UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Formalization and Runtime Verification of Invariants for Robotic Systems

Ricardo Jorge Dias Cordeiro

Mestrado em Engenharia Informática
Especialização em Interação e Conhecimento

Dissertação orientada por:
Prof. Doutor Alcides Miguel Cachulo Aguiar Fonseca
Prof. Doutor Christopher Steven Timperley

2022

Acknowledgments

I would like to thank my coordinator, Prof. Alcides Fonseca, for the exceptional way of teaching not only through the making of my thesis but also throughout all my academic courses.

My coordinator Prof. Chris Timperley for taking his time to help me in this chapter of his life where he had to take care of his baby.

My upperclassman Paulo and Catarina, for all the advice and help.

All my friends who spent time with me know that somehow you helped me through this process.

All my family, in particular my grandparents.

My little brother.

In the end, and more importantly, my mother for taking care of me all my life and giving me the opportunity to follow this path.

"Dreams breathe life into men, and can cage them in suffering. Men live and die by their dreams, but long after they've been abandoned, they still smolder deep in men's hearts. Some see nothing more than life and death. They are dead! For they have no dreams."

Kentaro Miura in Berserk

Resumo

A Robótica tem uma grande influência na sociedade atual, ao ponto que a falha em algum sistema robótico que seja crucial pode impactar o modo em como nós vivemos, se, por exemplo, um carro autónomo provocar a morte de algum passageiro devido a um defeito, futuros e atuais utilizadores deste modelo irão certamente ficar apreensivos em relação à sua utilização. Assegurar que robôs reproduzam um comportamento correto pode assim salvar bastante dinheiro em estragos ou até mesmo as nossas vidas.

As práticas atuais em relação a testes de sistemas robóticos são vastas e envolvem métodos como simulações, verificação de “logs”, ou testagem em campo. Frequentemente, um denominador comum entre estas práticas é a necessidade de um humano pessoalmente analisar e determinar se o comportamento de um sistema robótico é o correto. A automatização deste tipo de análise poderia não só aliviar o trabalho de técnicos especializados, facilitando assim a realização de testes, mas também possivelmente permitir a execução massiva de testes em paralelo que podem potencialmente detetar falhas no comportamento do sistema robótico que de outra maneira não seriam identificados devido a erros humanos ou à falta de tempo.

Apesar de existir alguma literatura relacionada com esta investigação, de uma maneira geral a automatização no campo da deteção de erros ou criação de invariantes continua a não ser adotada, pelo que o estudo apresentado nesta tese é justificado.

Esta dissertação visa assim explorar o problema da automatização na deteção de erros comportamentais em robôs num ambiente de simulação, introduzindo uma linguagem de domínio específico direcionada a especificar as propriedades de sistemas robóticos em relação ao seu ambiente, assim como a geração de “software” de monitorização capaz de detetar a transgressão destas propriedades.

A linguagem de domínio específico necessita de expressar requisitos de determinados estados ou eventos durante a simulação, desta maneira precisa de apresentar determinadas características. Palavras-chave para representar relações temporais de ou entre objetos, como, por exemplo, o robô ”nunca”, ou ”eventualmente”o robô. Referências a estados anteriores da simulação, como, por exemplo, a velocidade do robô está sempre a aumentar, ou seja, é sempre maior que no estado anterior. Atalhos para ser possível referir certas características de ou entre objetos, como, por exemplo, a ”posição”, ”velocidade”ou ”distância”de ou entre robôs.

A linguagem de domínio específico também assume que o sistema robótico irá ser executado por meio da framework ROS (Robot Operating System), que é amplamente utilizada para investigação e na indústria da robótica. A arquitetura do ROS engloba características como

“publish-subscribe” entre ”tópicos”e tipos de mensagem, estas características são tidas em conta e foram integradas no desenvolvimento da linguagem.

O “software” de monitorização gerado refere-se a um ficheiro python que correrá sobre a framework ROS. A geração deste ficheiro assume também que a monitorização será feita no simulador Gazebo, isto porque para obter dados como a posição ou velocidade absoluta de um robô durante a simulação é necessário aceder a ”tópicos”ROS específicos que na geração do ficheiro de monitorização estão “hardcoded”. A geração de um ficheiro capaz de executar a monitorização significa que esta pode ser executada independente de um sistema robótico, permitindo assim a automatização da monitorização a respeito de vários objetos e as suas relações.

Resultados mostram que é possível expressar propriedades temporais e posicionais de e entre robôs e o seu ambiente com o suporte da linguagem de domínio específico. O trabalho mostra também que é possível automatizar a monitorização da violação de alguns tipos de comportamentos esperados de robôs em relação ao seu estado ou determinados eventos que ocorrem durante uma simulação.

Evaluation ?

possiveis problemas, e futuro - proof of language or that it works - better information on the errors - frequency of checking the properties can be modified in some circunstances to not check at every iteration - alargar a outros simuladores - integration with other tools liek scenario generation

Palavras-chave: Robótica, Linguagem de domínio, Monitorização em tempo de execução, Deteção de erros

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.

Evaluation ?

Keywords: Robotics, Domain-specific language, Runtime Monitoring, Error detection

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This thesis aims to explore a possible solution for automation in the testing of robotic systems through the a domain-specific language (DSL) and simulation software.

This chapter intends to introduce the motivation for this work (section 1.1), present the problem statements of such an approach (section 1.2), discuss the objectives (section 1.3), show a motivational example of the developed work (section 1.4), present the expected contributions (section 1.5), and finally summarize the structure of the rest of the document (section 1.6).

1.1 Motivation

Robotics already significantly impact our current society, in industry, in medicine, in agriculture, or leisurely in sports contests or personal use. Robotics often take critical roles like the example of robot arms in car assembly lines or autonomous farms. The tendency is for robot usage to keep growing at a global level.

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.

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. Due to their broad practicality, the quality of software running on robots should be extremely important to us. Robot software, as well as the techniques used to test their quality, are very field-specific and different from the techniques employed in traditional Software Engineering mainly because of their real-world interaction. This peculiarity means automatic tests are barely used in robotics. Studying possible options for viable automation of tests in robotic systems could lead to an opening on its usage in both research and the industry. Also, allowing for multiple parallel executions of tests not depending on human monitoring could improve the quality of current and future robot software.

1.2 Problem Statement

These multiple challenges in robot testing influence the planning for testing a robot because there are tradeoffs among choices.

Using simulation-based testing, developers can take advantage of real values of objects' attributes to compare with what the robot system perceives. Using this alleyway, it is possible to, in a way, surpassing the need for human-in-the-loop testing.

While simulation-based tests are a promising approach for automation, there is still distrust in the precision and validity of the results. As a result, simulation-based autonomous testing is barely used due to reliability and factors like cost and complexity. Due to these factors, despite being dangerous, sometimes expensive, or work-intensive, real-life robot testing or other methods are still the main choices. The resulting product is a lack of quality in the software across projects.

In this thesis, we address the problem of defining automated tests for robotics systems.

1.3 Objectives

The ultimate goal of this thesis is to remove the need for human-in-the-loop testing of robotic systems by studying a possible solution for automation in simulation-based tests.

This work aims to provide developers with a way to verify their robotic systems' temporal and positional properties automatically. To this end, we propose the introduction of a DSL 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 DSL was designed from the point of view of the Robot Operating System (ROS) [7] developers and tries to abstract the underlying Linear Temporal Logic (LTL) system. The DSL allows 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.

The DSL should allow describing a robotic system's properties simply and intuitively while simultaneously expressing relevant temporal and positional arguments between robots components and objects in the simulation.

1.4 Motivational Example

Let us consider an autonomous car developer wanting to express that the car 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

```
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
```

Figure 1.1: Example of the displayed error when the robot does not stop at the stop sign.

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.

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.

1.5 Contributions

The expected contributions of this thesis are below enumerated.

1. Definition of a domain-specific language to specify robotic systems' properties.
2. Implementation of a compiler for the language that can generate software capable of monitoring relevant components while in a simulation.
3. Evaluation of the expressive capabilities of the solution.

1.6 Structure of the document

The document is organized as follows:

- chapter 2 - Background & Related Work:
- chapter 3 - Specification Language for Robotics Properties
- chapter 4 - Monitoring
- chapter 5 - Evaluation
- chapter 6 - Future Work
- chapter 7 - Conclusion

Chapter 2

Background & Related Work

This chapter gives an overview of the background software adopted while developing this work (section 2.1), and shines some light on the already existing similar work and adopted techniques on the subject (section 2.2).

2.1 Background

This section provides some background on the used software and the reason for its choice, what the Robot Operating System is (subsection 2.1.1), and the simulation software adopted (subsection 2.1.2).

2.1.1 Robot Operating System

The Robot Operating System (ROS) [7] 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 developers easily connect the different robot components through messages sent through communication channels (*topics*).

ROS has a modular architecture and other advantages built with the purpose of cross-collaboration and easy development. For all these reasons, ROS is used by hundreds of companies and research labs.

The ROS ecosystem is built so that the majority of the projects depend on a specific set of packages. Literature states that around eighty-two percent of ROS applications rely on packages released by a small subset of groups [5].

2.1.2 Gazebo

Robotic systems simulation is an essential tool for testing robots' behavior. For this reason, Gazebo [4] started with the idea of a high-fidelity simulator to simulate robots in any environment under mixed conditions.

Gazebo is an open-source 3D simulator that supports tools like sensors simulation, mesh management, and actuators control under different physics engines, among others, which makes it a simulator that very distinct robotic systems can use.

2.2 Related Work

This section shows some research on runtime testing, the different techniques, and the difficulties of implementing it already exist (subsection 2.2.1). The importance of Invariant specification and its relation to Linear Temporal Logic (LTL) (subsection 2.2.2). Some monitoring frameworks have already tried to implement similar runtime verification concepts (subsection 2.2.3).

2.2.1 Runtime Testing

Due to the unforeseen circumstances mentioned when executing robotic systems, runtime testing, although sometimes time-consuming, may be advantageous when identifying errors in these types of systems.

Implementing 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 [8].

TODO mention methods of runtime monitoring (mine and others) *mythra paper -> other types of machine learning mention it*

2.2.2 Linear Temporal Logic and Invariants

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 [9] shows that a considerable amount of bugs on autonomous robotic systems can be avoided when representing safety violations of systems and monitoring them.

TODO change text so Invariants are -> specifying the invariants and system for checking *integrate ltl with invariants for system checking*

Linear temporal logic (LTL) is a branch of logic responsible for representing and reasoning about modalities in reference to time.

As an approach for program verification, a formal system of temporal logic was suggested for both sequential and parallel programs [6]. LTL can be used as a method of model-checking [1] using its patterns as a form of property specification. It includes patterns such as "always", "finally", "until", "eventually", and others, which can be useful in the creation of invariants for program verification of robotics systems.

2.2.3 Monitoring Frameworks

Similar work on runtime monitoring that integrates with ROS already exists.

ROSMonitoring [2] 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. **Alcides**
>What are the high-level properties? Can you describe the difference? Or at least give examples of things that are not possible?<

ROSRV [3] 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. **Alcides** >*Why is it hard for non-experts? This needs to be explained!*<

Alcides >*Para cada uma destas ferramentas tens de ir com muito mais detalhe a dizer o que fazem, como funcionam e quais (e porque das) as suas limitações!.* <

Chapter 3

Specification Language for Robotics Properties

In this chapter, the structure and intricacies of the DSL are presented. The notations used in the DSL, like concepts and keywords, are introduced in (section 3.1). The DSL grammar is written in the Backus-Naur Form (section 3.2). Finally, some practical examples are written with the help of the DSL to display its expressiveness (section 3.3).

3.1 Language Notations

The high-level concepts that can be created in the language are:

- **Property -** A property represents a temporal specification or a blend of temporal specifications between components.
- **Declaration -** A declaration allows for the representation of ROS *topics* in order to interact with it.
- **Model -** A model allows for the declaration of specific *topics* that are required when correlating certain robots' and simulation components.
- **Association -** An association serves as a way to create program variables.

3.1.1 Temporal Keywords

We consider not only LTL basic operators but also some common shortcuts for useful combinations of such operators, like *after_until*.

- **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;

3.1.2 Temporal value

It is also possible to reference previous variable states:

$$@\{X, -y\} \tag{3.1}$$

This will represent the value of the variable X in the point in time -y.

3.1.3 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;

3.1.4 Operands

Besides the already mentioned operands, *Temporal values*, *Simulation primivitives*, and *Temporal Keywords*, the DSL also supports both Integer and Float values, Booleans, and declared variables.

3.1.5 Operators

The DSL supports operators to correlate components. The operators are *+* (addition), - (subtraction), *** (multiplication), */* (division), == (equals), *!=* (different), > (greater than), >= (greater or equal than), < (lower than), <= (lower or equal than), *and* (conjunction), *or* (disjunction), *implies* (implication), and for any comparison operator X *Xy* - the values being compared will have an error margin of y (Example: Z ==0.05 Y).

3.1.6 Protected Variables

Protected variables are variable names restricted to set determined monitoring parameters.

rate - Set the frame rate which properties are checked (By default, the rate is 30hz)

timeout - Set the timeout for how long the verification will last (By default, the timeout is 100 seconds)

margin - Set the error margin for comparisons

3.1.7 Topic declaration

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

The language cannot inherently have a way to interact with specific components of a robot because it does not know which topic to get information from. Therefore, the developer needs to declare these specific topics to be able to interact with them.

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

3.1.8 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

3.2 Grammar

```
<program>         ::=  <command> | <command> <program>
<command>         ::=  <association> | <declaration> | <model> | <pattern>
<association>     ::=  name = <pattern>
                   |   _rate_ = integer
                   |   _timeout_ = <number>
                   |   _default_margin_ = <number>
<declaration>     ::=  decl name topic_name <msgtype>
                   |   decl name name <msgtype>
<model>           ::=  model name : <modelargs> ;
<modelargs>       ::=  <name> topic_name <msgtype>
                   |   <name> <name> <msgtype>
                   |   <name> topic_name <msgtype> <modelargs>
                   |   <name> <name> <msgtype> <modelargs>
<name>            ::=  name | <func_main>
<func_main>       ::=  position | velocity | distance | localization_error | orientation
<msgtype>         ::=  <name> | <name> . <msgtype>
<pattern>         ::=  always <pattern>
                   |   never <pattern>
                   |   eventually <pattern>
                   |   after <pattern> , <pattern>
                   |   until <pattern> , <pattern>
                   |   after_until <pattern> , <pattern> , <pattern>
                   |   <conjunction>
<conjunction>     ::=  <conjunction> and <comparison>
                   |   <conjunction> or <comparison>
                   |   <conjunction> implies <comparison>
                   |   <comparison>
<comparison>      ::=  <multiplication> <opbin> <multiplication>
                   |   <multiplication> <opbin> <number> <multiplication>
                   |   <multiplication>
<opbin>           ::=  < | > | <= | >= | == | !=
<multiplication>  ::=  <multiplication> * <addition>
                   |   <multiplication> / <addition>
                   |   <addition>
<addition>        ::=  <addition> + <operand> | <addition> - <operand> | <operand>
<operand>         ::=  name
                   |   <number>
                   |   true
                   |   false
                   |   <func>
                   |   <temporalvalue>
                   |   ( <pattern> )
<number>          ::=  float | integer
<func>            ::=  name . <func_main> | name . <func_main> <funcargs>
<funcargs>        ::=  . <name> | . <name> <funcargs>
<temporalvalue>   ::=  @ name , integer
```

3.3 DSL Usage Examples

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

3.3.1 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

3.3.2 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)

3.3.3 Localization error

The localization error (difference between the robot's perception of its location and the actual simulation location) of the robot is never above a specific value.

model robot1:

position /odom Odometry.pose.pose.position

;

never robot1.localization error > 0.002

3.3.4 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

Chapter 4

Monitoring

This chapter explains the whole process of monitoring, from compilation to error detection. First, the overall process of compilation is explained (section 4.1), then the generated file and some of its specifications are described (section 4.2), and finally, the shown error messages are illustrated (section 4.3).

4.1 Runtime Monitoring

After writing all the desired robotic systems specifications, the file needs to be compiled to generate the monitoring python file. Currently, to compile a specification file, one has to do it through the console under the same directory of the *language.py* file. As for the command:

python language.py properties.txt /home/ros_workspace/src/my_pkg/src

The *language.py* file needs to be run as a python file and be given as arguments:

1. The specifications file name, and in case it is not in the same directory as the *language.py* file, give its absolute path.

2. The expected absolute path of the generated python monitoring file.

The given directory for the generated file should be under a ROS workspace for the compilation to succeed. This is because, during the compilation, access to information like the available ROS messages might be necessary.

The monitoring file can now run as an independent ROS node, integrated into a launch file, or using rosrun in the console to execute it.

4.2 Generated File

Declare the needed subscribers and use ApproximateTimeSynchronizer to call the callback function. The ApproximateTimeSynchronizer synchronizes messages by their timestamp and if they do not have a header, use the ROS time.

The callback function is called every time a new message from one of the subscribers is received. The callback function saves the relevant information for property checking in a global variable. This information serves as a current ”screenshot” of the simulation representing its current state.

The node executes a loop at a delineated rate, doing the following tasks.

Check if the defined simulation timeout time has reached.

Save the current simulation state obtained by the callback function. This is necessary because the callback function is called at fluctuating rates, and we want to save multiple ”screenshots” of the simulation at the loop fixed rate to make correlations with past states.

Verify the properties using the saved states and calling each created function. An independent function with the necessary computations for verifying the property is defined for each base property.

4.3 Error Messages

An error message starts by stating the line in the specification file which resulted in an error and showing the specification itself.

Afterward, the value at the time of failure of all the variables present at the specification is shown.

```
Error at line 19:
 until turtlebot3_burger.position.x <= -1 and turtlebot3_burger.position.y < -1,
 never turtlebot3_burger.velocity >= 0.1
Failing state:
    turtlebot3_burger.position.x: 0.1134222404473394
    turtlebot3_burger.position.y: 0.0011127060961216948
    turtlebot3_burger.velocity: 0.10048210526931797
```

Figure 4.1: Example of an error message.

Chapter 5

Evaluation

Chapter 6

Future Work

integrate gzscenic and my language

give better error messages

frequency of checking the properties can be modified in some circunstances to not check at every iteration

validation of the proposal (which is surprising, as the paper contains print screens of the system output). After the presentation of the rules formulation in the proposed language, the authors immediately turn into the Conclusions, without any experimental data to validate the proposal. This is certainly a major flaw of the paper as, taking into consideration what has been presented and the overall quality of the paper in other aspects (structure, quality of writing, ...), the authors could have gone a litter further in providing some evidence on the effective capabilities of their proposal. As it is, the readers only have a proposal of a language and a couple of examples of rule implementation, but no concrete evidence that the system actually detects any rule violation, when it occurs, nor that it can do so without disturbing the simulation by demanding excessive resources, for instance.

Integrate the language with more industrial simulators.

The tests could allow the robot's automatic correction and generation of code. Generating multiple alternatives and automatically evaluating how good they are, improving the code to do what we want.

Chapter 7

Conclusion

This is the paper conclusion

The proposed approach can express some interesting scenarios that developers care about. We are expanding our examples to include bugs found [1] in real work projects 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.

[1] https://github.com/robust-rosin/robust

[2] https://ricardocajo.github.io/error-monitor-ros-gazebo

Appendix A

Bibliography

[1] Matthew B Dwyer, George S Avrunin, and James C Corbett. Property specification patterns for finite-state verification. In *Proceedings of the second workshop on Formal methods in software practice*, pages 7–15, 1998.

[2] Angelo Ferrando, Rafael C Cardoso, Michael Fisher, Davide Ancona, Luca Franceschini, and Viviana Mascardi. Rosmonitoring: a runtime verification framework for ros. In *Annual Conference Towards Autonomous Robotic Systems*, pages 387–399. Springer, 2020.

[3] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon Moore, Qingzhou Luo, Aravind Sundaresan, and Grigore Rosu. Rosrv: Runtime verification for robots. In *International Conference on Runtime Verification*, pages 247–254. Springer, 2014.

[4] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE, 2004.

[5] Sophia Kolak, Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. It takes a village to build a robot: An empirical study of the ros ecosystem. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 430–440, 2020. doi: 10.1109/ICSME46990.2020.00048.

[6] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. ieee, 1977.

[7] Morgan Quigley, Ken Conle, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. *ICRA Workshop on Open Source Software*, 3(3.2):1–6, 01 2009.

[8] Marco Stadler, Michael Vierhauser, and Jane Cleland-Huang. Towards flexible runtime monitoring support for ros-based applications. In *RoSE'22: 4th International Workshop on Robotics Software Engineering Proceedings*, 2022.

[9] Milda Zizyte, Casidhe Hutchison, Raewyn Duvall, Claire Le Goues, and Philip 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)*, pages 41–44. IEEE, 2021.