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. These faults could otherwise not be found due to human error or a lack of time. We have developed a Domain Specific Language to specify the properties of robotic systems in the Robot Operating System (ROS). Developer written specifications in this language compile to a monitor ROS 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 significant impact on our current society. Due to their broad practicality, the quality of software running on robots should be extremely important to us. However, robotic Systems are non-deterministic, mainly because robots interact directly with the real world. Testing software in such environments is complex, as many variables can change, and verifying the success of a task or movement may not be possible from the robot's perspective, and external monitoring may be required.

The Robot Operating System (ROS) [1] is an open-source framework with a vast collection of libraries, interfaces, and tools designed to help build robot software. ROS provides an abstraction between hardware and software that helps

* This work was supported by the Fundação para a Ciência e a Tecnologia (FCT) under LASIGE Research Unit (UIDB/00408/2020 and UIDP/00408/2020), and under projects projects CMU|Portugal CAMELOT (LISBOA-01-0247-FEDER-045915) and RAP (EXPL/CCI-COM/1306/2021)

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 robot's behavior to determine whether the behavior is correct.

An invariant represents a property that holds through the execution of the system. Having a set of invariants for a robotic system and asserting them at runtime makes it able to prove the correctness of the system. Research on invariant checking [2] shows that a considerable amount of bugs on autonomous robotic systems can be avoided when representing safety violations of systems and monitoring them.

Due to the unforeseen circumstances mentioned when executing robotic systems, runtime monitoring, although sometimes time-consuming, may be advantageous when identifying errors in these systems. However, implementing a form of runtime monitoring adds load to the simulation. Therefore, not demanding excessive resources is essential when taking this approach. Some challenges in implementing such mechanisms are mentioned in the cited paper [3].

Similar work on runtime monitoring that integrates with ROS already exists. ROSMonitoring [4] can monitor and log errors at the level of *topic* malfunctioning, but it seems unable to express more high-level properties, which is the objective of this work. ROSRV [5] although able to express more high-level specifications, it is highly complex and, in some way, hard for non-expert users to work with. An intuitive domain-specific language will allow a broader set of users to specify a robotic system's properties.

This work aims to provide developers with a way to verify their robotic systems' temporal and positional properties automatically. We propose the introduction of a domain-specific language for developers to express their relevant properties. The given properties compile into monitors that can be used in simulation to ensure the correctness of the system. The language was designed from the point of view of ROS developers and tries to abstract the underlying Linear Temporal Logic (LTL) [6] [7] system, 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 the language that generates a ROS module that monitors an existing project. The module listens to relevant information from both the robot's components and the simulator. If a property violation occurs, it provides a detailed error message to the user.

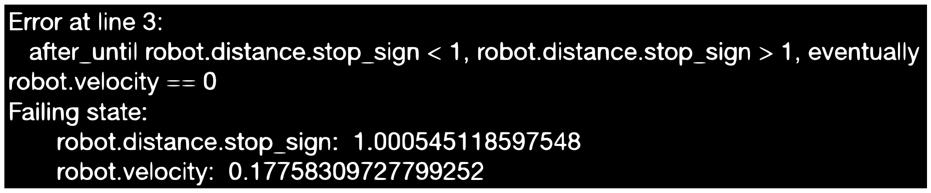
2 Motivational Example

Let us consider an autonomous car developer wanting to express that its system always stops when near a stop sign. The following example presents a property defined in the language that specifies the intended behavior of the developer.

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

Translating into natural language, the property states in the first section that after the robot's distance to the stop-sign is below the value of 1 in the simulator, and in the second section that up until the distance is again above 1, then in the third section the robot velocity will eventually be equal to 0.

The specified property compiles to a Python file 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 does not stop at the stop sign.

The flow of the process of monitoring a robotic system is described as follows:

(i) **Property formalization:** the developer describes in the DSL the properties of the robotic system one wants to monitor in a `.txt` file extension.
(ii) **Compilation:** The specified properties are compiled, and a python file is generated capable of running as a ROS node.
(iii) **Monitoring:** The node can be run whenever testing the system and will listen to pertinent 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 (section 3.1). The DSL has shortcuts to express the absolute values of certain useful concepts of objects in a simulation in order to make it usable (section 3.2). The DSL allows the declaration of robots' relevant topics (section 3.3). And finally, the DSL allows modeling of a whole robot's specific *topics* (section 3.4).

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 previous 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

A set of specific topics can be modeled for the robot, like *position* or *velocity*. The compiler will use these to call specific functions that need this information from the robot's 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 the first robot reaches a particular 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 Conclusion

The proposed approach can express some interesting scenarios that developers care about. We are expanding our examples to include bugs found [3] in real work projects and from a survey, we are conducting with expert developers. Our development is available online [4] 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. M. Quigley, K. Conle, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," *ICRA Workshop on Open Source Software*, vol. 3, pp. 1–6, 01 2009.
2. M. Zizyte, C. Hutchison, R. Duvall, C. Le Goues, and P. Koopman, "The importance of safety invariants in robustness testing autonomy systems," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, pp. 41–44, IEEE, 2021.
3. M. Stadler, M. Vierhauser, and J. Cleland-Huang, "Towards flexible runtime monitoring support for ros-based applications," in *RoSE'22: 4th International Workshop on Robotics Software Engineering Proceedings*, 2022.
4. A. Ferrando, R. C. Cardoso, M. Fisher, D. Ancona, L. Franceschini, and V. Mascardi, "Rosmonitoring: A runtime verification framework for ROS," in *Towards Autonomous Robotic Systems - 21st Annual Conference, TAROS 2020* (A. Mohammad, X. Dong, and M. Russo, eds.), vol. 12228 of *Lecture Notes in Computer Science*, pp. 387–399, Springer, 2020.
5. J. Huang, C. Erdogan, Y. Zhang, B. M. Moore, Q. Luo, A. Sundaresan, and G. Rosu, "ROSRV: runtime verification for robots," in *Runtime Verification - 5th International Conference, RV 2014* (B. Bonakdarpour and S. A. Smolka, eds.), vol. 8734 of *Lecture Notes in Computer Science*, pp. 247–254, Springer, 2014.
6. A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pp. 46–57, ieee, 1977.
7. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Property specification patterns for finite-state verification," in *Proceedings of the second workshop on Formal methods in software practice*, pp. 7–15, 1998.

[3] https://github.com/robust-rosin/robust
[4] https://ricardocajo.github.io/error-monitor-ros-gazebo