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 at exploring a possible solution for automation in the testing of robotic systems through the medium of 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), present the expected contributions (Section 1.4), and finally summarize the structure of the rest of the document (Section 1.5).

1.1 Motivation

Robotics already significantly impact our current society, industrially, for instance, in medicine and agriculture, or leisurely in 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

The multiple challenges in robot testing influence planning how to test a robot because there are tradeoffs among choices.

Using simulation-based testing, the 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.

When developing a DSL for simulation-based autonomous tests, problems like what components to monitor and how to express them arise. Having a DSL to specify a robotic system's properties can be useful. However, there is a need to control its complexity and accessibility, or it can become a burden in the testing process.

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. 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) [6] 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 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.

1.5 Structure of the document

The document is organized as follows:

- Chapter 2 - Background & Related Work:
- Chapter 3 - Proposed Approach:
- Chapter 4 -

Chapter 2

Background & Related Work

This chapter gives an overview of the software adopted while developing this work (section 2.1), followed by a brief explanation of Linear Temporal Logic (section 2.2), and finally shines some light on the already existing similar work on the subject (section 2.3).

2.1 Software

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) [6] 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.

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 Linear Temporal Logic

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 [5]. 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.

2.3 Robot Testing

Some research on the importance of invariants checking (subsection 2.3.1) and runtime testing and the difficulties of implementing it already exist (subsection 2.3.2). As well as some tools that already try to implement similar runtime verification concepts (subsection 2.3.3).

2.3.1 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 [8] shows that a considerable amount of bugs on autonomous robotic systems can be avoided when representing safety violations of systems and monitoring them.

2.3.2 Runtime Monitoring

Due to the unforeseen circumstances mentioned when executing robotic systems, runtime monitoring, 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 [7].

2.3.3 Similar work

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.

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.

Chapter 3

Specification Language for Robotics Properties

structure explanation

3.1 High Level Notations

- **Declaration -** aa
- **Property -** aa
- **Model -** aa
- **Association -** aa

3.1.1 Temporal Keywords

– **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.

3.2 Operands

Besides Number / Boolean / Var

- **Temporal value -** aa
- **Function -** aa
- **Property -** aa

3.2.1 Simulation primivitives

- **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 Protected Variables

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

3.4 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
```

3.5 Model robots

There are a set of specific topics that can be modeled for the robot, like *position* or *velocity*. These will be used by the compiler 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
```

3.6 Grammar

```
Start  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
```

Chapter 4

Monitoring

Compile -> generate file -> ROS

4.1 Generated File

4.1.1 Fetch simulation data

4.1.2 Verifying properties

4.2 Error Messages

Chapter 5

Proposed Approach

The proposed approach consists initially in creating a domain-specific language. The language will serve as a way to describe the properties of a robot. For instance, if our robot is an autonomous car navigating on the road, one property could be that the robot always stops at stop signs. To describe robot properties, we also need the description of the testing scenario. In the above example, the ”road” and ”stop sign” should be defined in the language as part of the testing scenario, without it there would be no way to describe the above property effectively. To describe the scenario itself we can use GzScenic in order to take advantage of the arbitrary creation of multiple scenario possibilities. This language will then be composed of a new domain-specific language in association with the already established GzScenic language.

Next in the approach, there is a need to build a compiler for the proposed language. The compiler should be able to interpret a property in the language and be able to identify the components of the robot necessary to monitor the said property. The monitorization could take place either during runtime or after using log files. Taking the above example into account, our compiler should have the information of which component of the robot is responsible for the car position as well as the position of the stop sign, it can then monitor the component and check if the property has been broken.

The language should be of high level in the sense that it should be intuitive to the writer. With this approach, the person doing the robot testing shouldn't need so much in-depth knowledge about the robot to perform a test. This is because of the writing simplicity of the language and the removal of the manual labor side of personally monitoring the robot.

The final scheme of the tool proposed is represented in the below diagram.

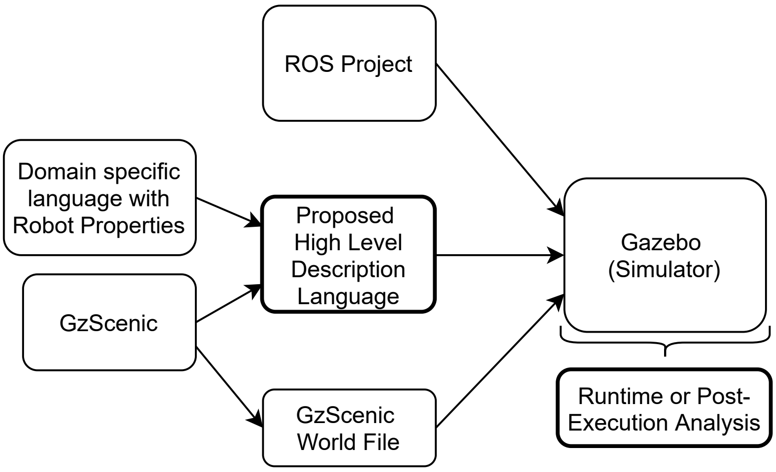

Figure 5.1: Tool for monitoring robot properties.

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

[6] 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.

[7] 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.

[8] 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.