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, seja industrialmente, na medicina, na agricultura, ou como forma de lazer, e, muitas vezes, toma um papel crucial em que uma falha em algum destes sistemas robóticos cruciais 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.

Os sistemas robóticos são não deterministicos, isto porque os robôs interagem diretamente com o mundo real. Testar *software* num destes ambientes é bastante complexo devido às variáveis serem imprevisíveis e mudarem constantemente. Verificar o sucesso de um movimento ou tarefa neste ambiente pode ser bastante difícil do ponto de vista de um robô, pelo que monitorização externa é muitas vezes necessária.

Devido à ampla utilização de sistemas robóticos, a qualidade do software que corre em robôs deveria ser de extrema importância para nós. O *software* destes sistemas, assim como os métodos utilizados para os testar, são bastante específicos da área e diferentes de *software* tradicional, em grande parte devido à sua já falada interação com o mundo real.

As práticas atuais em relação à testagem de sistemas robóticos são vastas e envolvem métodos como simulações, verificação de "logs", testes em campo, entre outras. Frequentemente, um denominador comum entre as práticas adotadas 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. Este tipo de testagem automática poderá potencialmente detetar falhas no comportamento do sistema robótico que de outra maneira não seriam identificadas devido a erros humanos ou mesmo à falta de tempo.

Uma invariante representa uma propriedade que se mantém durante toda a execução do sistema, dispor de uma lista de invariantes para um sistema robótico e ser capaz de as verificar em tempo de execução é uma forma de provar a qualidade desse sistema.

Atualmente já existe investigação e literatura substancial relacionada com a utilização de testes automáticos em sistemas robóticos, assim como ferramentas para realizar de alguma maneira este tipo de análise. No entanto, de uma forma geral, a automatização no campo da deteção de erros ou até mesmo na utilização de invariantes continua a não ser adotada para este tipo de sistemas. O problema na adoção deste tipo de ferramentas deve-se à complexidade ou à

falta de confiança nas soluções já desenvolvidas, algo que incentiva o estudo apresentado nesta tese.

Esta dissertação visa assim explorar o problema da automatização da 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 durante uma simulação.

A linguagem de domínio específico também assume que o sistema robótico irá ser executado por meio da framework de código aberto ROS (Robot Operating System). O ROS é uma framework que oferece uma vasta coleção de livrarias, interfaces e ferramentas especificamente desenhadas para ajudar na construção de *software* para robôs. O ROS fornece uma abstração entre *software* e *hardware* que ajuda desenvolvedores a conectar facilmente diferentes componentes de robôs através de mensagens enviadas por canais de comunicação chamados *tópicos*. O ROS tem uma arquitetura modular e outras vantagens que têm como objectivo a intra-colaboração e o fácil desenvolvimento de *software*. O seu ecossistema está construído de maneira a que a maioria dos projetos dependam de uma pequena lista de pacotes. Devido a todos os factos em cima mencionados o ROS é amplamente utilizado para investigação e na indústria da robótica, e por essas mesmas razões o escolhemos como a framework que seria integrada neste trabalho.

A Lógica Temporal Linear pode ser usada como um método de verificação de programas e também ser útil para a criação de invariantes em sistemas robóticos. Um sistema formal de lógica temporal contém padrões que podem ser usados como uma forma de especificação de propriedades deste tipo de sistemas.

A Lógica Temporal Linear é um ramo da lógica responsável por representar e relacionar componentes em referência a uma linha temporal. 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, e atalhos para ser possível referir certas características de ou entre objetos, como, por exemplo, a "posição", "velocidade" ou "distância" de ou entre robôs.

O *software* de monitorização gerado é um ficheiro *Python* que tem origem numa especificação feita através da linguagem de domínio específico e 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*, pois dependem do *software* de monitorização escolhido.

O Gazebo é um simulador de alta-fidelidade capaz de simular sistemas robóticos em qualquer tipo de ambiente ou condições. O Gazebo é um simulador de código aberto que suporta ferramentas como a simulação de sensores, manipulação de modelos, e o controlo de atuadores sob diferentes motores de física, entre outras ferramentas, o que o torna um simulador de que

vários sistemas robóticos diferentes entre si são capazes de tirar proveito, daí a sua escolha para o desenvolvimento deste trabalho.

A geração de um ficheiro capaz de executar a monitorização durante uma simulação significa também que este tipo de monitorização pode ser executada independente de um sistema robótico, permitindo assim a automatização da monitorização a respeito de vários objetos e as suas relações.

O objetivo deste trabalho é então fornecer uma maneira de desenvolvedores de *software* para sistemas robóticos conseguirem verificar propriedades posicionais e temporais dos seus sistemas de maneira automática através de uma linguagem de domínio específico, que deve, ao mesmo tempo, ser simples, intuitiva e permitir expressar propriedades que sejam relevantes entre componentes da simulação.

De maneira a avaliar o trabalho desenvolvido tentámos detetar erros em sistemas robóticos, inserindo propositadamente *bugs* no sistema. Estes *bugs* provêm de uma lista de *bugs* que foram previamente identificados por outros utilizadores destes sistemas robóticos. A lista contém *bugs* bastante diferentes entre si, dos quais a maior parte já foi corrigido em *software* atual, sendo que o nosso objetivo é identificar *bugs* antigos que ocorram em tempo de execução.

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.

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

Abstract

Robotic systems are critical in today's society, be it in manufacturing, medicine, or agriculture. 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.

I 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 at runtime. I have used this language to express the temporal and positional properties of robots using Linear Temporal Logic as a basis for the language stipulation. I have also automated the monitoring of some behavioral violations of robots in relation to their state or events during a simulation, resorting to relations between the internal information of the system and the corresponding information in the simulator.

To evaluate the developed work, I went through a list of documented ROS bugs and identified some that happen at runtime. Using these bugs as a basis I specified the robot's properties in the developed language that should be capable of detecting an error, in order to test both the expressiveness and the monitoring while running the system.

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

Contents

List of Figures

Chapter 1

Introduction

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

A DSL is a programming language that is written for a specific domain, it usually provides a higher level of abstraction and is simpler than other languages mainly because they are intended to be used by people knowledgeable of the domain.

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 rarely used in robotics [12, 16].

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 existing challenges in robot testing have influence in planning tests for a robotic system, because there are tradeoffs among the possible choices.

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

Human-in-the-loop testing refers to a method of testing that requires a human to perform a certain task that can't be covered by means of automation or simulation. The main problem with human-in-the-loop testing is that it requires the time of a high-skilled professional which could be used for something else, human-in-the-loop testing is also restricting because a human can only supervise one test at a time.

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 rarely used due to reliability and factors like cost and complexity [12, 16]. Due to these factors, despite being dangerous, sometimes expensive, or work-intensive, field 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, I address problems of defining simulation-based 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' properties in relation to their position in a simulation (positional properties) as well as correlations between current and past events (temporal properties). To this end, I 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) [14] developers and tries to abstract the underlying Linear Temporal Logic (LTL) system. LTL is a branch of logic responsible for representing and reasoning about modalities in reference to time. 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. For this reason, the expected design goals of the DSL that I believe conform with ROS developers are:

- Temporal operators based on but not restricted to Linear Temporal Logic.

- Primitives that reference simulation environment variables, like the position, velocity, and others.
- Basic operators to make comparisons between defined components, like greater than, equals, and others.

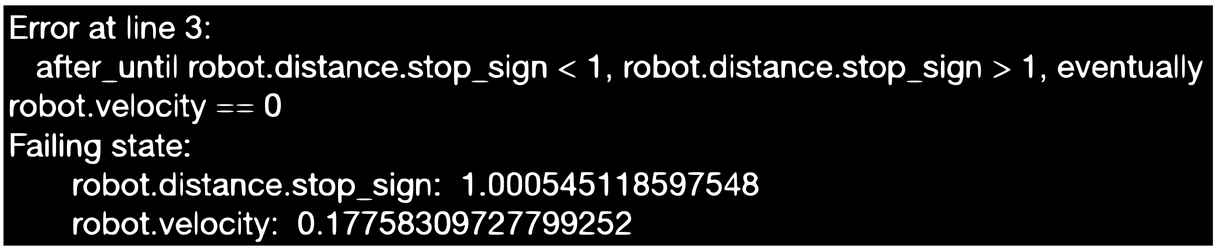
1.4 Motivational Example

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 up until the distance is again above 1, then (in the third section) the robot velocity will eventually be equal to 0.

The toolchain compiles the DSL specification to executable Python code that 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
```

Figure 1.1: 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 text file.

(ii) **Compilation:** The specified properties are compiled, and a Python file, capable of running as a ROS node, is generated.

(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 enumerated below.

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

  An overview of the background literature and similar work.

- Chapter 3 - Specification Language for Robotics Properties

  The structure and concepts of the domain-specific language are presented.

- Chapter 4 - Monitoring

  From compilation to error detection, the whole process of monitoring is explained.

- Chapter 5 - Evaluation

  How the developed work was evaluated.

- Chapter 6 - Future Work

  The work left undone or possible improvements are presented.

- Chapter 7 - Conclusion

  A summary and opinion on the developed work.

Chapter 2

Background & Related Work

This chapter gives an overview of the background software adopted while developing this work (section 2.1), and shines 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). The research on runtime testing, the different techniques, and the difficulties of implementing it that already exist (subsection 2.1.3), and also, the importance of invariant specification and its relation to Linear Temporal Logic (LTL) (subsection 2.1.4).

2.1.1 Robot Operating System

The Robot Operating System (ROS) [14] 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. This abstraction helps developers easily connect the different robot components predominantly through messages sent through communication channels called *topics* via a publish-subscribe architecture. A *topic* decouples the production of information from its consumption, ROS nodes will subscribe and publish to relevant *topics* and not know who they are comunicating with. Every *topic* as a message type and can only receive messages of that type.

ROS has a modular architecture, is built with the purpose of cross-collaboration and easy development [4], is one hundred percent open source, available in multiple platforms, and ready for use across a wide array of robotics applications. For all these reasons, ROS is the norm for teaching robotics and is the basis for most robotics research, not only this but multiple companies rely on ROS for robotic development.

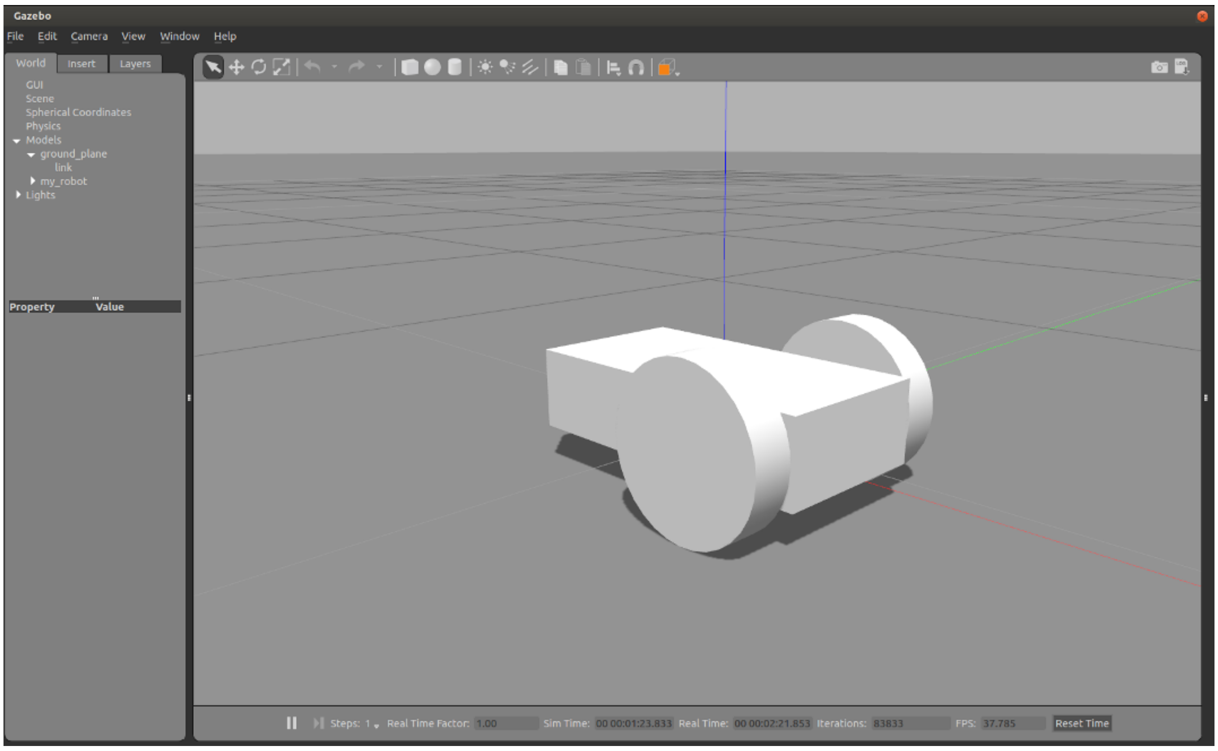
The ROS ecosystem ended up with a very strong «standard library» of packages that are used by almost everyone, and a large number of packages that don't belong to that standard library that are rarely used by anyone. This is generally seen as a bad thing in software ecosystems, but for this work, it is an advantage because it allows specifying the behavior of a smaller standard library and still be able to represent the behavior of a large number of ROS systems. Literature states that around eighty-two percent of ROS applications rely on packages released by a small subset of groups [12].

2.1.2 Gazebo

Robotic systems simulation is an essential tool for testing robots' behavior. For this reason, Gazebo [11] 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.

Gazebo was chosen as the simulator for the work to follow up the work done by Afsoon Afzal in GzScenic [7], an automatic scene generation for the Gazebo simulator. In this way, it would be possible to have a tool to perform automatic tests in multiple arbitrarily generated scenarios.



Steps: 1 Real Time Factor: 1.00 Sim Time: 00 00:01:23.833 Real Time: 00 00:02:21.853 Iterations: 83833 FPS: 37.785

Figure 2.1: Gazebo 7's interface.

2.1.3 Runtime Testing

Runtime Testing is an analysis that takes place during the execution of a program, taking advantage of information from the running system to make inferences on if the observed information violates certain properties of the program.

Due to the mentioned unforeseen circumstances 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 since the computer has to allocate resources to the monitoring software. Therefore, not demanding excessive resources is essential when taking this approach.

Stadler, Vierhauser and Cleland-Huang *"Towards flexible Runtime Monitoring Support for ROS-based Applications"* [15] pinpoint several challenges in developing runtime monitoring tools for robotic-based applications:

- *"Provisioning of an initial overview of the system structure."*
- *"Diversity and need to individually configure monitoring needs."*
- *"Only a subset of these properties likely need to be monitored on a continuous basis."*
- *"Data needs to be collected and analyzed differently."*
- *"Diverse types of constraints need to be defined and checked on the data."*
- *"Make the outcome of the runtime monitoring data and services accessible to the user."*

These challenges are relevant in choosing which direction to take when specifying the DSL or implementing the monitoring tool for my approach. For instance, while I want the DSL to be intuitive and easily learned, it should also be capable of expressing all monitoring needs and giving an overview of the system. Also, identifying what type of data we need to monitor influences how the checking of the properties is computed.

2.1.4 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 [16] demonstrates that important safety bugs in real-world autonomous robotic systems can be identified when representing safety violations of systems and monitoring them.

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 [13]. LTL can be used as a method of model-checking [8] using its patterns as a form of property specification. It includes patterns such as "always", "finally", "until", "eventually", and others, which can be used to define invariants for program verification of robotics systems.

2.2 Related Work

Other monitoring frameworks that have already tried to implement similar runtime verification concepts are presented in subsection 2.2.1.

2.2.1 Monitoring Frameworks

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

ROSMonitoring [9] can monitor and log errors at the level of *topic* (communication buses between the systems' components) malfunctioning, however, it was designed with portability and scalability in mind which means it is highly complex to use and can be not very user-friendly unless one is already knowledgeable of the subject.

ROSMonitoring also does not provide a ROS-oriented DSL, instead, it takes advantage of the already built and somewhat generic language Runtime Monitoring Language (RML) [2] which doesn't provide the expressiveness and intuitiveness expected from a ROS-oriented invariant specification language.

ROSRV [10] is another tool that provides the runtime monitoring of ROS systems, however, ROSRV implements safety measures while the system is running, interfering with its normal execution. ROSRV lacks documentation on its functioning and usage. Additionally, ROSRV seems to have been discontinued since its last version is only capable of running under ROS Groovy Galapagos, which has not been supported since 2014.

Chapter 3

A 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 (BNF) (section 3.2). BNF is a notation used to describe the grammar or syntax of computable languages. Finally, some practical examples are written with the help of the DSL to display its expressiveness (section 3.3).

3.1 Language Notations

This section presents the notations that are possible to express with the DSL. Some high-level concepts are explained in subsection 3.1.1. Temporal notations are enumerated in subsection 3.1.2 and subsection 3.1.3. Section 3.1.4 enumerates the primitives that can be used to related simulation and robot components. Section 3.1.5 and subsection 3.1.6 talk, respectively, about the operands and operators of the DSL. Section 3.1.7 shows the protected variables of the DSL. Section 3.1.8 talks about *topic* declaration in the DSL. Finally, subsection 3.1.9 talks about robot modeling in the DSL.

3.1.1 High-level Concepts

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.

  Ex. *after robot.position.x > 1, never robot.velocity > 5*

- **Declaration -** A declaration allows for the representation of ROS *topics* in order to interact with them.

  Ex. *decl robot_position /odom Odometry.pose.pose.position*

- **Model -** A model allows for the declaration of specific *topics* that are required when correlating certain robots' and simulation components.

  Ex. *model robot1: position /odom Odometry.pose.pose.position ;*

- **Association -** An association serves as a way to create program variables.

  Ex. *property1 = always robot.velocity > 1*

3.1.2 Temporal Keywords

The language considers 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;

Note: For the property "after X, always Y" the property "always Y" only has to hold after the first condition is met, this is what is meant by subsequent path.

3.1.3 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*.

3.1.4 Simulation primivitives

To support comparing the internal state of the robotic system with the environment, the language provides basic primitives 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;

3.1.5 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.

3.1.6 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 *A A{y}* - the values being compared will have an error margin of y, for instance (*Z =={0.05} Y*) is the similar to saying (*Y-0.05 < Z < Y+0.05*) and (*Z <{0.05} Y*) the same as (*Z < Y+0.05*).

3.1.7 Protected Variables

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

rate - Set the frame rate at 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

3.1.8 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.

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

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

3.1.9 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
```

The localization_error function requires the position topic of a robot to be modeled in order to compare it with the actual simulation position.

3.2 Grammar

The language grammar is written in BNF and is presented below.

The *<association>* production highlights a basic notation of the language and is related to the high-level concepts of the DSL mentioned in subsection 3.1.1.

```
<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, I present some 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 in section 4.1, then the generated file and some of its specifications are described in section 4.2.

4.1 Runtime Monitoring

After writing all the desired robotic systems specifications, the file needs to be compiled into a monitoring Python module. This is currently done running the following script, from its location: 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 takes as arguments:

1. The specifications file;
2. The path where the Python monitoring module will be generated.

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 Compilation

In this section, we start by giving an overview of what the generated code does while executing (subsection 4.2.1), next, we explain the data structures used in compilation and the translation rules to the generated code are explained in (subsection 4.2.2). We discuss how the various states of the simulation are saved in subsection 4.2.3. Finally, we discuss the generation of error messages in subsection 4.2.4.

4.2.1 Architecture

The diagram in Figure 4.1 tries to capture the essence of what the generated code is doing when running alongside a simulation of a robotic system. The topic where its fetched the real simulation data */gazebo/model_states*, and the multiple topics with the robot perception of the simulation. The diagram highlights the two parallel processes running in the node, the “Callback Function” and the “Monitoring Loop” and their job in the verification.

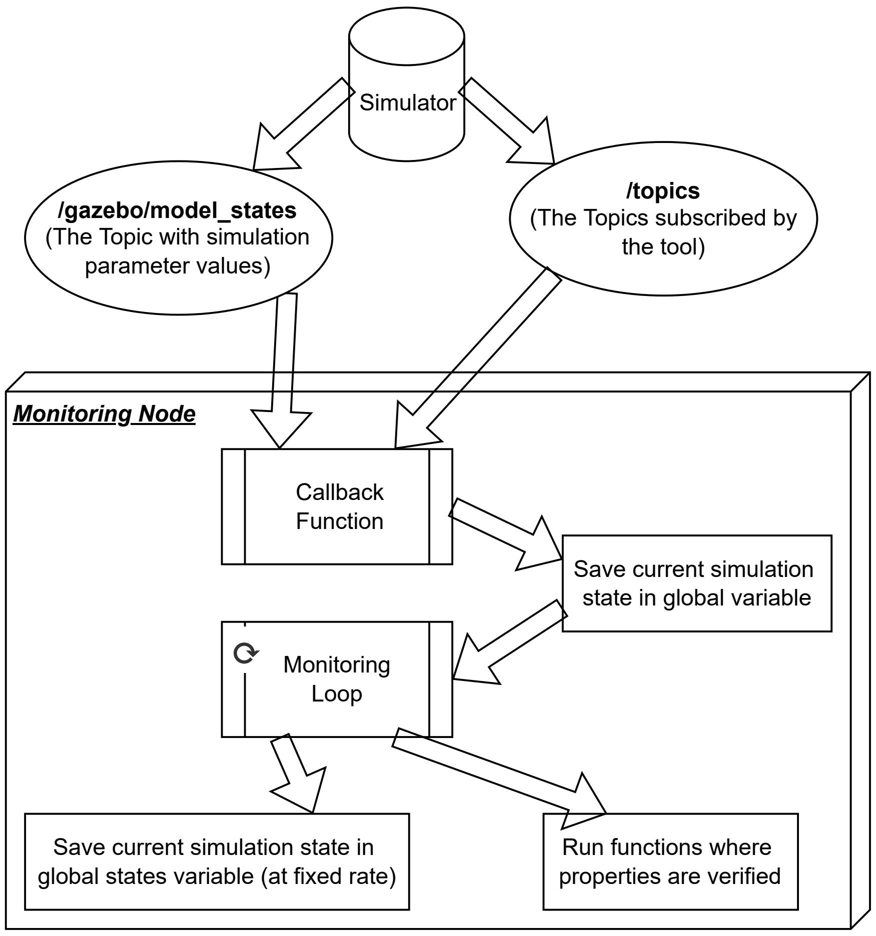

Simulator
/gazebo/model_states
(The Topic with simulation parameter values)
/topics
(The Topics subscribed by the tool)
Monitoring Node
Callback Function
Save current simulation state in global variable
Monitoring Loop
Save current simulation state in global states variable (at fixed rate)
Run functions where properties are verified

Figure 4.1: Diagram of the execution of the generated code.

The callback function is called every time a new message is received in one of the subscribed topics, it saves the relevant information for the property checking in a global variable and this information serves as the current “screenshot” of the simulation representing its current state.

The node executes a loop at a delineated rate, which is represented in the diagram as the “Monitoring Loop”, doing the following tasks:

1. Check if the defined simulation timeout time has been reached.

2. Save the current simulation state in a global variable of saved states.

3. Verify the properties using the saved states and calling each dedicated property function.

4.2.2 Code Generation

In order to generate the monitoring code, some context is saved when parsing through the abstract syntax tree created from the specification file.

The saved context is essentialy separated in five Dictionary lists: Models, Associations, Variables, Subscribers, and Properties.

The information saved in these data structures will afterward be used in translation rules between the source code to create the generated code.

Models

The data structure of each model saved in the context is composed of:

1. The name of the object in the simulation that is being modeled.
2. The correspondent function the model refers to, for example *velocity* or *distance*.
3. The Message type of the data structure of the *topic* to subscribe.

For each entry in the *Model* of an object, a model data structure will be saved in the context. A subscriber data structure will also be created that subscribes to the correspondent *topic* of the modeled object.

The subscriber and variable in the correspondent generated code that are associated with each entry of the Model, will have a unique name composed of the object name plus the function associated with that entry.

Associations

The data structure of each association saved in the context is composed of:

1. The name of the association in the specification.
2. The name of the variable associated with the association.

When specifying an association a new variable data structure is created in the context. The new variable has the prefix of an association as well as its name. In the generated code the new variable will point to the variable created on the right side of the association.

Variables

The data structure of each variable saved in the context is composed of:

1. The variable name to use in generated code.
2. The generated code representation of the expression to be used to fetch the data of the variable.

The variable names to be used in the generated code will have different suffixes and prefixes depending on: Their types, and if they originate from a model, declaration, function, or association. In this way, it is easier to identify variable relations during the compilation of the generated code.

All simulation primitives have a hardcoded representation to get their absolute value from the simulator *topic* information, which is stored in a global *state* variable (subsection 4.2.3). To retrieve the correct string for each primitive to be used in the generated code, an auxiliary function *sim_funcs* was created:

```
def sim_funcs(object_, func, args, ctx):
    """Add var to the context and return var name for the output file (Considering the function used)"""
    var_name, extract = None, None
    if func == "position":
        args = ["position"] + args
        var_name = object_ + "_" + "_".join(args) + "_var_sim"
        extract = (
            "model_states_msg.pose[model_states_indexes['"
            + object_
            + "']]."
            + ".".join(args)
        )
    elif func == "velocity":
        var_name = object_ + "_velocity_" + "_".join(args) + "_var_sim"
        if args == []:
            extract = (
                "((model_states_msg.twist[model_states_indexes['"
                + object_
                + "']].linear.x)**2 + (model_states_msg.twist[model_states_indexes['"
                + object_
                + "']].linear.y)**2 + (model_states_msg.twist[model_states_indexes['"
                + object_
                + "']].linear.z)**2"
                + ")**(0.5)"
            )
        else:
            extract = (
                "model_states_msg.twist[model_states_indexes['"
                + object_
                + "']]."
                + ".".join(args)
            )
    elif func == "localization_error":
        var_name = object_ + "_localization_error"
        args = ctx.model_msgtype(object_, "position")
        extract = (
            "((model_states_msg.pose[model_states_indexes['"
            + object_
            + "']].position.x - "
            + object_
            + "_position_msg."
            + args
            + ".x)**2 + (model_states_msg.pose[model_states_indexes['"
            + object_
            + "']].position.y - "
            + object_
            + "_position_msg."
            + args
            + ".y)**2 + (model_states_msg.pose[model_states_indexes['"
            + object_
            + "']].position.z - "
            + object_
            + "_position_msg."
            + args
            + ".z)**2)**(0.5)"
        )
    elif func == "distanceZ":
        object2 = args[0]
        var_name = object_ + "_" + object2 + "_distanceZ"
        extract = (
            "((model_states_msg.pose[model_states_indexes['"
            + object_
            + "']].position.x - model_states_msg.pose[model_states_indexes['"
            + object2
            + "']].position.x)**2 + (model_states_msg.pose[model_states_indexes['"
            + object_
            + "']].position.y -"
            + "model_states_msg.pose[model_states_indexes['"
            + object2
            + "']].position.y)**2 + (model_states_msg.pose[model_states_indexes['"
            + object_
            + "']].position.z - model_states_msg.pose[model_states_indexes['"
            + object2
            + "']].position.z)**2)**(0.5)"
        )
    elif func == "distance":
        object2 = args[0]
        var_name = object_ + "_" + object2 + "_distance"
        extract = (
            "((model_states_msg.pose[model_states_indexes['"
            + object_
            + "']].position.x - model_states_msg.pose[model_states_indexes['"
            + object2
            + "']].position.x)**2 + (model_states_msg.pose[model_states_indexes['"
            + object_
            + "']].position.y -"
            + "model_states_msg.pose[model_states_indexes['"
            + object2
            + "']].position.y)**2)**(0.5)"
        )
    ctx.add_var(var_name, extract)
    return "states[0]['" + var_name + "']"
```

For instance, if the compilation is processing the primitive *robot1.distance.robot2* that relates the distance between two robots, this function will compute the variable named *robot1_robot2_distance* and how to fetch its value during execution:

```
robot1_robot2_distance = ((model_states_msg.pose[model_states_indexes'['robot1]].position.x -
    model_states_msg.pose[model_states_indexes'['robot2]].position.x)**2 +(model_states_msg.pose[
    model_states_indexes'['robot1]].position.y - model_states_msg.pose[model_states_indexes'['robot2]].
    position.y)**2)**(0.5)
```

Subscribers

The data structure of each subscriber saved in the context is composed of:

1. The *Topic* name to which to subscribe.
2. The Message type associated with the *topic*.
3. The Library from which the Message type originates.
4. The Subscriber name in the generated code.

A subscriber relates in the specification to a *declaration*, a *model* entry, or the default simulator information *topic*.

The Library is necessary in order to make an import of the Message type in the generated code. The Library is obtained through a python subprocess:

```
command = f"cd {self.filepath} | rosmsg show {msg_type}"
```

The generated monitoring file declares the needed subscribers and uses ApproximateTimeSynchronizer, from the *message_filters* package, to call the callback function. The ApproximateTimeSynchronizer synchronizes messages by their timestamp and if they do not have a header, uses the ROS time.

Properties

The data structure of each property saved in the context is composed of:

1. The correspondent line in the specification file.
2. The correspondent type of property.
3. A List of Strings with the necessary code representation of boolean assertions to be used in the generated code as a property verification.

A property is represented by one base property, which is the one at the first level of the specification. In the next example, the base property saved in the context will be the "after". The "never" property will only influence the way the property is checked on the generated code and no data structure of it will be saved in the context.

```
after robot.distance.wall < 1, never robot.velocity > 0.5
```

A property, in order of complexity, is comprised of: Other properties, conjunctions (*and*, *or*, and *implies*), comparisons (<, >, <=, >=, ==, and *!=*), operations (+, -, *, and /), and operands (variable, number, boolean, simulation primitive, and temporal value).

The abstract syntax tree is parsed from the lowest to the highest complexity field, in this way the List of strings that represent boolean assertions in the generated code can be cumulatively built because the information from the lowest fields is always present in the higher ones. In the end, a List with the necessary comparisons is built for a base property.

In the generated code an independent function with the necessary boolean assertions for verifying the property will then be defined for each base property.

4.2.3 State

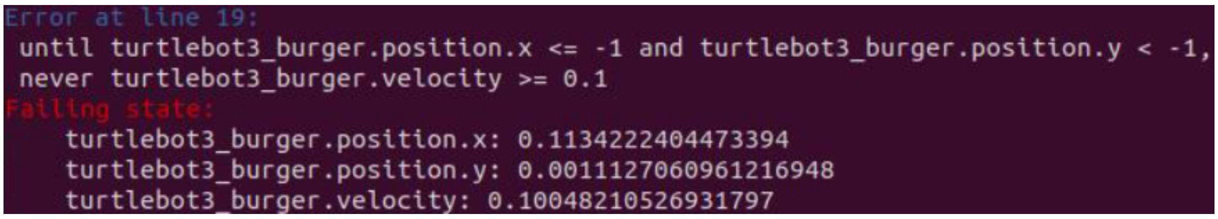
A callback function is called every time a new message from one of the subscribers is received. The callback function saves the relevant information for property checking in a global variable as a Dictionary. This information serves as a current "screenshot" of the simulation representing its current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

The callback function is called at fluctuating rates, for this reason, we take advantage of the loop that is running at a fixed rate to save multiple current states of the simulation in another global variable as a Dictionary List. This variable will be the one used when checking the properties to make correlations with past states.

4.2.4 Error Messages

An error message starts by stating the line in the specification file which resulted in an error as well as showing the property where the error originated.

Afterward, I use the current saved state of the simulation to show the values at the time of failure of all the variables present in the property that originated the error.



```
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 4.2: Example of an error message.

Chapter 5

Evaluation

This chapter introduces the evaluation process of the work. Section 5.1 gives a broad overview of the whole process, and section 5.2 goes into more detail about each one of the experiments.

5.1 Evaluation Overview

To evaluate the developed work, I decided to go through a list of already documented ROS bugs and identify three that happen at runtime. After that, I specified a robot's properties in the DSL that should be capable of detecting an error for said bugs while running the system.

ROBUST [3] is a dataset that documents over two hundred bugs in multiple robots using ROS. After going through the dataset in an initial skim, three bugs that happened at runtime and didn't halt the robot execution were identified:

- Calculation Error Inverts Turning Direction [1] (subsection 5.2.1) - "Due to an error in velocity calculations, when Kobuki was issued a very low negative linear speed (very slow backwards movement), it would also inadvertently invert its turning direction. That is, if it was supposed to move backwards while turning left, it would move backwards and turn right instead."

- Robot Getting Stuck When Auto-docking [2] (subsection 5.2.2) - "The movement speeds were hard-coded for the auto-docking algorithm, and worked well for regular Kobuki and Turtlebot, but were too slow for heavier robots, causing them to get stuck."

- Unexpected Movement Due to Wrong Calculation [3] (subsection 5.2.3) - "Kobuki moves using differential drive. Originally, the command velocities (linear and angular) were provided as 'short', and were converted to 'short' after each step, even though the calculations yielded floating point numbers. This lead to calculation errors in some special cases, where the robot was supposed to move forward but ended up moving backwards instead."

To replicate each bug, a *daemon* node is inserted into the system that interferes with the normal runtime flow and replicates the desired bug. Figure 5.1 represents the natural flow of

[1] https://github.com/robust-rosin/robust/blob/master/kobuki/e964bbb/e964bbb.bug
[2] https://github.com/robust-rosin/robust/blob/master/kobuki/0416c81/0416c81.bug
[3] https://github.com/robust-rosin/robust/blob/master/kobuki/1c141a5/1c141a5.bug

the systems whilst Figure 5.2 shows the system flow when trying to replicate a bug with the help of the *daemon* node.

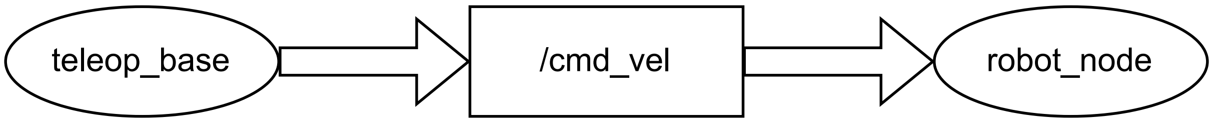


Figure 5.1: The normal runtime flow of the system.

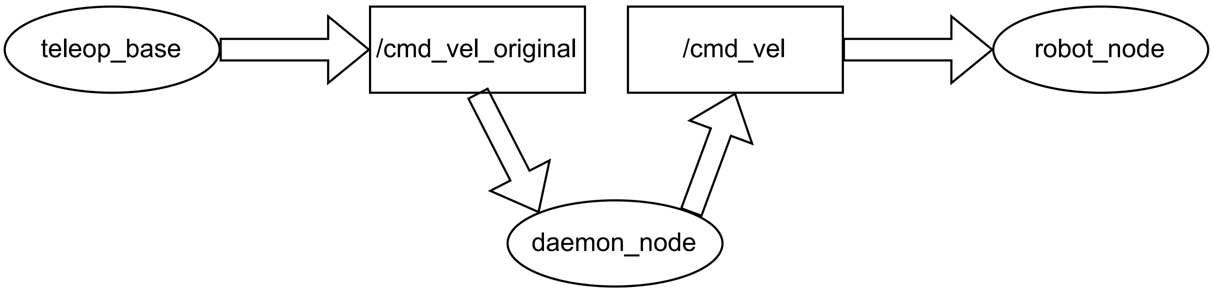


Figure 5.2: The flow of the system when the adding the daemon node.

When injecting a bug, all the data before addressed to the robots' */cmd_vel* topic is remapped to a new topic called */cmd_vel_original*, the *daemon* node subscribes to this new topic and modifies the data before sending it to the robots' */cmd_vel* topic so that the robot behaves like the expected bug.

With this new system flow, it is possible to specify properties between the */cmd_vel* topic which represents the robots' behavior, and the */cmd_vel_original* topic, which represents the actual command given to the robot.

The robots' */cmd_vel* topic expects to receive messages of type *Twist*, a data structure composed of two *Vector3* objects, that express velocity in its linear and angular parts, for this reason, our */cmd_vel_original* will also need to receive and send messages of the *Twist* type.

5.2 Experiments

This section goes through the property specification and runtime monitoring for the three mentioned selected bugs. Calculation Error Inverts Turning Direction (subsection 5.2.1), Robot Getting Stuck When Auto-docking(subsection 5.2.2), and Unexpected Movement Due to Wrong Calculation (subsection 5.2.3).

5.2.1 Calculation Error Inverts Turning Direction

DSL property specification:

First, I declare the cmd_vel_original topic, which will represent the commands given to the robot. Then I make a correlation between the topic that represents the given commands and the topic that represents the actual robot's behavior.

decl angular_vel_robot_perception cmd_vel_original Twist.angular.z

after turtlebot3.velocity.angular.z < 0, never angular_vel_robot_perception > 0

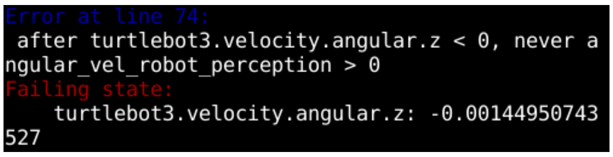
Translating into natural language, the property states in the first section that after the robot's actual simulation z parameter of the angular velocity is less than zero, then in the second section, the z parameter of the angular velocity of the command given to the robot is never more than zero.

Daemon node code and behavior:

```python
class Direction_invert_error:

    def ___init___(self):
        print("simulating direction_invert_error_behavior...")
        self.cmd_vel_pub = rospy.Publisher("cmd_vel", Twist, queue_size=1)
        self.twist = Twist()
        self.direction_invert_error()

    def get_vel(self):
        return rospy.wait_for_message("cmd_vel_original", Twist)

    def direction_invert_error(self):
        while not rospy.is_shutdown():
            vel = self.get_vel()
            self.twist = vel
            if abs(vel.linear.x) < 0.012:
                self.twist.angular.z = -vel.angular.z
            self.cmd_vel_pub.publish(self.twist)
```

The *daemon* node checks when the given robot's command linear velocity is below *0.012* and injects the opposite value of the z parameter of the angular velocity to the actual robot's velocity in the simulation.

Now when giving commands to the robot, if the given velocity is below *0.012*, a value chosen at random to start emulating the behavior, and I make a turn, the robot will turn the opposite way. The output of the monitoring node for a test case is demonstrated in Figure 5.3.



```
Error at line 74:
 after turtlebot3.velocity.angular.z < 0, never a
ngular_vel_robot_perception > 0
Failing state:
    turtlebot3.velocity.angular.z: -0.00144950743
527
```

Figure 5.3: Calculation Error Inverts Turning Direction bug error message.

5.2.2 Robot Getting Stuck When Auto-docking

DSL property specification:

First, I declare the cmd_vel_original topic, which will represent the commands given to the robot. Then I make a correlation between the topic that represents the given commands and the topic that represents the actual robot's behavior.

decl vel_robot_perception cmd_vel_original Twist.linear.x

after vel_robot_perception > 0, never turtlebot3.velocity =={0.005} 0

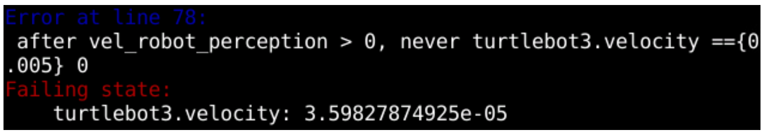
Translating into natural language, the property states in the first section that after the given robot's commands x parameter of the linear velocity is greater than zero, then in the second *section, the actual simulation linear velocity is never equal to the interval -0.005 to 0.005, which is roughly zero.*

Daemon node code and behavior:

```
class Auto_docking_error:

    def __init__(self):
        print("simulating auto_docking_error_behavior...")
        self.cmd_vel_pub = rospy.Publisher("cmd_vel", Twist, queue_size=1)
        self.twist = Twist()
        self.auto_docking_error()

    def get_vel(self):
        return rospy.wait_for_message("cmd_vel_original", Twist)

    def auto_docking_error(self):
        while not rospy.is_shutdown():
            vel = self.get_vel()
            self.twist = vel
            if abs(vel.linear.x) < 0.015:
                self.twist.linear.x = 0.0
            self.cmd_vel_pub.publish(self.twist)
```

The *daemon* node checks when the given robot's command linear velocity is below *0.015* and injects a value of *0.0* to the linear velocity of the actual robot's velocity in the simulation.

Now when giving commands to the robot, if the given velocity is below *0.015*, a value chosen at random to start emulating the behavior, the robot will stay stationary. The output of the monitoring node for a test case is demonstrated in Figure 5.4.



```
Error at line 78:
 after vel_robot_perception > 0, never turtlebot3.velocity =={0
.005} 0
Failing state:
    turtlebot3.velocity: 3.59827874925e-05
```

Figure 5.4: Robot Getting Stuck When Auto-docking bug error message.

5.2.3 Unexpected Movement Due to Wrong Calculation

DSL property specification:

First, I declare the cmd_vel_original topic, which will represent the commands given to the robot. Then I make a correlation between the topic that represents the given commands and the topic that represents the actual robot's behavior.

decl vel_robot_perception cmd_vel_original Twist.linear.x

after turtlebot3.velocity.linear.x < 0, never vel_robot_perception > 0

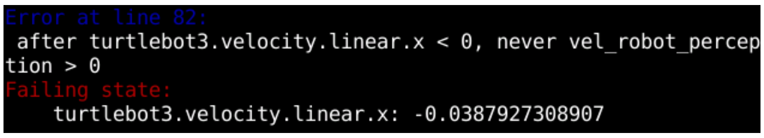
Translating into natural language, the property states in the first section that after the robot's actual simulation x parameter of the linear velocity is less than zero, then in the second section, the x parameter of the linear velocity of the command given to the robot is never more than zero.

Daemon node code and behavior:

```
class Backwards_movement_error:

    def __init__(self):
        print("simulating backwards_movement_error_behavior...")
        self.cmd_vel_pub = rospy.Publisher("cmd_vel", Twist, queue_size=1)
        self.twist = Twist()
        self.backwards_movement_error()

    def get_vel(self):
        return rospy.wait_for_message("cmd_vel_original", Twist)

    def backwards_movement_error(self):
        while not rospy.is_shutdown():
            vel = self.get_vel()
            self.twist = vel
            if abs(vel.linear.x) > 0.03:
                self.twist.linear.x = -vel.linear.x
            self.cmd_vel_pub.publish(self.twist)
```

The *daemon* node checks when the given robot's command linear velocity is above *0.03*, a value chosen at random to start emulating the behavior, and injects the opposite value of the x parameter of the linear velocity to the actual robot's velocity in the simulation.

Now when giving commands to the robot, if the given velocity is above *0.03*, the robot will start moving backward. The output of the monitoring node for a test case is demonstrated in Figure 5.5.



```
Error at line 82:
 after turtlebot3.velocity.linear.x < 0, never vel_robot_percep
tion > 0
Failing state:
    turtlebot3.velocity.linear.x: -0.0387927308907
```

Figure 5.5: Unexpected Movement Due to Wrong Calculation bug error message.

Chapter 6

Future Work

In this chapter, the possible work left undone or that could improve our study is presented. Improvement of the developed work performance is mentioned in section 6.1, section 6.2 mentions the validation of the work, section 6.3 discusses about the work's error messages, section 6.4 talks about the integration with scenario generation tools, section 6.5 mentions the possible integration with other simulators besides Gazebo, and finally section 6.6 talks about a more advanced quality assurance tool.

6.1 Performance Tweaking

The generated code performance can be improved so that the load of the monitoring node on the simulation is reduced.

For instance, the frequency at which some properties are checked could fluctuate. In some circumstances, a particular property does not need to be checked at every simulation iteration. Implementing some mechanism that can skip certain property checks per iteration will undoubtedly decrease the load the monitoring node will have on the simulation.

6.2 Validation of the Proposal

Although some evaluation was done for the work done, more evidence and experimental data on the effective capabilities of the proposal are still needed:

1. How expressive is the DSL from the developers' point of view in specifying robots' properties.
2. Proof of concept that the system is able to detect the rule violations specified by the DSL.
3. Evidence that the monitoring does not disturb the simulation by demanding excessive resources.

6.3 Better Error Messages

Giving developers helpful and comprehensible error messages is a shared concern amongst all compilers. One can argue that even the best compilers still have space for improvement when talking about delivering good error messages.

Although in this work I gave some thought to the error messages delivery, I believe that a more narrow error localization is still possible. For instance, in cases where a property has assertions for a time interval, like an *after_until* or an *eventually*, maybe not only the value at the time of failure of the variables present in the specification should be shown but also a selection of the values at multiple previous states of the simulation.

Also, there was no time for a thorough validation of the proposal, which means that some bugs could be present when delivering the error messages, and the users could have some problems with the delivery or ideas on how to improve it.

6.4 Integrate the DSL with Scenario Generation Tools

Integrating this work with a scenario generation tool would improve the whole test automation process by creating unpredictable environments on where to test our specified properties, allowing the execution of multiple tests.

For instance, GzScenic [7] is a tool that generates random scenarios for the Gazebo simulator based on a defined model.

6.5 Integration with other Industrial Simulators

This work was developed with the Gazebo simulator in mind, which means the monitoring is currently not compatible with other simulators.

Although Gazebo is widely used, many other simulators are also currently being used, like RoboDK [1], Webots [6] or Unity [5]. Therefore, adapting our work to integrate other widely used simulators would be helpful for many users that choose not to use Gazebo.

6.6 Automatic Quality Assurance

This work can be used to measure the quality of software modules. 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. Thus, it can be used in automated program synthesis, repair, and improvement.

Chapter 7

Conclusion

Due to the fact that robots interact with the real world, robotic systems are unpredictable. Coming up with a reliable and efficient method for automatic robot testing is a challenge, one of the reasons being that verifying the success of a task may not be possible from the robot's perspective.

My approach takes advantage of simulation software to perform a type of external monitoring in order to achieve automatic monitoring of the robotic system.

My approach relies on simulation-based testing so that developers can take advantage of the real values of objects' attributes in the simulation to compare with what the robot system perceives, trying in this way to surpass the need for human-in-the-loop testing.

I succeeded in developing a DSL that allows for the specification of a ROS robotic system's properties and that abstracts an underlying LTL system. It is possible to express relevant temporal and positional arguments between robots' components and objects in the simulation, and also properties that relate the internal information of the system with the corresponding information in the Gazebo simulator.

Although better validation by developers on the expressiveness of the DSL is still needed, I believe the developed DSL provides an easy-going, intuitive and in-depth way for both experts and non-experts to specify properties for robotic systems.

I have also succeeded in developing a tool for the generation of automatic monitoring software that can monitor some behavioral violations of robots in relation to their state or events during a Gazebo simulation. At the same time, I have shown that the approach's generated monitoring software can monitor some interesting scenarios that developers care about.

Bibliography

[1] RoboDK. https://robodk.com/. [Online; accessed 22-September-2022].

[2] Runtime Monitoring Language. https://rmlatdibris.github.io/. [Online; accessed 13-September-2022].

[3] ROBUST: ROS Bug Study. https://github.com/robust-rosin/robust. [Online; accessed 28-August-2022].

[4] ROS-INDUSTRIAL. https://rosindustrial.org/. [Online; accessed 11-September-2022].

[5] Unity. https://unity.com/. [Online; accessed 22-September-2022].

[6] Webots. https://www.cyberbotics.com/. [Online; accessed 22-September-2022].

[7] Afsoon Afzal, Claire Le Goues, and Christopher S. Timperley. GzScenic: Automatic Scene Generation for Gazebo Simulator, 2021.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.