Tool for ROS code generation

Danying Hu and Yangmin Li and Mohammad Haghighipanah and Blake Hannaford

Abstract— **(Let's cooperate on this google doc and we can convert to Latex just before submission)**

TBD: Let?s come up with a cool and short name for the tool?. ideas?
ros-helper??
ros_tool??
ros-o-matic??

we can do a global search-replace with [ros_tool] → new-name

I. INTRODUCTION

A. ROS

ROS[**?**] is a widely adopted middleware package for robotics which greatly facilitates cooperation among software modules in a robotic system. Unfortunately, ROS suffers from a difficult learning curve, especially for programmers who are not well versed in build systems such as CMake. ROS has available an excellent set of tutorials[**?**] which quickly expose the new user to the basic functions. However, the examples in the tutorials can be hard for new users to generalize into new applications. Modifying the tutorial examples into desired applications which function independently of the tutorial packages is not straightforward.

One simple approach to making the learning curve easier is to create template, "starter", files, which can be customized by the new user. However, ROS has a large number of capabilities and supports multiple languages. Thus there are many needed templates. Specific attributes of ROS nodes which generate different elements in the node code are

- Language options: C++ and Python (other languages will not be considered here but ROS support for Java and Lisp exists)
- Communication tasks: which can be one or more of Messages, Services, and Actions
- Direction of communication: inbound and outbound communication is referred to by names specific to the task. For example Subscribe and Publish describe input and output for messages.

The total number of templates is thus 12 (assuming only one ROS feature is used in the application). A template-based approach would also require the user to create build configuration files including ROS and node specific manifest.xml, CMakeLists.txt, and package.xml depending on which of the two ROS build systems (rosbuild or catkin) will be used. Counting the two build-system options, the total number of permuations becomes 24! Many ROS applications (even at the beginner level) require more than one of the above combinations. For example, a node may publish a message and call a service. Thus a set of fixed templates is of limited practical use.

For each node which instantiates one or more particular task/language combinations, a ROS application must have

- Includes/imports for dependencies
- Initialization of the node
- Instantiation of classes for task units (message variables, service requests and responses, etc.)
- Callbacks (for inbound communications)
- Example invocations of some classes (such as `roscpp::publish() or rospy.publish()`)
- Proper argument lists for the class invocations and methods
- Correctly configured CMakeLists.txt and package.xml or manifest.xml files.

II. APPROACH

A. Software Description

ros_tool assumes that the user has a properly configured ROS workspace already created, that roscore is running, and that a package has been created and initialized in the current workspace. ros_tool is written in Python and contains classes which describe and process the ROS package, and a new ROS node, as well as helper functions to explore the package file tree and determine any existing .msg and .srv files.

The first part of ros_tool execution collects information from the user about the desired application template including

- path to ROS workspace
- package name
- name of existing messages (.msg) or service (.srv) (if any)

*This work was supported by NSF Grant xxxxxxxxxxxx via subcontract from Stanford University

Biorobotics Laboratory, Department of Electrical Engineering, University of Washington, Seattle WA 98195-2500

Filename	Template Target
pyt2_template.py	Python nodes
cpp_template.cpp	C++ nodes
CMakeListTemplate.txt	CMakeList.txt file
MakefileTemplate	
manifestTemplate.xml	
msgTemplate.msg	.msg files
srvTemplate.srv	.srv files

TABLE I

TEMPLATE FILES USED BY ROS_TOOL TO GENERATE USER NODE AND ASSOCIATED BUILD FILES.

- name of existing ros nodes in the package
- Desired tasks (Message, Service, or Action) and their directions (e.g. Publish or Subscribe)

There are three ways this information is collected. First, the user can edit a file (param_node.py) which initializes values for any internal ros_tool variable. The param_node.py file is executed (via `exec()`) by the main Python script. The user can thus initialize any value using any valid Python syntax, but basic assignment statements are sufficient for most purposes. For example, the following lines contained in param_node.py

```
# path to your ROS workspace
rws = '/home/blake/Projects/Ros2/RosNodeGen/'
```

initialize the variable "rws" which contains the ROS workspace path.

A second mode of data collection is to query the user. This is done with "rawinput()" statements in the command line. For example one such input statement prompts the user to enter the name of the node as:

```
Enter your new node name: [test_node]:
```

The default value, in this case "test_node", is accepted if the user hits enter without typing a new value.

Finally, the ros_tool gets some information by exploring the file system of the selected package. We use the `rospkg` library (http://www.ros.org/reps/rep-0114.html) to look for msg and service descriptors (.msg and .srv files) which may exist in the package. Information collected by the three methods is populated into the package and node objects.

One of the most important user inputs is to specify the language (C++ and Python only in the current version). Language selection determines the initial template file as well as the templates for the included statements. As each message or service is fully specified, appropriate statements in the selected language are generated by substituting their data tags and then appended to the section tag statement lists. After the end of user input, the output files are generated by completing substitution of any remaining data tags and writing the output files.

B. Generating Output

ros_tool uses several template files containing tags which will be filled in with the user's information. The template files are listed in Table I.

An example tag is "`$PKG$`" which will be replaced by the package name.

The ros_tool contains a large number of tag/value pairs which are populated from the node and package classes and used to create the required output files from the templates. Tags are further divided into two levels, data tags and section tags. Data tags contain simple values such as the name of a ROS message. Section tags contain entire sections of generated code such as statements to instantiate several ROS messages to which the application will subscribe or all the user-specific imports required in a Python node. Section tags may contain data tags.

When generating a new node with ros_tool, the output generates files which are in the right locations and are ready for building with the selected ROS build tool. All configurations including CMakeList.txt and manifest.xml/package.xml have been automatically updated. Template files for build files are selected based on which of the two build systems are selected.

Another issue which generates multiple branches in the logic is the potential use of existing message/service/action templates vs. creating custom ones. As a user prepares an application template, there are several possibilities (although they will be illustrated here with messages they apply to services and actions as well):

1) User wishes to use an existing message from another package (e.g. std_msgs)
2) User wishes to use a custom message in the current package, .msg file already exists
3) User wishes to use a custom message in the current package, but has not yet created the .msg file.

ros_tool supports all three of these cases, prompting the user for necessary information to build a new .msg file if necesssary.

III. EXAMPLE

In the following example, we create a new Python node in the "test2" package called "icra_2015_node". This new node will publish a message (with an existing .msg file in the test2 package) and will call a service ("example_serv_2015") as a client. The user dialog is reproduced in Figure 1:

Because this is a Python node, ros_tool selects the pyt2_template.py template file. The before and after versions of that file are reproduced in Figure 2.

IV. DISCUSSION AND FUTURE WORK

The contribution of this work is a new software package to streamline the process of creating a new ROS node for robotics control. The package automatically generates code for a new node source file in either C++ or Python. It also creates build-configuration files for either rosbuild or catkin build systems and .msg or .srv files if necessary

ros_tool source code and template files are available on github: *****URL******

```
Welcome to ROS node generator  *****
Please answer a few questions about your new node:
Enter your package name: test2
start to generate Makefile
Enter your new node name: [test2_node]: icra_2015_node
Enter your language: [Python or C++]: Py
Do you want to add a message? (y for yes, n/CR for no) y
[P]ublish or [S]ubscribe?: Pub
Enter the package that contains your message [test2]
getting path for package: [test2]
Found package path: /home/blake/Projects/Ros2/RosNodeGen/src/test2
Find the following messages in the package /home/blake/Projects/Ros2/RosNodeGen/src/test2/msg/:
1: String.msg
Select message by number: 1
Enter the topic of your message: default [String_topic]icra_2015_tpc
getting path for package: [test2]
Found package path: /home/blake/Projects/Ros2/RosNodeGen/src/test2
this is a comment line
this is a header line
this is an empty line
Do you want to add a message? (y for yes, n/CR for no)
Do you want to add a service? (y for yes, n/CR for no) y
[C]lient or [S]erver?: Cli
Enter the package that contains your service, default package: [ test2 ]
getting path for package: [test2]
Found package path: /home/blake/Projects/Ros2/RosNodeGen/src/test2
Find the following services in the package /home/blake/Projects/Ros2/RosNodeGen/src/test2/srv/:
1: bh_service.srv
Select service by number: 1
Enter the name of your service: default [bh_service_name]example_serv_2015
getting path for package: [test2]
Found package path: /home/blake/Projects/Ros2/RosNodeGen/src/test2
Do you want to add a service? (y for yes, n/CR for no)
start to generate the source file for node icra_2015_node in language Python
start to update manifest.xml
start to update CMakeList.txt
```

Fig. 1. Terminal dialog in which ros_tool creates a node in Python language which can publish to the topic "icra_2015_tpc" and invoke the service "bh_service.srv" as a client.

A. Limitations

As with all software, the ros_tool has limitations: Much of the user input is based on a clunky command line interface. GUI support (through a package like tkinter/ttk) should be implemented. Python indentation needs adjustment by the user and should be automatically corrected as tags are filled in with values. Configuration files such as CMakeLists.txt are created from the generic template files and so if a package contains existing nodes, the old CMakeLists.txt file must be saved and manually merged with the new one. Variable names for new custom messages and services are auto-generated and thus not descriptive for the applications semantics.

B. Future Work

Our current priorities for future improvements to ros_tool are

- Generate correct indentation on Python output
- Extend functionality to ROS actions
- Adapt the build-file output system to modify existing CMakeList.txt instead of creating from scratch.

The ros_tool code is open source (LGPL) and we encourage users to identify and fix bugs and limitations to ros_tool and participate in a community effort to simplify ROS development.

```
#!/usr/bin/env python
"""nav.py, A minimal ROS node in Python.
"""
### This file is generated using ros node template, feel free to edit
### Please finish the TODO part
#  Ros imports
import rospy
import roslib; roslib.load_manifest('$PKG$')

## message import format:
##from MY_PACKAGE_NAME.msg import MY_MESSAGE_NAME
#from   $PKG$.msg import  $MSG$

$IMPs$

##############################################################
##   Message Callbacks
$MCBs$

##############################################################
##  Service Callbacks
$SCBs$

##############################################################
# Main Program Code
# This is run once when the node is brought up (roslaunch or rosrun)
if __name__ == '__main__':
  print "Hello world"
# get the node started first so that logging works from the get-go
  rospy.init_node("$RNN$")
##############################################################
##  Service Advertisers
  $SADs$

##############################################################
##  Message Subscribers
  $SUBs$

##############################################################
##  Message Publishers
  $PUBs$

##############################################################
##  Service Client Inits
  $SCIs$

############## Message Object for Publisher ####################
  $MOBs$
  $MSVs$

############## Service Object for client ####################
  $SROs$

##############################################################
##  Main loop start
  while not rospy.is_shutdown():
##############################################################
##  Message Publications
    $PBLs$
##############################################################
##  Service Client Calls
    $SVVs$
    $SCCs$
    rospy.loginfo("$RNN$: main loop")
    rospy.sleep(2)
##############################################################
#
# end of main wile loop
```

```
#!/usr/bin/env python
"""nav.py, A minimal ROS node in Python.
"""
### This file is generated using ros node template, feel free to edit
### Please finish the TODO part
#  Ros imports
import rospy
import roslib; roslib.load_manifest('test2')

## message import format:
##from MY_PACKAGE_NAME.msg import MY_MESSAGE_NAME
#from   test2.msg import  String
from test2.msg import String
from test2.srv import bh_service

##############################################################
##   Message Callbacks

##############################################################
##  Service Callbacks

##############################################################
# Main Program Code
# This is run once when the node is brought up (roslaunch or rosrun)
if __name__ == '__main__':
  print "Hello world"
# get the node started first so that logging works from the get-go
  rospy.init_node("icra_2015_node")
  rospy.loginfo("Started template python node: icra_2015_node.")
##############################################################

##############################################################
##  Message Subscribers

##############################################################
##  Message Publishers
  String_pub1  = rospy.Publisher("icra_2015_tpc",String)

##############################################################
##  Service Client Inits
  rospy.wait_for_service("example_serv_2015")
  bh_service_cli1 = rospy.ServiceProxy("example_serv_2015", bh_service)

############## Message Object for Publisher ####################
  String_obj1 = String()

############## Service Object for client ####################
  bh_service_obj1 = bh_service()


##############################################################
##  Main loop start
  while not rospy.is_shutdown():
##############################################################
##  Message Publications
    String_pub1.publish(String_obj1)

##############################################################
##  Service Client Calls
    bh_service_obj1.rng.py = 0
    bh_service_obj1.rng.py = 0
    try:
      result = bh_service_cli1(bh_service_obj1.rng.py)
      print(result)
    except:print("Something wrong with client call example_serv_2015")
    rospy.loginfo("icra_2015_node: main loop")
    rospy.sleep(2)
##############################################################
#
# end of main wile loop
```

Fig. 2. Python template file (left) and completed node file (right). Boxes and arrows illustrate one case of tag substitution.