



Problem

- ROS is a framework for programming robots
- The first version, ROS1, is not compatible with ROS2
- Want to automatically upgrade ROS1 to ROS2

Motivation

- Difficult to update code (e.g. Python 2.7 vs Python 3)
- Coding robots is more interesting and fun than refactoring
- ROS1 has major security vulnerabilities

```
1 void chatterCallback(const std_msgs::String::ConstPtr&
2   msg){
3   // ...
4 }
5 int main(int argc, char **argv){
6   ros::init(argc, argv, "listener_node");
7   ros::NodeHandle n;
8   ros::Subscriber sub = n.subscribe("chatter", 1000,
9     chatterCallback);
10  ros::spin();
11  return 0;
12 }
```

Figure 1: Original C++ code for a “listener” node, using ROS1 APIs.

```
1 void chatterCallback(const std_msgs::String::ConstPtr&
2   msg){
3   // ...
4 }
5 int main(int argc, char **argv) {
6   std::string node_name = "listener_node";
7   ??? // ros::init
8   ??? // ros::NodeHandle
9   std::string topic_name = "chatter";
10  int queue_size = 1000;
11  ??? // ros::NodeHandle::subscribe
12  ??? // ros::spin
13  return 0;
14 }
```

Figure 2: The sketch created from the code in Fig. 1 by putting holes (denoted by ???) in the place of any ROS APIs. The original ROS1 API is shown in the comments after each hole.

ROS
ROS1 Code

Sketch Generation

Synthesis

ROS2
ROS2 Code

Testing

Sketch Generation

A “sketch” could be generated using data flow analysis (DFA) to create blocks of code. However, in this first prototype, we have manually divided up the code into blocks.

ROS APIs were tagged with 16 different keys that encoded programmer intent (see Table 1).

As first test cases for refactoring, we chose two node programs: a “listener” and a “talker” that communicate over a channel called a *topic*.

Tag	Overlap with					ros	node	Total
	other	sub	pub	constructor				
node	0	4	4	14	0	-	-	66
ros	0	3	0	6	-	-	-	41
constructor	7	9	6	-	-	-	-	31
pub	7	0	-	-	-	-	-	16
sub	0	-	-	-	-	-	-	16
other	-	-	-	-	-	-	-	37

Table 1: Breakdown of the top 5 tags, and all others are in the “other” category. The average number of tags per method was 1.5, thus the numbers in the total column don’t add up to the number of methods in the search space: 142.

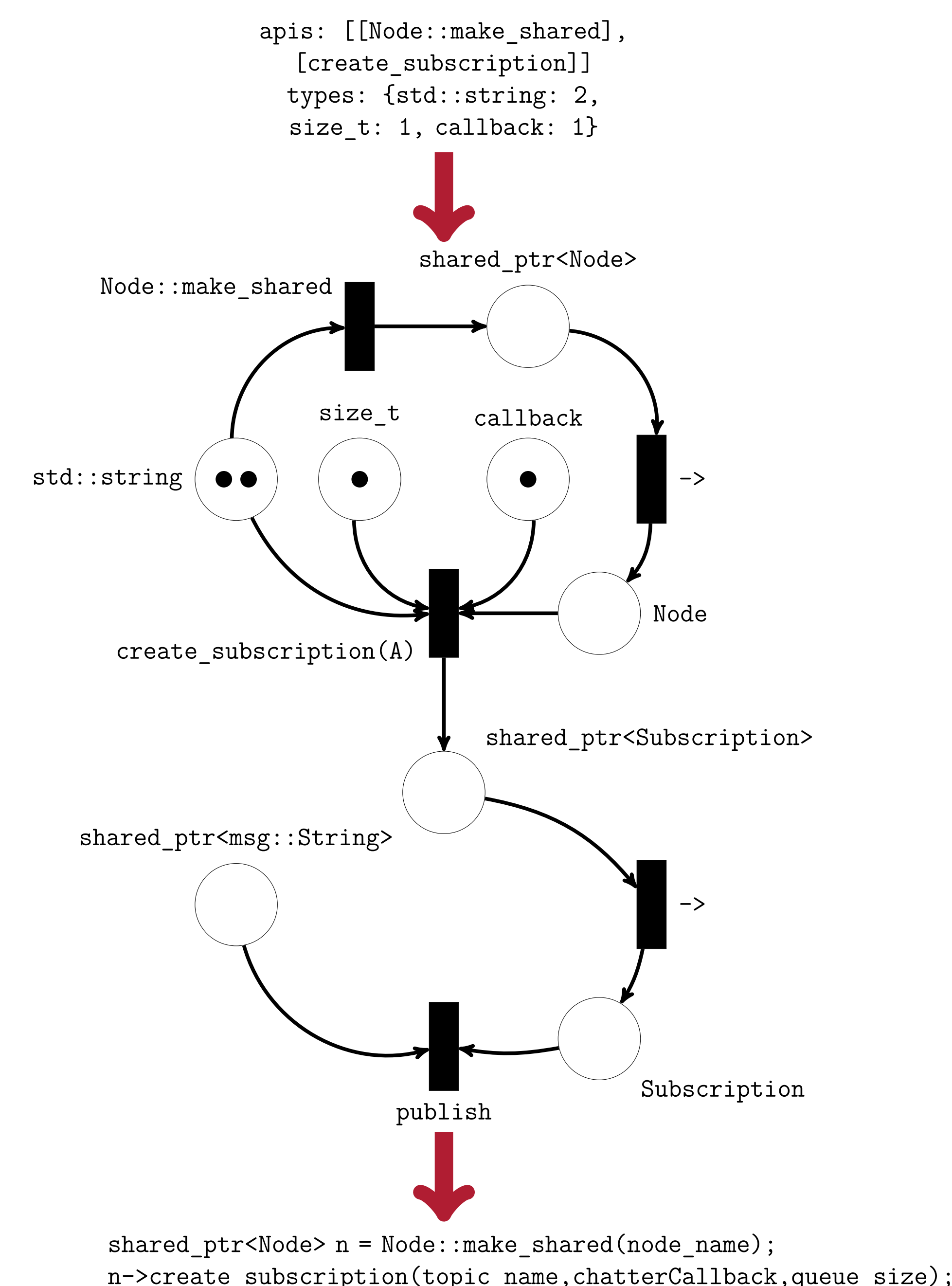


Figure 3: Partial petri net of ROS2 APIs for the listener example (see Fig. 1). Input and output for synthesis is shown above and below, respectively.

Synthesis

The ROS2 code synthesis is driven by searching a petri net (see Fig. 3), where the nodes are types and the transitions are various ROS2 APIs.

Results

The resulting petri net has

- 308 places (=types)
- 560 transitions (=ROS APIs)

If we tried a naive approach to search the petri net, we would have at least 560^n solutions, where n is the number of holes, or:

- $560^4 \approx 9.83 \times 10^{10}$ listener candidates and
- $560^{14} \approx 2.98 \times 10^{38}$ talker candidates.

Using tags and types to guide the search, we get:

- Listener: 16 candidate solutions
- Talker: 64 candidate solutions.

```
1 void chatterCallback(const std_msgs::msg::String::
2   SharedPtr msg){
3   // ...
4 }
5 int main(int argc, char **argv){
6   std::string node_name = "listener_node";
7   rclcpp::init(argc, argv);
8   auto the_freshest_id_0 = rclcpp::Node::make_shared(
9     node_name);
10  std::string topic_name = "chatter";
11  int queue_size = 1000;
12  auto the_freshest_id_1 = the_freshest_id_0->
13    create_subscription<std_msgs::msg::String>(topic_name,
14    chatterCallback, queue_size);
15  rclcpp::spin(the_freshest_id_0);
16  return 0;
17 }
```

Figure 4: An example of a correct synthesized candidate program, for the listener.

Challenges

- C++ grammar: complicated and ambiguous
- C++ types: parametric polymorphism
- Diagnosing cause of failing test
- Slow ROS compile time