Automatic Refactoring of Robotics Code

Carnegie Mellon University

Audrey Seo[1], Ian Voysey[2], Ruben Martins[2]

[1]Wellesley College; [2]School of Computer Science, Carnegie Mellon University

Problem

- ROS is a framework for programming robots
- First version, ROS1, incompatible with ROS2
- Want to automatically upgrade ROS1 to ROS2

Motivation

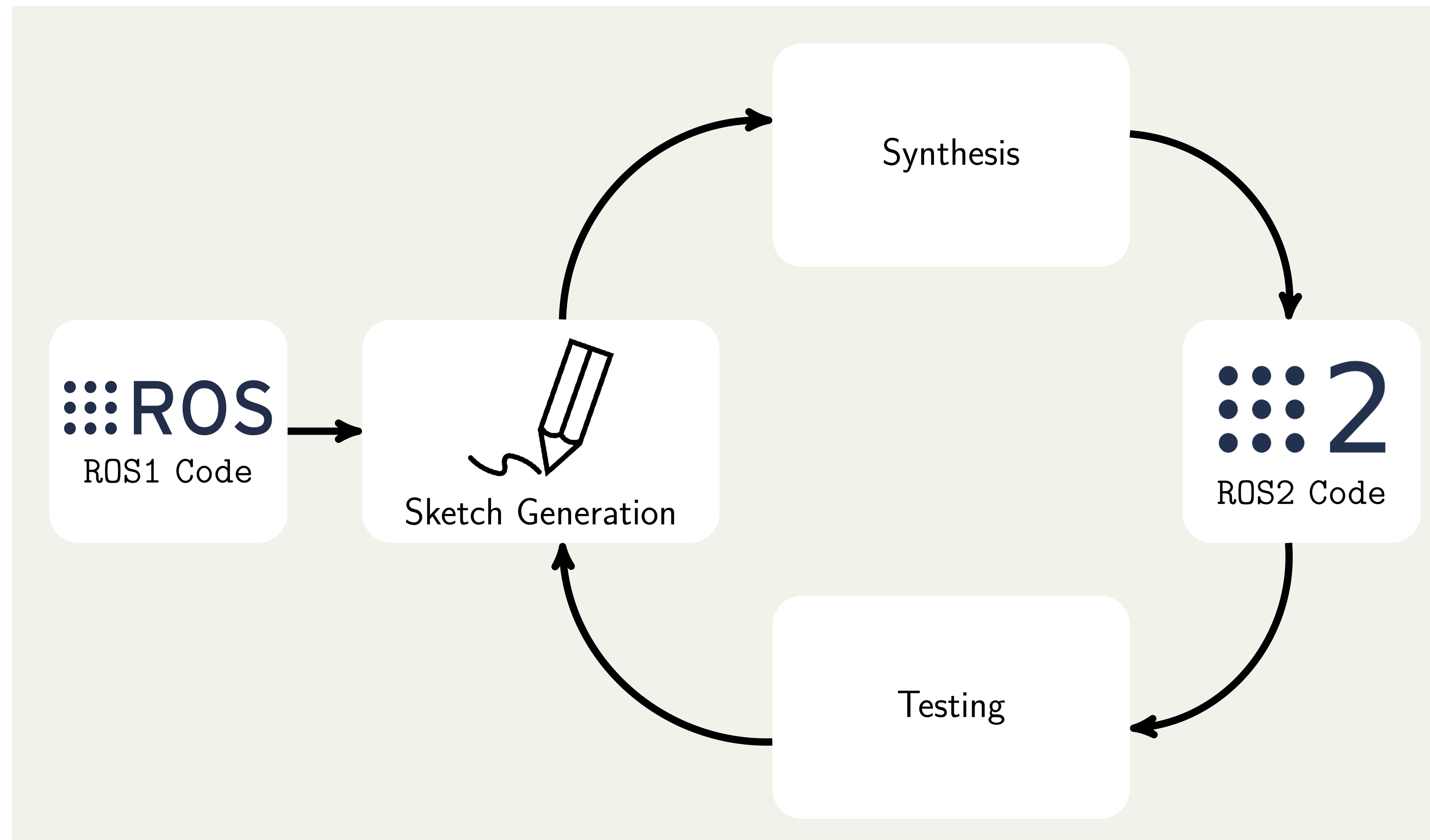
- Difficult to update code (e.g. Python 2.7 vs 3)
- ROS1 has major security vulnerabilities
- Programming robots, which has real world applications and consequences, is higher priority than updating dependencies and refactoring code

```cpp
void chatterCallback(const std_msgs::String::ConstPtr&
    msg){
  // ...
}

int main(int argc, char **argv){
  ros::init(argc, argv, "listener_node");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("chatter", 1000,
    chatterCallback);
  ros::spin();
  return 0;
}
```

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

```cpp
void chatterCallback(const std_msgs::String::ConstPtr&
    msg){
  // ...
}

int main(int argc, char **argv) {
  std::string node_name = "listener_node";
  ?#? // ros::init
  ?#? // ros::NodeHandle
  std::string topic_name = "chatter";
  int queue_size = 1000;
  ?#? // ros::NodeHandle::subscribe
  ?#? // ros::spin
  return 0;
}
```

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



Sketch Generation

A **sketch** is a partially defined program with all ROS1 code removed (see Fig. 2). We had to create these manually, but in the future, this could be automatically generated using data flow analysis. All ROS2 methods were **tagged** with 16 different keys that encoded programmer intent (see Table 1). As first test cases for synthesis, we chose two node programs: a “listener” and a “talker” that communicate over a channel called a *topic*, a core part of the ROS framework. The talker sends messages, which the listener receives.

	Overlap with						
Tag	other	sub	pub	constructor	ros	node	Total
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.

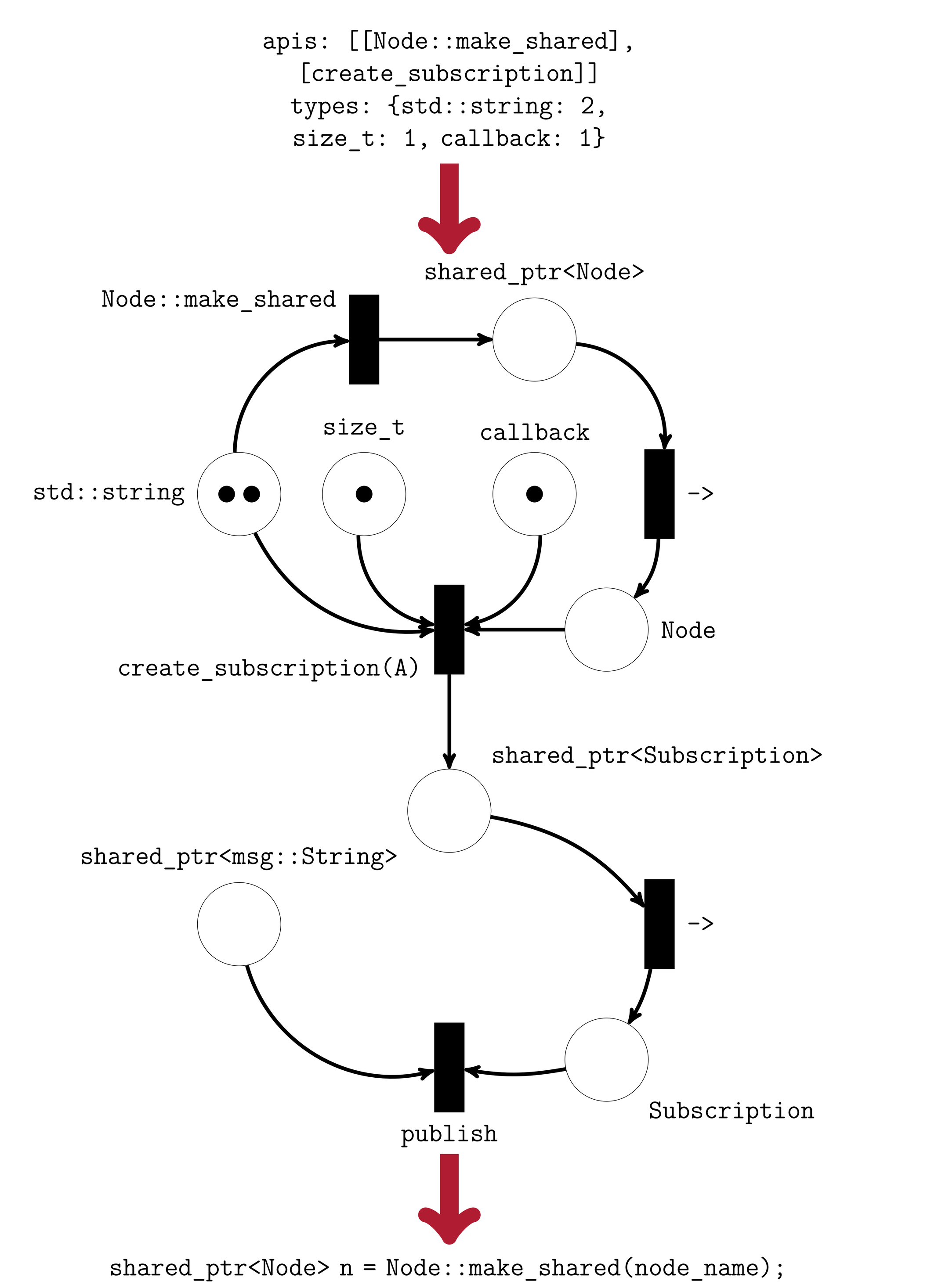


Figure 3: Partial petri net of ROS2 methods for the listener example (see Fig. 1). Input and output for synthesis is shown above and below, respectively.

Synthesis

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

```cpp
void chatterCallback(const std_msgs::msg::String::
    SharedPtr msg){
  // ...
}

int main(int argc, char **argv){
  std::string node_name = "listener_node";
  rclcpp::init(argc, argv);
  auto the_freshest_id_0 = rclcpp::Node::make_shared(
    node_name);
  std::string topic_name = "chatter";
  int queue_size = 1000;
  auto the_freshest_id_1 = the_freshest_id_0->
    create_subscription<std_msgs::msg::String>(topic_name
    , chatterCallback, queue_size);
  rclcpp::spin(the_freshest_id_0);
  return 0;
}
```

Figure 4: An example of a “correct” synthesized listener program, where “correct” means it should typecheck, compile, and communicate with the talker.

Results

The resulting petri net has

- 308 places (types)
- 560 transitions (ROS methods)

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.

Challenges

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