URCap Program Node Configuration

Universal Robots A/S

Version 1.15.0

Abstract

As of URCap API version 1.2, child program nodes in a URCap program node contribution's sub tree can be configured. This can be useful if a URCap needs to provide a partially or fully configured template for the end user. It is also possible to easily traverse the sub tree to extract configurations of the nodes in it. This document explains how to work with the configurations and traversal of sub trees.

Contents

1 Introduction

In URCap API version 1.1 the concept of template program nodes was introduced. This allowed a URCap program node contribution to add a sub tree when inserted in/into the program tree.

With version 1.2 of the URCap API, it is now possible to apply and read configurations of the child program nodes in a URCap program node contribution's sub tree. This can be useful to set up a fully or partially configured sub tree when a URCap program node is inserted.

2 Creating a Configuration

Configurations are created using a factory specific to a given built-in PolyScope program node type. For many built-in nodes in the URCap API, it is possible to get a configuration factory by calling the `getConfigFactory()` method on an instance of a built-in node. This will return an instance that can be used to create configurations applicable for built-in nodes of that specific type. Depending on the complexity of the configuration, the factory can have one or more methods to create parts of the final configuration. In this case, all parts has to be created and then combined into a final configuration.

All configurations are immutable, meaning it is not possible to change them once created. If a single value in a configuration needs to be changed, a new configuration has to be created using all values from the existing configuration except for the one that should be modified.

Configurations should in no way be cached (e.g. in member variables), but always retrieved when needed, as the configuration could have changed.

Many of the parameters needed to create a configuration are simple value objects created using the `SimpleValueFactory` interface. This factory can be accessed using the `getSimpleValueFactory()` method in the `ValueFactoryProvider` interface.

When specifying configuration parameters which have validation in PolyScope, an error handler must be provided to the factory's specific create method. This error handler is used in case the specified value is outside the valid range. It is possible to use the `ErrorHandler.AUTO_CORRECT` constant (or `null`) as error handler. This will automatically correct the value, if it is outside the allowed range. In most cases, the value will be adjusted to the nearest valid value. It is also possible to implement a custom error handler, if a different action should be taken when a parameter is outside the valid range.

Note that some simple built-in program nodes (e.g. If and Comment nodes) do not use the described configuration approach, but instead provide simple set and get methods in the actual program node API interface.

2.1 Applying a Configuration

Applying a configuration is simply a call to the `setConfig()` method on a program node instance. In some cases, `setConfig()` can throw a checked exception, which needs to be handled in the URCap code.

For instance for the `SetNode` interface, the `setConfig()` method can throw an `InvalidDomainException`, if trying to set a voltage value on an analog output which is configured to use (electric) current domain in the currently loaded installation (or vice versa). In this case, the developer must implement some sort of error handling by either using a different output, informing the end user of the situation or skipping configuration of the node.

3 Traversal of Sub Tree

When navigating a sub tree, the `TreeNode` interface is the starting point. From here it is possible to use the traditional approach of using a list to iterate through the children, their children and so on. For each child, a check of its type is performed and an appropriate cast has to be performed before working with the child.

It is also possible to use a more sophisticated and elegant traversal approach by utilizing the `ProgramNodeVisitor` class and the `traverse()` method in the `TreeNode` interface. This alternative approach can be utilized to visit all child nodes or all child nodes of specific types.

The `traverse()` method iterates all children in a depth-first fashion (corresponding to top-down when looking at the program tree in PolyScope). For each node encountered, a callback to the `ProgramNodeVisitor` is made. In the following section this abstract class is described in more detail.

3.1 ProgramNodeVisitor

The `ProgramNodeVisitor` is an abstract class that must be extended and have one or more `visit()` methods overridden. All `visit()` methods in the base class have empty implementations, meaning that nothing will happen if the method is not overridden. This way it is sufficient to only implement the `visit()` methods necessary in the given context depending on which node types should be visited.

The parameters of each `visit()` method are the specific program node, the sibling index and the depth of the program node. This means that the program node (passed) is ready to be worked with as it is already the appropriate type (no casting required). The sibling index is a zero-based index for the program node position among the siblings at this depth. Lastly the depth parameter is depth of the program node counting from the `TreeNode` from which the `traverse()` method was called. Immediate children of the calling node will have depth 1. The `index` and `depth` parameters can be used to do extra filtering to find specific nodes.

Below in Listing 1 is an example demonstrating how to visit all Set and Wait nodes. The `traverse()` method uses an anonymous implementation of the abstract class `ProgramNodeVisitor` that overrides `visit(SetNode,...)` and `visit(WaitNode,...)` to retrieve configurations from these types of program nodes.

Listing 1: Example using the `ProgramNodeVisitor` to visit all Set and Wait nodes

```
1  TreeNode rootTreeNode = URCapAPIFacade.getURCapAPI().getProgramModel().
       getRootTreeNode(this);
2  rootTreeNode.traverse(new ProgramNodeVisitor() {
3    @Override
4    public void visit(SetNode programNode, int index, int depth) {
5      SetNodeConfig config = programNode.getConfig();
6    }
7
8    @Override
9    public void visit(WaitNode programNode, int index, int depth) {
10     WaitNodeConfig config = programNode.getConfig();
11   }
12 });
```

The URCap API also contains the `URCapProgramNodeInterfaceVisitor<URCAP_PROGRAM_NODE_INTERFACE>` abstract class which extends `ProgramNodeVisitor`. This class can be used to visit URCap program nodes that implement a custom API interface (as well as visit any other node type). Only one type of custom interface can specified for the visitor. Custom URCap node API interfaces

are described in section 5. URCap Program Node proprietary API. The method `visitURCapAs(URCAP_PROGRAM_NODE_INTERFACE, int, int)` will pass the encountered URCap program node already cast to the provided interface type, so that the node is ready to work with.

4 Extracting a Configuration

Many built-in program node interfaces have a `getConfig()` method serving as the entry point for accessing the node's configuration. This method returns a base configuration interface with an associated `enum` type. Based on this type, the configuration instance can be cast to a specific type to retrieve more detail.

This is a general pattern that applies not only to the configuration itself, but also to some of the individual parts of the configuration, in cases where the configuration or a setting/option has several different types or "states". These are typically represented in PolyScope by radio buttons, drop-down lists or checkboxes in the screens for configuring a program node.

4.1 Casting of Configurations

Listing 2 shows an extension of the implementation of the `visit(SetNode,...)` method from the visitor example in Listing 1. It extracts the configuration of a Set node, and if the node is configured to apply an electric current to an analog output (configured for current domain), it will output to the console the specific settings for that.

This includes checking what output signal selection has been made, which follows the same pattern of first determining the selection type, and in case the selection is an analog output (as opposed to no selection), the selection is cast to the appropriate type. In this example, the type is `AnalogOutputSelection`. Once cast, the name of the analog output is output to the console. Finally the value of the electric current is output to the console.

Listing 2: Extracting configurations of Set nodes

```
1  @Override
2  public void visit(SetNode programNode, int index, int depth) {
3    SetNodeConfig config = programNode.getConfig();
4    if (SetNodeConfig.ConfigType.ANALOG_OUTPUT_CURRENT.equals(config.
         getConfigType())) {
5      AnalogOutputCurrentSetNodeConfig analogConfig = (
           AnalogOutputCurrentSetNodeConfig) config;
6
7      SetNodeAnalogIOSelection outputSelection = analogConfig.getOutputSelection
           ();
8      if (SetNodeAnalogIOSelection.SelectionType.ANALOG_OUTPUT.equals(
           outputSelection.getType())) {
9        AnalogIO output = ((AnalogOutputSelection) outputSelection).getOutput();
10       System.out.println(output.getName());
11     } else {
12       System.out.println("No output selected");
13     }
14     System.out.println(analogConfig.getCurrent().getAs(Current.Unit.mA) + " mA
           ");
15   }
16 }
```

5 URCap Program Node proprietary API

It is also possible to access state/configuration of another URCap program node as well as configure or interact with it, if that URCap's program node contribution implements a specific custom interface (API). Such an interface can be accessed using the `getAs(Class<T>)` method in the `URCapProgramNode` interface. This returns the program node contribution already cast to the given interface.

For an example of this, see the code snippet in Listing 3 where the parent URCap program node (A) needs to call methods on the child URCap program node (B). In this example, the custom API is used when the child URCap program node is created, but it could also be used with an existing program node found by navigating to it. A call is also made to the method `canGetAs(Class<?>)` in `URCapProgramNode` in order to check, if the node implements the expected custom interface.

Listing 3: Example of providing and using a custom URCap node API

```
1  // CustomAPI provided by URCapBProgramNode
2  interface CustomAPI {
3    Setting getSetting();
4    void setSetting(Setting setting);
5
6    boolean isOptionEnabled();
7    void reset();
8  }
9
10 // URCapBProgramNodeService is the ProgramNodeService responsible for creating
       URCapBProgramNode instances
11
12 // Child program node (B)
13 public class URCapBProgramNode implements ProgramNodeContribution, CustomAPI
       ...
14
15 //Parent program node (A)
16 URCapAPI api = URCapAPIFacade.getURCapAPI();
17 ProgramNodeFactory factory = api.getProgramModel().getProgramNodeFactory();
18 URCapProgramNode child = factory.createURCapProgramNode(
       URCapBProgramNodeService.class);
19
20 if (child.canGetAs(CustomAPI.class)) {
21   CustomAPI childAs = child.getAs(CustomAPI.class);
22   childAs.setSetting(new Setting());
23 }
```

5.1 Giving Access to 3rd Party URCaps

If two URCap program node contributions in different URCaps need to communicate using the above mentioned technique, the proprietary API must be made available to the consuming part. This is done in the `maven-bundle-plugin` element of the project `pom.xml` file. The custom API should only be made available, if it is strictly necessary, since the API will be exposed to all installed URCaps.

The URCap defining the API must add the package in which the API is placed under the `<Export-Package>` element of the configuration. See the snippet in Listing 4 for an example:

Listing 4: Exposing a custom API through export in the `pom.xml` file

```
1  <plugin>
2      <groupId>org.apache.felix</groupId>
3      <artifactId>maven-bundle-plugin</artifactId>
```

```
4       ...
5       <configuration>
6           <instructions>
7               ...
8               <Import-Package>
9                   com.ur.urcap.api*;version="[1.0.0,2.0.0)",
10                  *
11              </Import-Package>
12              <Export-Package>
13                  com.mycompany.urcap.proprietary.api
14              </Export-Package>
15          </instructions>
16      </configuration>
17  </plugin>
```

The <Export-Package> element might not exist. In this case, add it next to the <Import-Package> element.

The consuming client must import the newly exported package in order to resolve its dependencies. This is done by adding the package explicitly or using an asterisk as wildcard to import anything needed. An example of the explicit import can be seen in Listing 5.

Listing 5: Importing a custom API in the `pom.xml` file

```
1  <Import-Package>
2    com.ur.urcap.api*;version="[1.0.0,2.0.0)",
3    com.mycompany.urcap.proprietary.api;version="[1.0.0,2.0.0)",
4    *
5  </Import-Package>
```

The exported API will have the version number of the exporting URCap and care must be taken to ensure compatibility between the exporting and importing parties. A good rule of thumb is to use semantic versioning and explicitly state the valid range for the imported API as exemplified (see also `http://semver.org`).

6 When to use Native Nodes

The built-in program nodes offer a lot of functionality and can be preconfigured to suit the URCap program node's needs. However, the end user can change the configuration as he sees fit.

If that is undesirable from the URCap program node's point of view, the part that needs locking should be handled by the URCap program node. Either directly in its own script code generation, or if the action needs to happen at a specific point in the middle of its children, a special non user insertable URCap program node could be used. This node can be configured by the parent URCap program node as demonstrated previously and cannot be changed by the end user. If the sequence furthermore is locked, it cannot be deleted, moved etc. either.

A very simple example could be the situation, where it is required to wait a specific fixed amount of time, in which case the Wait program node should not be used, but instead a call to the script function `sleep()` should be included in a URCap node's script generation. Another example could be, if very specific parameters are required for movements and the behavior of the URCap node would break, if these were modified by the end user. This should be implemented in script code and not by using Move and Waypoint program nodes.