allowed.
name	Alphanumeric characters. No spaces allowed.
constraints	Object containing attributes **featureIncluded**, **featureExcluded**, **bindingTimeAllowed**, **bindingModeAllowed**
featureIncluded	Array of existing feature "ids"
featureExcluded	Array of existing feature "ids"
bindingTimeAllowed	Early or Late or Any
bindingModeAllowed	Static or Dynamic or Any
group	XOR or OR
isMandatory	True or False

- To add a sub-feature to a defined feature, add the sub attribute to the feature and assign an array of sub-feature objects to it. However, the sub attribute is optional. Features that do not have sub-features can exist without a sub attribute.

Listing 1: featX Model Instance of SimpleRobot (Compressed)

```

{
    "id": "root_feature",
    "name": "SimpleBot",
    "group": "",
    "isMandatory": true,
    "isSelected": true,
    "sub": [
    {
        "id": "compcontrol",
        "name": "ComponentControl",
        "constraints": {
            "featuresIncluded": [],
            "featuresExcluded": [],
            "bindingTimeAllowed":"Early",
            "bindingModeAllowed":"Static"
        },
        "group": "OR",
        "isMandatory": true,
        "sub":[
        {
            "id": "localztn",
            "name": "Localisation",
            "constraints": {
                "featuresIncluded": [],
                "featuresExcluded": [],
                "bindingTimeAllowed":"Early",
                "bindingModeAllowed":"Static"
            },
            "group": "OR",
            "isMandatory": true,
            "sub":[...]
        },
        {
            "id": "gestrec",
            "name": "GestureRecognition",
            "constraints": {
                "featuresIncluded": [],
                "featuresExcluded": [],
                "bindingTimeAllowed":"Early",
                "bindingModeAllowed":"Static"
            },
            "group": "OR",
            "isMandatory": false,
            "sub":[...]
        },
        {...}
        ]
    }
    ]
}

```

Listing 2: featX Configuration Instance of SimpleRobot (Compressed)

```

{
        "properties": [
        {
                "id": "compcontrol",
                "props": {
                        "time":"Early",
                        "mode":"Static"
                }
        },
        {
                "id": "localztn",
                "props": {
                        "time":"Early",
                        "mode":"Static"
                }
        },
        {
                "id": "gestrec",
                "props": {
                        "time":"Early",
                        "mode":"Static"
                }
        },
        {
                "id": "motplannctrl",
                "props": {
                        "time":"Early",
                        "mode":"Static"
                }
        },
        {
                "id": "amcl",
                "props": {
                        "time":"Early",
                        "mode":"Static"
                }
        },
        {
                "id": "amclros",
                "props": {
                        "time":"Early",
                        "mode":"Static"
                }
        },
        {
                "id": "slam",
                "props": {
                        "time":"Early",
                        "mode":"Static"
                }
        },
        {...}
        ]
}
```

4.2 Instantiate a featX Configuration

- Again in your instance folder, there is a config.json with an empty properties array value.
- In the config.json file of your project, for every feature added in your model.json file, a corresponding configuration object must be added. For each feature configuration object, there must be an id attribute which references the id of a feature in your model.json file. The props attribute must contain time and mode attributes. Listing 2 shows a featX configuration translation of the features present in Figure 1.
- The values of each configuration attribute for your model must conform to strict type systems defined within featX. A list of all the configuration attributes together with their expected values is provided below in Table 2.

Table 2: Configuration Attributes and Types

Attribute	Expected Value
id	Alphanumeric characters. No spaces allowed.
props	Object containing attributes **time** and **mode**
time	Early or Late
mode	Static or Dynamic

4.3 Use the Standalone Model Validator Tool

The inbuilt featX model validator tool aids in validating models together with their corresponding configurations. To validate your model and its configuration, navigate to the /src/motivml directory and run the following command in the ROS console:

Listing 3: featX Validation Command

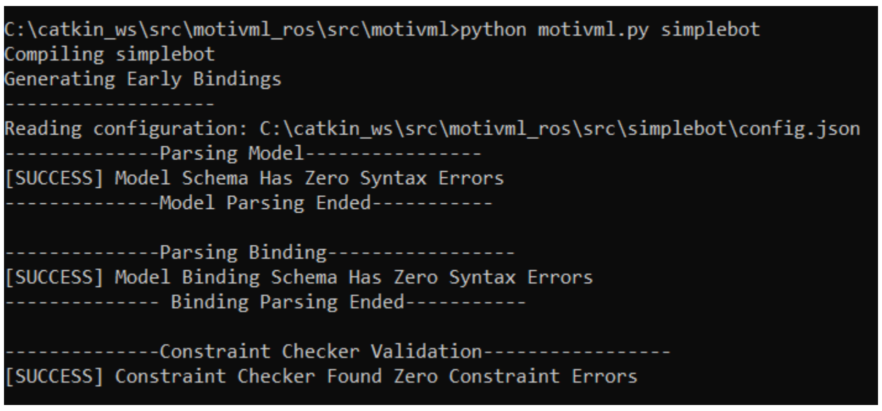
```

python validator.py <project_directory_name>

```

All models created in the featX language are validated on two distinct levels. i.e. syntax validation and semantic validation. Syntactical validation has to do with feature and configuration schemas being evaluated for errors, while semantic validation has to do with modelling and binding constraints evaluation.

To demonstrate this we evaluate our constructed model and configuration shown in Listing 1 and 2 and discuss the results accordingly. A careful observation of the console output generated from the validation indicates the presence of zero errors.

When our in-built schema and constraint checker identify an error in either a model or a configuration, an error message is outputted to the console indicating where the error is located and why the language has flagged it as an error.

Figure 2: Simplebot Validation Output



```
C:\catkin_ws\src\motivml_ros\src\motivml>python motivml.py simplebot
Compiling simplebot
Generating Early Bindings
-------------------
Reading configuration: C:\catkin_ws\src\motivml_ros\src\simplebot\config.json
-------------Parsing Model---------------
[SUCCESS] Model Schema Has Zero Syntax Errors
-------------Model Parsing Ended----------

-------------Parsing Binding----------------
[SUCCESS] Model Binding Schema Has Zero Syntax Errors
------------- Binding Parsing Ended----------

-------------Constraint Checker Validation----------------
[SUCCESS] Constraint Checker Found Zero Constraint Errors
```

4.4 Feature Class Implementation

Listing 4: Implementation of featX Feature Class

```
{
        namespace motivml_feature{

                enum BindingTimeAllowed{Early, Late, Any};
                enum BindingModeAllowed{Static, Dynamic, Any};

                class Feature{
                private:
                        std::string id;
                        std::string name;

                        std::vector<std::string> featuresIncluded;
                        std::vector<std::string> featuresExcluded;
                        BindingTimeAllowed bindingTimeAllowed{Early};
                        BindingModeAllowed bindingModeAllowed{Static};

                        std::string group;
                        bool isMandatory;

                public:
                        Feature(){};

                        //setters
                        void setId(std::string id){
                                id = id;
                        }

                        void setName(std::string name){
                                name = name;
                        }

                        void setBindingTimeAllowed(BindingTimeAllowed bindingTimeAllowed){
                                bindingTimeAllowed = bindingTimeAllowed;
                        }

                        void setBindingModeAllowed(BindingModeAllowed bindingModeAllowed){
                                bindingModeAllowed = bindingModeAllowed;
                        }

                        void setGroup(std::string group){
                                group = group;
                        }

                        void setIsMandatory(bool isMandatoory){
                                isMandatoory = isMandatoory;
                        }

                        void setFeaturesIncluded(std::vector<std::string> inclusions){
                                for(std::string &included : inclusions){
                                        featuresIncluded.push_back(included);
                                }
                        }

                        void setFeaturesExcluded(std::vector<std::string> exclusions){
                                for(std::string &excluded : exclusions){
                                        featuresExcluded.push_back(excluded);
                                }
                        }

                        //getters
                        std::string getName() const{
                                return name;
                        }

                        std::string getId() const{
                                return id;
                        }
                };
        };

}
```

The a capabilities of a featX feature are wrapped in a motivml_feature name space with a set of private attributes abstracted through a bunch of getters and setters. Within a feature, we can identify both descriptive and constraint attributes, implemented using c++ constructs.

4.5 Configuration Class Implementation

Listing 5: Implementation of featX Configuration Class

```
{
        class Configuration: public motivml_feature::Feature{
        private:
                std::string id;
                BindingTimes time;
                BindingModes mode;

        public:
                Configuration(){}
                Configuration(motivml_feature::Feature modelledFeature, BindingTimes timeSetting, BindingModes modeSetting)
                :id{modelledFeature.getId()}, time{timeSetting}, mode{modeSetting}{}
};
}
```

To decouple relevant features necessary for binding a feature, our time and mode binding attributes are encapsulated in a configuration class that inherits directly from our feature class. Just like we saw in the feature class structure, the binding time and bindimg mode date a hidden and abstracted with the help of getters and setters.

4.6 Implement a Statically Bound Feature

4.6.1 Static Early

- In the featx directory, create a static class in the namespace static_integration which inherits from the class public static_base::StaticInterface. NB: The feature id in your model must match your class name. For Example:

Listing 6: Sample Static Early Class

```

namespace static_integration{

        class MyStaticEarlyClass: public static_base::StaticInterface{
        private:
                std::vector<std::string> ex{"featureID_1"};
                std::vector<std::string> inc{"featureID_2"};

```

```
	public:
		Feature myStaticEarlyFeature;
		MyStaticEarlyClass(){
			myStaticEarlyFeature.setId("xxxx");
			myStaticEarlyFeature.setName("JohnDoe");
			myStaticEarlyFeature.setFeaturesExcluded(ex);
			myStaticEarlyFeature.setFeaturesIncluded(inc);
			myStaticEarlyFeature.setGroup("XOR");
			myStaticEarlyFeature.setIsMandatory(false);

			Configuration(myStaticEarlyFeature, Early, Static);
		};

		void executeFeature(){
			std::cout << myStaticEarlyFeature.getName() << "feature run successfully" << std::endl;
		};

	};

};

```

- Statically set the feature class attributes id to the same value as the id attribute in the featX model.
- Set configuration binding time and binding mode to Early and Static respectively.
- Set other class attributes to values of your choice that represent the feature you are implementing.
- For all included and excluded features in relation to the said feature, add their ids to a vector and pass them to the setter for either inclusion or exclusion constraints.
- Add the following preprocessor include above your class to include the static base interface class your class will be inheriting from:

Listing 7: Class Inclusion

```

#include "../include/motivml_ros/static_base.h"

```

- Define a static early feature constant at the top of plugin_feature_loader.cpp. For Example

Listing 8: Constant Definition

```

#define UNKOWNFEATURE

```

- In static_feature_loader.h, if the feature constant you created in the previous step is true, include your static class. See Listing 9 as an example

Listing 9: Including Static Early

```

#ifdef UNKOWNFEATURE
	#include "../../featx/MyStaticClass.h"
#endif

```

- Extent the static early conditional block of the callback_load_plugin_features method in plugin_feature_loader.cpp to run a method in your static early class.
- Run the catkin_make command to build the project.

4.6.2 Static Late

- In the featx directory, create a static class in the namespace static_integration which inherits from the class public static_base::StaticInterface. NB: The feature id in your model must match your class name. For Example:

Listing 10: Sample Static Late Class

```

namespace static_integration{

	class MyStaticLateClass: public static_base::StaticInterface{

	private:
		std::vector<std::string> ex{"featureID_1"};
```

```cpp
		std::vector<std::string> inc{"featureID_2"};

	public:
		Feature myStaticLateFeature;
		MyStaticLateClass(){
			myStaticLateFeature.setId("xxxx");
			myStaticLateFeature.setName("JohnDoe");
			myStaticLateFeature.setFeaturesExcluded(ex);
			myStaticLateFeature.setFeaturesIncluded(inc);
			myStaticLateFeature.setGroup("XOR");
			myStaticLateFeature.setIsMandatory(false);

			Configuration(myStaticLateFeature, Late, Static);
		};

		void executeFeature(){
			std::cout << myStaticLateFeature.getName() << "feature run successfully" << std::endl;
		};

	};

};

```

- Statically set the feature class attributes id to the same value as the id attribute in the featX model.
- Set configuration binding time and binding mode to Late and Static respectively.
- Set other class attributes to values of your choice that represent the feature you are implementing.
- For all included and excluded features in relation to the said feature, add their ids to a vector and pass them to the setter for either inclusion or exclusion constraints.
- Statically include the feature in the configuration by adding it to the motivml_plugins.h file. For Example

Listing 11: Sample Static Late Class

```cpp

#include "../../featx/MyStaticLateClass.h"

```

- To get the feature to be loaded at runtime, use pluginlib to export your class to the static interface. For example

Listing 12: Sample Static Late Class

```cpp

PLUGINLIB_EXPORT_CLASS(static_integration::MyStaticLateClass, static_base::StaticInterface)

```

- Provide meta-date about your static late feature to pluginlib, by registering the exported class in the motivml_plugins.xml file stating the class name and description of your class. For example

Listing 13: Sample Static Late Class

```xml

<class type="static_integration::MyStaticLateClass" base_class_type="static_base::StaticInterface">
	<description>Description of class goes here</description>
</class>

```

4.7 Implement a Dynamically Bound Feature

- In the featx directory, create a dynamic class in the namespace motivml_plugins which inherits from the class public plugin_base::PluginInterface. NB: The feature id in your model must match your class name. For Example:

Listing 14: Sample Dynamic Class

```cpp

namespace motivml_plugins{

	class MyDynamicClass: public plugin_base::PluginInterface{
	private:
		std::vector<std::string> ex{"featureID_1"};
		std::vector<std::string> inc{"featureID_2"};

```

```
        public:
                Feature myDynamicEarlyFeature;
                MyDynamicClass(){
                        myDynamicEarlyFeature.setId("xxxx");
                        myDynamicEarlyFeature.setName("JohnDoe");
                        myDynamicEarlyFeature.setFeaturesExcluded(ex);
                        myDynamicEarlyFeature.setFeaturesIncluded(inc);
                        myDynamicEarlyFeature.setGroup("XOR");
                        myDynamicEarlyFeature.setIsMandatory(false);

                        Configuration(myDynamicEarlyFeature, Early, Dynamic);
                };

                void executeFeature(){
                        std::cout << myDynamicEarlyFeature.getName() << "feature run successfully" << std::endl;
                };
        };

};

```

- Set configuration binding mode to Dynamic and binding time to either Late or Early.
- Set other class attributes to values of your choice that represent the feature you are implementing.
- For all included and excluded features in relation to the said feature, add their ids to a vector and pass them to the setter for either inclusion or exclusion constraints.
- Include Plugin: Include the feature in the configuration by adding it to the motivml_plugins.h file. For Example

Listing 15: Sample Static Late Class

```

#include "../../featx/MyDynamicClass.h"

```

- Encapsulate Feature As Plugin: In order to allow a class to be dynamically loaded, it must be marked as an exported class. This is done through the special macro PLUGINLIB_EXPORT_CLASS [1]. For example

Listing 16: Sample Dynamic Class Export

```

PLUGINLIB_EXPORT_CLASS(motivml_plugins::MyDynamicClass, plugin_base::PluginInterface)

```

- Add Plugin Description: The plugin description file is an XML file that serves to store all the important information about a plugin in a machine readable format. It contains information about the library the plugin is in, the name of the plugin, the type of the plugin, etc [1]. Ihe plugin description file (motivml_plugin.xml), would look something like this:

Listing 17: Sample Static Late Class

```

<class type="motivml_plugins::MyDynamicClass" base_class_type="plugin_base::PluginInterface">
        <description>Description of class goes here</description>
</class>

```

4.8 featX Console Interface

To add some interactivity to our variability modelling language, we have provided an in-built console interface for interacting with models and features through featX specific commands.

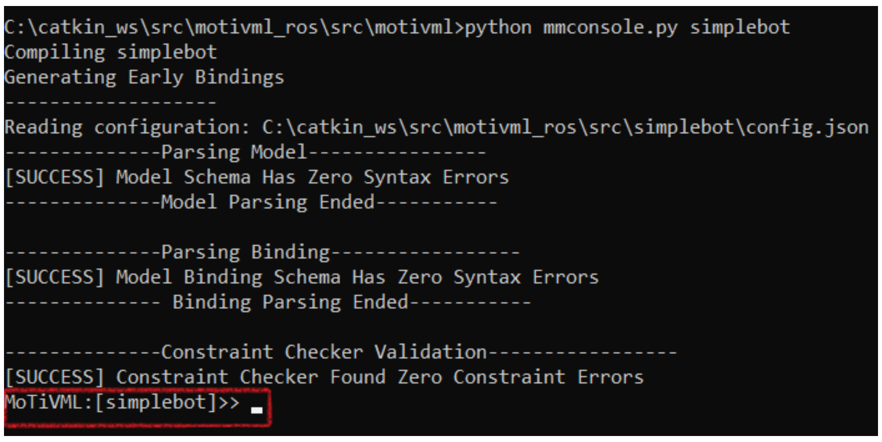
To launch the featX console interface, navigate to the /src/motivml directory and run the following command in Listing 18:

Listing 18: featX Console Launch Command

```

python mmconsole.py <project_directory_name>

```

When the above command is run, the model and configuration defined in the project_directory_name is compiled and validated, prior to the console interface being launched. If there are no errors detected, the console interface is then launched successfully.

Figure 3: Simplebot Console



```
C:\catkin_ws\src\motivml_ros\src\motivml>python mmconsole.py simplebot
Compiling simplebot
Generating Early Bindings
-------------------
Reading configuration: C:\catkin_ws\src\motivml_ros\src\simplebot\config.json
-------------Parsing Model---------------
[SUCCESS] Model Schema Has Zero Syntax Errors
-------------Model Parsing Ended----------

-------------Parsing Binding----------------
[SUCCESS] Model Binding Schema Has Zero Syntax Errors
------------- Binding Parsing Ended----------

-------------Constraint Checker Validation----------------
[SUCCESS] Constraint Checker Found Zero Constraint Errors
MoTiVML:[simplebot]>> _
```

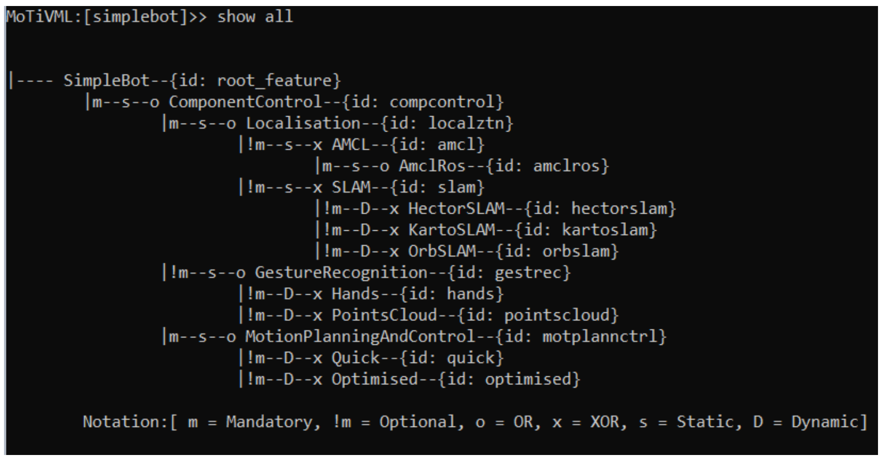
In Figure 3 we can observe the launched featX console. As highlighted in the figure above, the console prompt always bears the name of the project which was launched with the console. For this reason, all featX console commands that are run will be affiliated with the model highlighted in the console prompt.

4.9 featX Console Commands

- show < *parameter* >: The show command invokes the motivml model visualizer. The show command is used to display graphically a user constructed model. This command prints the hierarchical structure of a selected model while displaying details such as its mandatory status, group and binding mode. The show command takes a single parameter. This parameter can either be all or config.

  The show all command only visualizes all selected features that exist in a model tree. By default all features are selected when instantiated. Thus, this command shows every single feature in the model tree.

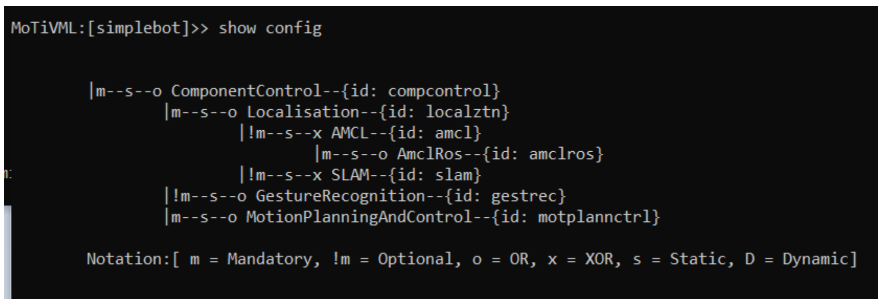
  Figure 4: show all Command

```
MoTiVML:[simplebot]>> show all


|---- SimpleBot--{id: root_feature}
        |m--s--o ComponentControl--{id: compcontrol}
                |m--s--o Localisation--{id: localztn}
                        |!m--s--x AMCL--{id: amcl}
                                |m--s--o AmclRos--{id: amclros}
                        |!m--s--x SLAM--{id: slam}
                                |!m--D--x HectorSLAM--{id: hectorslam}
                                |!m--D--x KartoSLAM--{id: kartoslam}
                                |!m--D--x OrbSLAM--{id: orbslam}
                |!m--s--o GestureRecognition--{id: gestrec}
                        |!m--D--x Hands--{id: hands}
                        |!m--D--x PointsCloud--{id: pointscloud}
                |m--s--o MotionPlanningAndControl--{id: motplannctrl}
                        |!m--D--x Quick--{id: quick}
                        |!m--D--x Optimised--{id: optimised}

        Notation:[ m = Mandatory, !m = Optional, o = OR, x = XOR, s = Static, D = Dynamic]
```

  The show config command only visualizes currently selected or bound features. That is, unselected or unbound features in the model are excluded from the visualizer output.

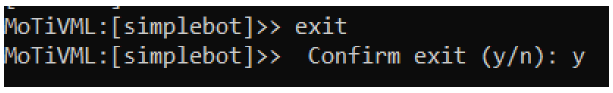
  Figure 5: show config Command

```
MoTiVML:[simplebot]>> show config


        |m--s--o ComponentControl--{id: compcontrol}
                |m--s--o Localisation--{id: localztn}
                        |!m--s--x AMCL--{id: amcl}
                                |m--s--o AmclRos--{id: amclros}
                        |!m--s--x SLAM--{id: slam}
                |!m--s--o GestureRecognition--{id: gestrec}
                |m--s--o MotionPlanningAndControl--{id: motplannctrl}

        Notation:[ m = Mandatory, !m = Optional, o = OR, x = XOR, s = Static, D = Dynamic]
```

- exit: The exit command shuts down the console interface and returns back to the original ROS console. The exit command also asks for a yes or no confirmation before proceeding. Figure 6 shows an exit command execution and output.

  Figure 6: Exit Command

```
MoTiVML:[simplebot]>> exit
MoTiVML:[simplebot]>>  Confirm exit (y/n): y
```

- load $< featureid >$: The load command attempts to add a feature to the existing configuration. Once the feature is added successfully to the configuration, the feature is executed to show its existence.
- unload $< featureid >$:The unload command attempts to remove a feature from the existing configuration. Once the feature is removed successfully from the configuration, an output message is sent to the console. Like wise, if the unload attempt is unsuccessful, a prompt is printed in the console.
- dump: The dump configuration command the ids of all features that have been bound in the configuration, in the form a categorised list as a console output.

5 Launching a Configuration in the Framework

After modelling your robot with featX and implementing all the modelled features, the robot configuration can be launched to simulate loading and unloading of features. To launch the configuration, the following steps must be performed in the exact sequence presented.

The output of all commands executed in the console interface can be viewed in the plugin manager console.

1. Launch Parameter Server: To launch the server parameters responsible for dynamic binding, run the following command in your terminal/console. Using roslaunch, parameters for each binding combination are created on the ros parameter service

Listing 19: Application ROS Parameters Launch Command

```

roslaunch motivml_ros motivml.launch

```

2. Launch User Interface: The user interface needs to be launched next because in its start up sequence it generated all static features and binds them to the the already initialised ROS parameters. To this, run the following command in a separate console/terminal. To tun the command successfully, navigate to the "/src/motivml_ros/src/motivml" directory of the project.

Listing 20: Application Cole Interface Launch Command

```

python mmconsole.py <project_name>

```

3. Launch Plugin Manager Interface: The plugin interface represents the live version of the robot configuration defined in your featX model instance. In here users can display all statically and dynamically bound features. Users can also observe the output of feature load and unload attempts. To launch this interface, run the following ROS command.

Listing 21: Application Cole Interface Launch Command

```

rosrun motivml_ros plugin_feature_loader

```

References

[1] "ROS Pluginlib Documentation," [Online] Available: http://wiki.ros.org/pluginlib.