Formalization and Runtime Verification of Invariants for Robotic Systems

Ricardo Cordeiro[1], Alcides Fonseca[1], and Christopher S. Timperley[2]

[1] Faculdade de Ciências da Universidade de Lisboa, Lisboa, Portugal
[2] Carnegie Mellon University, Pittsburgh, PA

Abstract. Robotic systems are critical in today's society. A potential failure in a robot may have extraordinary costs, not only financial, but can also cost lives. Current practices in robot testing are vast and involve methods like simulation, log checking, or field testing. However, current practices often require human monitoring to determine the correctness of a given behavior. Automating this analysis can not only relieve the burden from a high-skilled engineer, but also allow for massive parallel executions of tests, that can detect behavioral faults in the robots that would otherwise not be found due to human error or lack of time. We have developed a DSL (domain-specific language) to specify properties of robotic systems in ROS. Specifications written by developers in this language are compiled to a monitor ROS (Robot Operating System) module, that detects violations of those properties in runtime. We have used this language to express the temporal and positional properties of robots, and we have automated the monitoring of some behavioral violations of robots in relation to their state or events during a simulation.

Keywords: Robotics · Domain-specific language · Runtime Monitoring · Error detection.

1 Introduction

Robotics already have a great impact on our current society. Due to their broad practicality, the quality of software running on robots should be of extreme importance to us. Robotic Systems are non-deterministic, mainly because robots interact directly with the real world. Testing software in such environments is difficult, as there are many variables that can change, and verifying if a task or movement was successful may not be possible from the robots perspective, and external monitoring may be required.

ROS is an open-source framework with a vast collection of libraries, interfaces, and tools that were design to help build robot software. ROS provides an abstraction between hardware and software that helps developers easily connect the different robot components through messages sent through communication channels (*topics*).

Current practices in testing robot software mainly involve field testing, simulation testing, and log checking and require a human to analyze the behavior of the robot to determine whether the behavior is correct.

The goal of our work is to provide developers with a way to automatically verify temporal and positional properties of their robotic systems. Due to the cyber-physical nature of the experiments, we propose a domain-specific language for developers to express their relevant properties and a tool that compiles properties expressed in that language to a monitor component that can be used in simulation to detect violations of those properties.

While our language is based on Linear Temporal Logic (LTL), it was designed from the point of view of ROS developers, allowing properties to reason about native ROS constructs, like *topics*, *messages* and simulation information. Thus, it is possible to express properties that relate the internal information of the system with the corresponding information in the simulator.

We have developed a compiler for this language that generates a ROS module that can be loaded in an existing project. This module will listen to relevant information from both the component and the simulator, and if a violation of any property occurs, it provides a detailed error message to the user.

The paper starts with a motivational example of a hypothetical scenario, followed by a more in detail specification of the domain-specific language, afterward some examples of the DSL usage are shown, and finally, a conclusion on the work is presented.

2 Motivational Example

Let us consider the example of a developer of an autonomous car wanting to express that the robot always needs to stop when going near a stop sign, he could write something like:

```
   after_until robot.distance.stop_sign < 1, robot.distance.stop_sign
> 1, eventually robot.velocity == 0
```

Translating to a more human language we are saying that, after the robot's distance to the stop-sign is below the value of 1 in the simulator, up until the distance is again above 1, the robot velocity will eventually be equal to 0.

The specified property is compiled to a Python file, which is capable of running as a ROS node. The node listens only to relevant topics and performs the computations to verify the specified property.

```
Error at line 3:
  after_until robot.distance.stop_sign < 1, robot.distance.stop_sign > 1, eventually
robot.velocity == 0
Failing state:
     robot.distance.stop_sign:  1.000545118597548
     robot.velocity:  0.17758309727799252
```

Fig. 1. Example of the displayed error when the robot doesn't stop at the stop sign.

The whole flow of the process of monitoring a robotic system is below enumerated:

(i) Using the DSL write the properties of the robotic system one wants to monitor in a .txt file extension
(ii) The specified properties are compiled and a python file is generated which is capable of running as a ROS node
(iii) The node can be run whenever testing the system and will listen to relevant topics and perform the computations needed to verify the specified properties.

3 Specification Language for Robotics Properties

The domain-specific language relies on an adaptation of LTL to express temporal relations of and between simulation objects. The domain-specific language also has shortcuts to express the absolute values of certain useful concepts of objects in a simulation.

3.1 Temporal Keywords

We consider not only LTL basic operators, but also some common shortcuts for useful combinations of such operators, like *after_until*, used in the example:

- **always X** - X has to hold on the entire subsequent path;
- **never X** - X never holds on the entire subsequent path;
- **eventually X** - X eventually has to hold, somewhere on the subsequent path;
- **after X, Y** - after the event X is observed, Y has to hold on the entire subsequent path;
- **until X, Y** - X holds at the current or future position, and Y has to hold until that position. At that position, Y does not have to hold anymore;
- **after_until X, Y, Z** - after the event X is observed, Z has to hold on the entire subsequent path up until Y happens, at that position Z does not have to hold anymore;

It is also possible to reference previous states of variables, using @{X, −y}, representing the value of variable X at time −y.

3.2 Simulation primivitives

To support comparing the internal state of the robotic system with the environment, we provide basic primitives in the language to refer to the simulation environment:

- **X.position** - The position of the robot in the simulation;

- **X.position.y** - The position in the y axis of the robot in the simulation. Also works for x and z;
- **X.distance.Y** - The absolute distance between two objects in the simulation. For the x and y axis;
- **X.distanceZ.Y** - The absolute distance between two objects in the simulation. For the x, y, and z axis;
- **X.velocity** - The velocity of an object in the simulation. This refers to linear velocity;
- **X.velocity.x** - The velocity in the x axis of an object in the simulation. This refers to linear velocity;
- **X.localization_error** - The difference between the robot's perception of its position and the actual position in the simulation;

3.3 Topic declaration

In order to relate robot components with the simulation, the developer can declare the relevant *topic*.

The variable robot_position was declared with the type Odometry.pose.pose.position and is linked to the topic /odom;

```
decl robot_position /odom Odometry.pose.pose.position
```

3.4 Model robots

There are a set of specific topics that can be modeled by robot, like *position* or *velocity*. These will be used by the compiler to call specific functions that need this information from the robot perspective.

```
model robot1:
 position /odom Odometry.pose.pose.position
 ;

never robot1.localization error > 0.002
```

4 Examples

To validate the expressive power of our language, we present examples of expressions, inspired by real-world scenarios.

4.1 Vehicle Maximum Speed

Some robots have a maximum safe speed at which they can move. Sometimes this limit is imposed by law, but some other times by physical constraints.

The robot velocity will never be above 2 for the duration of the simulation;

```
never robot.velocity > 2.0
```

4.2 Follow the Leader

The first robot being above 1 velocity implies that the second robot is at least at 0.8 distance from the first robot. Up until they reach a certain location;

```
   until (robot1.position.x > 45 and robot1.position.y > 45), always
(robot1.velocity > 1 implies robot2.distance.robot1 > 0.8)
```

4.3 Drone height rotors control

After a drone is at a certain altitude both rotors always have the same velocity up until the drone decreases to a certain altitude

```
   decl rotor1_vel /drone_mov/rotor1 Vector3.linear.x
   decl rotor2_vel /drone_mov/rotor2 Vector3.linear.x

   after_until drone.position.z > 5, drone.position.z < 5, rotor1_vel
== rotor2_vel
```

5 Related work

6 Conclusion

The proposed approach is capable of expressing some interesting scenarios that developers care about. We are in the process of expanding our examples to include bugs found in real work projects [1] and from a survey we are conducting with expert developers.

Our development is available online [2], for those that want to experiment with it. We also intend on expanding the primitives with more simulator information, or with the possibility of integrating sensors for supporting field testing in alternative to simulation.

References

1. GitHub repository of ROBUST: ROS Bug Study, https://github.com/robust-rosin/robust
2. GitHub repository of our work, https://ricardocajo.github.io/error-monitor-ros-gazebo