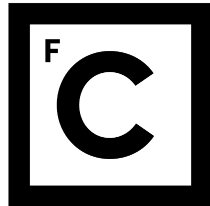
UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA



Ciências
ULisboa

RobotFix : Detecting Bugs On Variables In Robot Programs

Miguel Rodrigues Tavares

MESTRADO EM ENGENHARIA INFORMÁTICA

Dissertação orientada por:
Prof. Doutor Alcides Miguel Cachulo Aguiar Fonseca
e co-orientado pelo Prof. Doutor Christopher Steven Timperley

2022

Agradecimentos

O final desta etapa da minha vida que também representa o final da minha vida académica. Antes de tudo tenho de agradecer aos meus pais por ter tido esta oportunidade na minha vida, e por me terem apoiado em tudo e sempre quererem o melhor para mim. Além dos meus pais também tive muito apoio do resto da minha família que ao longo dos anos sempre se preocupavam como estava a correr os meus estudo e tese.

Depois queria agradecer ao Professor Alcides, que desde o primeiro ano de faculdade me deu aulas e que neste ano foi o meu orientador. Obrigado por nos momentos em que tinhas dúvidas achava que era demasiado difícil e que não ia conseguir mostrar que as coisas eram simples. Além disso, foi muito importante para mim toda a liberdade que me deu e ter sempre algo para fazer ao longo do ano. Também gostaria de agradecer ao Chris por ser meu coorientador e por toda a ajuda, e todas as ideias que deu ao longo do ano (Thank you Chris!).

Durante este ano também tive muita ajuda do Paulo, a quem tenho de agradecer por toda a paciência e preocupação que teve comigo. A Catarina, que mesmo não gostando de robôs, sempre teve disponível para me ajudar e me explicar as coisas da melhor forma que ela conseguia. Por fim, tenho de agradecer ao Guilherme por todos os "abre olhos" que deu ao longo do ano. Sem eles o meu trabalho não teria tanta qualidade e este ano seria muito mais frustrante.

Também tenho de agradecer aos meus amigos de Torres Vedras, por me acompanharem há muitos anos, e que embora às vezes me distraem um pouco dos meus deveres quando tinha de ir trabalhar, mas que também foram importantes para eu sair do mundo da tese e me divertir.

Finalmente quero agradecer a todas as pessoas que conheci durante este e os últimos anos na universidade e que fizeram toda esta experiência muito melhor e memorável.

À minha família e amigos

Resumo

Atualmente os sistemas robóticos são muito usados na nossa sociedade, podendo ser encontrados em diversos lugares, tais como fábricas ou cirurgias. Por outro lado, é desafiador programar sistemas robóticos e além disso é necessária uma vasta experiência em programar robôs e em robótica no geral. Adicionalmente, erros em sistemas robóticos podem conduzir a elevados custos.

Entre diversas frameworks para programar robôs, decidi trabalhar com a framework Robot Operating System visto que possui um extenso número de *packages*, componentes e simuladores.

Robot Operating System (ROS) é uma framework flexível para escrever código para robôs. ROS é usado tanto em pequenos projetos como em grandes projetos. Um exemplo de um projeto que usa ROS são as simulações para exploração de robôs na lua.

Contudo é fácil introduzir erros a programar em ROS. Por exemplo, os programadores podem confundir a unidade de variáveis entre “m/s” e “km/h”, ou podem nem saber a unidade do valor que introduziram para o robô. Além de erros que acontecem em todas as execuções, também podem acontecer erros que acontecem em pequenas percentagens de execuções e que são mais difíceis de encontrar.

Internamente, os nós do robô comunicam por mensagens que não têm limites definidos pelo compilador do ROS, sendo que os programadores podem introduzir valores inesperados nestas mensagens que vão conduzir a comportamentos errados por parte do robô. Por exemplo, numa variável de uma mensagem um programadores poderá definir os segundos do comportamento com um valor negativo sem levantar erros.

Erros nos programas robóticos podem levar a demasiados custos económicos para as empresas, por exemplo um bug num robô pode fazer com que uma fábrica pare até que o bug seja resolvido. Um exemplo real é o acidente da orbita climática em Marte em 1999 [53]. Por este motivo, devem ser criadas maneiras de evitar erros em robôs, especialmente estaticamente, ou seja, sem colocar o robô a correr o programa, visto que não é fácil reproduzir com fidedignidade o ambiente do mundo real assim como não é fácil fazer testes no mundo real. Por outro lado, programadores inexperientes em robótica tendem a superestimar as suas habilidades, cometendo muitos erros. Deste modo necessitam de ajuda enquanto programam.

Foi criado o RobotFix, que pretende encontrar erros estaticamente nas variáveis que os programadores usam, em relação à sua unidade e à deteção de valores inesperados. Deste modo, pretendo que os programadores que usem RobotFix estejam seguros de que as suas variáveis estão a ser usados do modo esperado e correto. Para esse efeito, construi uma *Domain Specific Language (DSL)* que suporta uma ferramenta de análise estática que deteta um subconjunto de erros em programas em ROS.

Estudos existentes sobre ferramentas que identificam as unidades das variáveis em ROS salientam que estas ferramentas possuem falhas e também afirmam que se os programadores definissem as unidades através de anotações teriam melhores resultados. Além disso, apontam que a junção das duas formas de encontrar erros, ou seja, os programadores definirem as unidades com ajuda de uma ferramenta que dá sugestões seria o ideal.

Outras ferramentas feitas para o ROS não pretendem resolver os mesmos erros que o RobotFix e nenhuma usou anotações feitas pelos programadores para detetar erros em ROS, sendo esta uma nova abordagem que busca ser mais eficiente que as restantes.

Além disso existem outras ferramentas que usam Liquid Types com sucesso, algo que foi implementado no RobotFix como forma de criar predicados lógicos sobre variáveis de modo a verificar propriedades semânticas do código. Por exemplo, com LiquidTypes é possível fazer anotações como "x > 10" numa variável de modo a verificar se existem infrações à anotação ao longo do código.

O RobotFix pretende usar os LiquidTypes para fazer restrições nas variáveis, deste modo os programadores podem fazer anotações usando o *Annotated* que é importado do *typing_extensions*. Com o *Annotated* os programadores podem criar anotações que vão ser lidas pelo RobotFix, como por exemplo, `x : Annotated[float,"Unit("m/s") and x < 20 and x > -20"]`. Nesta linha a variável x é atribuída com a unidade "m/s" e com a anotação de que deve conter valores entre 20 e -20. Ao longo do programa vai ser verificado se a variável x é usada com outras variáveis de diferentes unidades ou se é atribuída com valores fora do intervalo designado pelo programador.

Com o RobotFix, os programadores podem criar anotações nas variáveis normais nos ficheiros Python, ou nas mensagens do ROS, que vão ser utilizadas por todas as mensagens criadas em código Python com o aquele tipo de mensagem.

Foi desenvolvida uma gramática para a criação de anotações por parte dos programadores, em que nas mensagens do ROS são criados comentários com "#RobotFix#" (identifica que é um comentário para ser lido pelo RobotFix) à frente do texto, e em Python o RobotFix vai ler as anotações, mas desta vez através do *Annotated*. Contudo esta gramática é muito semelhante ao código Python simplificando o processo de aprendizagem dos utilizadores em relação ao RobotFix.

Para fazer o parse do código em Python, foi usada a ferramenta Pytype, que já fazia o parse do código Python para fazer análises sobre os tipos das variáveis, utilizei também esta funcionalidade do Pytype para manter o rastreamento dos tipos das variáveis que os

programadores vão usar. O Pytype também foi utilizado para ler anotações dentro do código Python. Foi preciso fazer modificações no código do Pytype para ser usado com código específico do ROS, que muitas vezes é mais complexo do que o esperado pelo Pytype.

Para fazer o parse das mensagens (são criadas em ficheiros específicos do ROS) foram usadas funções já criadas pelo Roswire (que é uma biblioteca responsável por fazer análise estática e dinâmica para o ROS) e para as anotações nas mensagens foi criada uma linguagem com o Lark á qual se fez o parse.

Com as informações adquiridas através do código dos programadores são criados dois contextos, um para as variáveis e outro para as mensagens, estes contextos servem para guardar informações do código e criar funções que vão ser usadas para criar os inputs para o SMT-Solver Z3.

Com os contextos, o Robotfix vai construir os inputs para o Z3 SMT-Solver que recebe as informações do código dos programadores e procura inconsistências entre as informações. Por exemplo, o SMT-Solver vai receber informações como "var_unit(x) == "m/s"" ou "var_value(x) > 20" sendo que com estas informações, o Z3 vai procurar informações contraditórias. Se forem encontradas informações contraditórias, será devolvido ao utilizador uma mensagem de erro com as informações que levaram à contradição encontrada pelo SMT-Solver.

Para avaliar o RobotFix foram identificados bugs feitos por programadores de ROS, estes bugs vieram de perguntas no ROS Answers, onde programadores podem colocar as suas dúvidas sobre ROS, encontrados pelo Phys [39] (que encontra erros com inconsistências entre unidades) ou pelo ROBUST [27] (biblioteca com bugs em ROS), através destes bugs foi criado um teste simples inspirado no bug com o mesmo erro. Com estes testes, pretendeu-se demonstrar que abordagem implementada no RobotFix é útil para resolver bugs em programas com sistemas robóticos. Os resultados foram muito positivos, sendo que foi possível encontrar os bugs em 20 testes, não ocorrendo falsos positivos nos testes em que o bug foi corrigido.

No futuro existe espaço para continuar a evoluir o RobotFix. Algumas direções que a ferramenta pode seguir no futuro é construir um parser próprio para Python e para C (linguagem que também pode ser utilizada para programar em ROS). Adicionar um plugin para IDE que dá sugestões das unidades e anotações que o programador pode fazer. Além disso é possível adicionar mais funcionalidades, tanto as que já existem na linguagem como por exemplo ser possível criar as próprias funções como as de definir a unidade.

Palavras-chave: Robot Operating System, Sistemas Robóticos, Domain Specific Language, Verificação estática, Sistemas de tipos

Abstract

Robotic systems are everywhere (e.g., factories or medical surgeries) and are widely used with many real-world applications. However, programming robot systems is challenging, requiring domain-specific expertise.

The Robot Operating System (ROS) is a flexible framework for writing software for large-scale robots, targeting a wide range of developers, from students to industry professionals. With ROS, developers can program robots to follow complex behaviors. However, due to its specificities, it is easy to introduce bugs in ROS projects through a typo or making wrong assignments to variables regarding their unit or objective. Developers need to detect these and then fix them, which takes time and reduces their productivity and motivation. Furthermore, there is a lack of tools that help in detecting and fixing bugs, as well as verifying the absence of domain-specific errors. Additionally, existing tools do not provide the desired assurance on unit values and units. This work aims to improve the robot programming process and help users make fewer mistakes while programming.

This work created RobotFix tool to help programmers in finding errors related to variables. With RobotFix, developers can annotate the variable unit and intervals that variable value should respect. With our approach, developers will know that they are using the wrong unit (e.g., adding *km/h* and *m/s*) or a value that does not respect certain conditions (e.g., a programmer introducing negative seconds into a variable).

This approach has the advantage of providing feedback about the correctness of the code before executing, saving time and cost in testing the robot both in simulation or in the field. Testing in robotics has its disadvantages, like the shortcomings of simulators and the infeasible cost of testing all combinations of inputs. The approach was implemented into a prototype called RobotFix. Then, I evaluated the approach on programs related to 20 bugs made by ROS programmers. RobotFix succeeded in finding the bugs and showed that it could be a helpful tool for robotic programmers in the future.

Keywords: Robot Operating System, Robotic Systems, Domain Specific Language, Static Verification, Type System

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Objectives 3
1.3 Contributions 3
1.4 Structure of the document 3

2 Background & Related Work **5**
2.1 Robot Operating System 5
2.1.1 ROS Supported Languages 5
2.1.2 Nodes and Topics 6
2.1.3 ROS Messages 8
2.1.4 ROS Packages 9
2.2 Related Work 10
2.2.1 Robot Programming Studies 10
2.2.2 Modeling and Verification Tools for Robotics Systems 12
2.2.3 Type Annotations 15
2.2.4 Type annotations tools with LiquidTypes 15

3 Approach **17**
3.1 Liquid Types in Python 17
3.2 Base Example 17
3.3 System Requirements 20
3.4 System Design 21
3.5 Refinable Targets 21

4 Implementation **25**
4.1 Parsing 25
4.1.1 Grammar 25
4.1.2 Pytype 28
4.1.3 Python Code Parse 29
4.1.4 ROS Message Parsing 30

4.2 Building Contexts . . . 31
4.2.1 Messages Context . . . 31
4.2.2 Variable Context . . . 33
4.2.3 Building SMT Solver Inputs . . . 33
4.3 Verification . . . 34
4.4 Errors . . . 36
4.5 Current Limitations of Implementation . . . 38

5 Evaluation **39**
5.1 Validity . . . 39
5.2 Test Examples . . . 40
5.2.1 Test 15 - Timeout value is too long . . . 41
5.2.2 Test 16 - Calculations lead to a unexpected value . . . 42
5.3 Discussion . . . 42
5.4 Conclusion . . . 45

6 Future Work **47**
6.1 Create a parser for Python and C++ . . . 47
6.2 Implement other Python functionalities . . . 47
6.3 Modeling unit transformations . . . 47
6.4 Let developers create their functions . . . 48
6.5 Create suggestions for developers while they program . . . 48
6.6 Improve error messages . . . 48

7 Conclusion **49**

A Test Description **51**
A.1 Test 1 - Defining limits in the robot velocity . . . 51
A.2 Test 2 - Defining the variable units on Twist Message. . . . 51
A.3 Test 3 - Assignment between two variables with different units . . . 55
A.4 Test 4 - Calculations lead to an unexpected value . . . 55
A.5 Test 5 - Variable assigned with a unexpected value . . . 57
A.6 Test 6 - Different values for PI . . . 59
A.7 Test 7 - Variable assigned with a negative value for seconds . . . 59
A.8 Test 8 - Variable was not initialized . . . 61
A.9 Test 9 - Unexpected range value assigned to one variable . . . 61
A.10 Test 10 - Assignment between variables with different type of frames . . . 64
A.11 Test 11 - Assignment between variables with different types . . . 65
A.12 Test 12 - Assignment between variables with different type of frames . . . 66
A.13 Test 13 - Two variables should have different values . . . 66

A.14 Test 14 - Variable with unexpected value 67
A.14.1 Test 15 - Timeout value is too long 69
A.14.2 Test 16 - Calculations lead to a unexpected value 69
A.15 Test 17 - Variable not being initialized 70
A.16 Test 18 - Calculations lead to a unexpected value 72
A.17 Test 19 - Calculations lead to a unexpected value 72
A.18 Test 20 - ROS Message variable used but not initiliazed 74

Abbreviations **79**

Bibliography **86**

Index **87**

List of Figures

2.1 Nodes, Messages and Topics in ROS environment. 7

3.1 Architecture of RobotFix. 22

4.1 Complete architecture of RobotFix. 26
4.2 Architecture of RobotFix - Parse. 27
4.3 Grammar for the Annotations Language. 28
4.4 Architecture of RobotFix - Context and SMT Solver Inputs. 32
4.5 Architecture of RobotFix - Verification. 35
4.6 Architecture of the ROS Verification System - Errors. 37

5.1 Test 16 - Calculations lead to a unexpected value : ROS Message and Python Code. 43

A.1 Test 1 - Defining limits in the robot velocity: ROS Message and Python Code. 52
A.2 Test 2 - Defining the variable units on Twist Message : ROS Message and Python Code. 54
A.3 Test 5 - Variable assigned with a unexpected value : ROS Message and Python Code. 58
A.4 Test 7 - Variable assigned with a negative value for seconds : ROS Message and Python Code. 60
A.5 Test 8 - Variable was not initialized : ROS Message and Python Code . . 62
A.6 Test 9 - Unexpected range value assigned to one variable: ROS Message and Python Code. 63
A.7 Test 16 - Calculations lead to a unexpected value : ROS Message and Python Code. 71
A.8 Test 19 - Calculations lead to a unexpected value : ROS Message and Python Code. 75
A.9 Test 20 - ROS Message variable used but not initiliazed : ROS Message and Python Code. 76

List of Tables

5.1 Summary of the tests with their related bugs 41

Chapter 1

Introduction

This thesis proposes a method for statically detecting problems in robotics systems.

This chapter presents the motivation for this work on Section 1.1, as well as the objectives on Section 1.2, contributions on Section 1.3 and introduces the rest of the document on Section 1.4.

1.1 Motivation

Robotic Systems are widely used and have many real-world applications, such as cleaning with robitic vacuum cleaners, assembling cars in Toyota factories [49], or the most recent Mars Rover [53]. However, programming robotic systems requires domain-knowledge as well as a vast experience not only in the programming language or in the robotics framework. This is difficult because of the combined complexities associated with each software component, their integration with real-world sensors and actuators, the design of an architecture that combines these components [34], and the need for domain knowledge of the intended application or use [48]. Small mistakes in writing code can have a sizeable economic impact [28].

Developers can choose one of many frameworks to program robots, such as the Robot Operating System (ROS) [13, 48], CARMEN [1], Player [5], Open Robot Control Software [4], Orca [3], and MOOS [2]. These frameworks allow the user to program robots at a high-level. This work, focuses on ROS, sometimes referred to as the "Linux of Robotics" [18], as it is the most popular framework and has an extensive number of reusable packages, components, and simulators.

The Robot Operating System is a flexible framework for writing robot software [13, 48]. It is used in small, hobbyist projects as well as large-scale projects. For example, some universities work on autonomous cars with ROS to compete on the Indy Autonomous Challenge [37]. NASA is also using ROS to do Planetary Rover Simulation [26] for Lunar Exploration Missions. These are only two examples where ROS is used to program complex robots with complex tasks.

ROS employs a loosely coupled architecture that connects different components to create a robot. If one ROS component changes, it may affect the behavior of the other ROS components. The loosely coupled architecture improves the experience of developers as they are no longer required to understand the entirety of the robotic system and allows developers to reuse the software components. The ROS ecosystem includes a collection of tools, libraries, and conventions to simplify creating complex and robust robot behaviors across a variety of hardware. These characteristics allowed ROS to become more popular than other frameworks.

Given the system architecture, developers can reuse components that are already implemented. For example, ROS contains a natively implemented module to move robots, different robots can use it. This programmers not start to building the robotic system from scratch, not requiring ROS developers to be mechanical or electrical engineering experts. ROS already provides several modules that abstract this knowledge. For instance, the developer may import a specific velocity module to induce movement in the robot. However, programmers tend to rely on the same packages as stated by Kolak et al..

Nevertheless, developing software is not easy, and programmers frequently make mistakes. As stated by Canelas et al., we can identify different categories of challenges when developing in ROS. One example is missing unit conversion, which occurs when a programmer creates a component using the wrong magnitude units. One practical example is when programmers use a method that receives velocity in m/s, and they believe the method requires velocity in km/h. Another common mistake is that the dynamic architecture of ROS allows the incorrect connection between different components, which may occur due to typographical errors in strings or selecting the wrong identifier. The connection to other ROS components can be inaccurate, and the compiler does not help with this type of mistake. These problems are harder for new developers [30], making ROS challenging to learn. Another case is that a program can work well and compile; however, does not ensure the correct behavior of the robot since there may be domain-knowledge restrictions to the system. Some problems may happen only in a small percentage of executions, allowing bugs to be released to production.

In ROS, there are no boundaries to what the developer can write in the messages to communicate with the robot component. For example, a developer can choose the velocity at which the robot should move, but the compiler does not check if the value makes sense according to real world restrictions such as the maximum velocity that one robot can move. Thus, programming in ROS can lead to run time problems because the robot runs with commands it cannot perform.

Bugs in robotic systems can cost significant amounts of money. One such example is when a whole factory halts because one robot stops working properly, and it can cause monetary losses in inventory while it is not fixed. Another example, is the Mars Climate Orbiter crash in 1999 [36] a minor unit conversion bug. Bugs in robotic systems are

worrying, and one should invest in techniques to avoid them.

Currently, developers have multiple ways to test their program [24] to check the correctness of their system. But testing is time consuming and expensive. Tools such as Phys [39] or Phriky-Units [46] have been proposed to help programmers finding bugs and testing their programs. While these tools have found some success in discovering bugs, they are prone to reporting both false positives and false negatives.

1.2 Objectives

This work created the RobotFix tool to address the challenges of statically detecting bugs in ROS programs, with the goal of improving robotic systems correctness. I aim to help developers by identifying errors in their programs, and providing assurance on the values that the developers assign to the variables.

In particular, the bugs that the work targets are: assignments between variables with different units, variables set with unexpected values, and the use of uninitialised variables.

To this end, I developed a Domain Specific Language (DSL) that supports a static verification tool that can detect a subset of bugs in ROS programs. Bugs like the mismatch in velocity units can be detected statically [46], before running the program. By having an annotation of the units each variable, a static analysis tool can keep track of these units and emit a warning or an error when there is not enough information to confirm the unit usage is correct.

1.3 Contributions

This work aims to contribute for the state of art with:

- A study on the programming problems in ROS by inexperienced programmers on *An Experience Report on Challenges in Learning the Robot Operating System* [30].
- A Domain-Specific Language for writing ROS with semantic annotations.
- A typechecker for Python programs that detects domain-specific bugs in ROS projects.
- An evaluation of the potential of this language to detect common bugs in ROS projects.

1.4 Structure of the document

The remainder of the document is organized into six chapters. Firstly, I introduce the Background and Related Work (Chapter 2) that presents Robot Operating System and identifies studies on ROS, tools created for ROS programmers and Liquid Types in other

languages. In the next chapter, I introduce the Approach (Chapter 3) that presents the proposal and how I want to tackle the problem. In Implementation (Chapter 4), I explained how I implemented in detail each part of RobotFix. Afterwards, I explained how I evaluated the approach in the Evaluation (Chapter 5). To conclude, the Future Work (Chapter 6) explains the future goals for RobotFix and in the Conclusion (Chapter 7), I present the takeaways of this dissertation.

Chapter 2

Background & Related Work

2.1 Robot Operating System

This section describes the underlying concepts of the Robot Operating System, and the challenges and errors that can arise when developing ROS programs.

Robot Operating System [48] is a component-based framework built to manage software complexity and to reduce the effort in building robot prototypes. ROS was designed for different hosts connected at run-time with a peer-to-peer topology. ROS also supports different languages and uses an Interface Description Language (IDL) to support cross-language development. ROS contains small tools used to run and build ROS components. These small tools perform various tasks like navigating the source code tree and getting and setting configuration parameters. ROS developers can create small executables using library functionalities hide the complexity inside each library, making coding in ROS easier. ROS is also free and has the source code public intending to make debugging at all levels of the software stack easier. ROS developers can use the rich ROS ecosystem of reusable software components and do not need to build everything from scratch.

2.1.1 ROS Supported Languages

This work presents the examples in Python, although ROS programs can be written in Python, C++, or a mix of both. ROS is agnostic to the language and is independent of any specific programming language. Additionally, an Interface Definition Language describes the interfaces of components and the communication between components.

In ROS, programmers can create a ".launch" file to run the program, which uses a specific XML format used to run ROS projects and ROS components. ROS developers can run projects using command `roslaunch <package_name> <file_name.launch>` , which uses ".launch" files or the command `rosrun <package_name> <executable_name>`, which runs the specific Python or C++ files. The difference between both is that **rosrun** runs one node at a time and **roslaunch** two or more process at the same time (the nodes can be from different packages).

ROS also uses a Unified Robot Description Format (URDF) file in an XML format that contains the specifications for robot models, sensors, and scenes. ROS developers read this file to understand the geometry of the robot and its cells because it is code-dependent and more readable.

2.1.2 Nodes and Topics

A ROS node is a process that performs computation. The robot control system comprises nodes, and each node accounts for one part of the behavior of the robot. For instance, in robotic systems, movement and localization are usually necessary. The programs should have one node to take care of the wheels and another to take care of the localization. For instance, **kobuki/laser/scan** is one node used in many ROS programs and is responsible for reading the planar laser and identifying if something is close, is a typical example of a node in ROS projects.

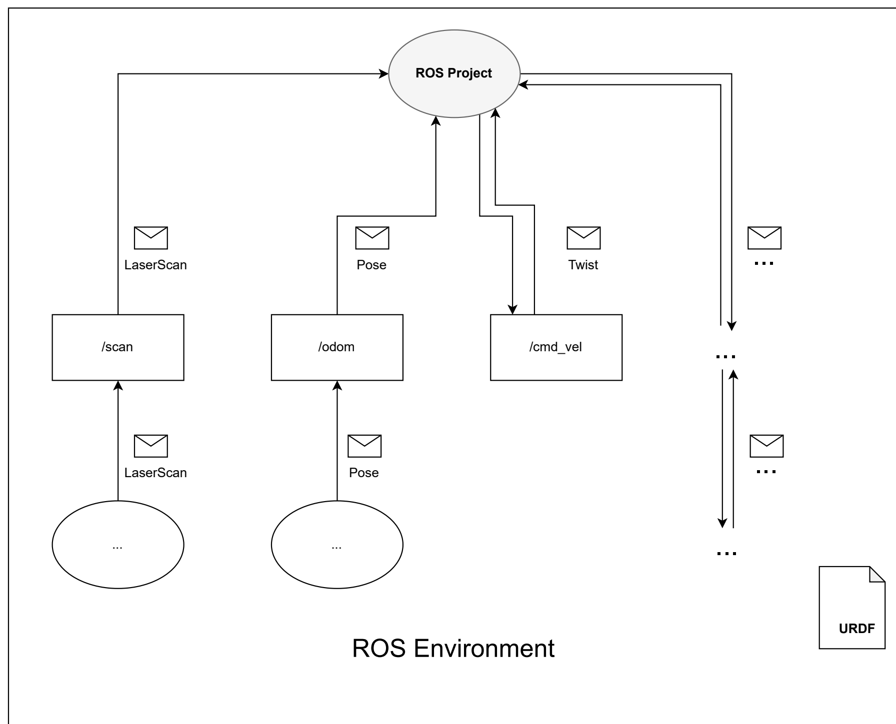
Nodes serve as an encapsulation and abstraction layer, similar to classes in Object-Oriented Programming (OOP). They also provide fault tolerance by isolating the crashes in individual nodes and reducing the code complexity.

ROS topics [16] are used as buses over which nodes exchange messages. Nodes communicate through topics to which they publish and subscribe for information. These do not have information about who they communicate with because topics have anonymous publish/subscribe semantics that decouples the received and set data.

Nodes that are interested in the data on a relevant topic can subscribe and receive information. Nodes can send data to the topics by publishing. Topics can have multiple publishers and subscribers simultaneously.

Some examples of requestly used topics are **/cmd_vel**, which provides the values of the motor for navigation, **/scan** receives and read the robot planar scan values, or **/odom** used to read the position and orientation of the robot.

Figure 2.1 presents a ROS project that interacts with the three topics previously presented. This ROS project is a node that can receive data from the three topics. The objective of the example project is to obtain the position and orientation of the robot, and the values of the scanner to determine the velocity of the robot. To obtain the values of the scanner, it receives information from the **/scan** topic that has the values submitted by the LaserScan node. The project gets the position and orientation of the robot by subscribing to the **/odom** topic. Finally, the project can subscribe to the **/cmd_vel** topic to get the velocity of the wheels and publish to the topic to choose the values of the velocity the robot should move.



Legend:

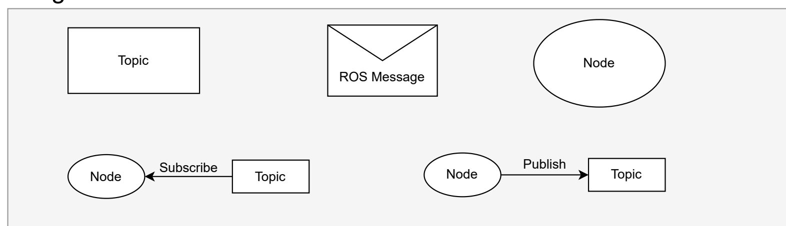


Figure 2.1: Nodes, Messages and Topics in ROS environment.

```
geometry_msgs/Vector3.msg
  float64 x
  float64 y
  float64 z
```

Listing 2.1: Example of a message: Vector3.

```
geometry_msgs/Twist.msg
  Vector3 linear
  Vector3 angular
```

Listing 2.2: Example of a message: Twist.

2.1.3 ROS Messages

ROS contains three types of messages implemented in the Interface Definition Language. These messages have differences between them and are used in different cases. Some of these messages that developers can use are in built-in ROS packages.

Listing 2.1 introduces a message with fields with basic types. Other ROS messages like the one represented in Listing 2.2 also include fields of different message types. These message types describe the contents that are expected, but not enforced, to go through topics. In the example, the Twist message type describes what data is published to the **cmd_vel** topic: a vector with the linear velocity of the robot and another with the angular velocity.

Listing 2.3 introduces one example of ROS messages used in Python code. For example, Developers can use the Twist Message to publish the direction and the velocity the robot should move, and the robot is expected to have that behavior.

Messages are also used for Services and Actions. Services provide ROS components with a synchronous RPC-style communication (unlike the asynchronous nature of raw topics). Service message types discriminate between the fields in the request (`data` in Listing 2.4) and in the response (`success` and `message`).

```
move_pub = rospy.Publisher("/cmd_vel", Twist)

# Create the message
message = Twist()
message.linear = Vector3(1.0, 0, 0)
message.angular = Vector3(0, 0, 0)

# Publish the velocity
move_pub.publish(message)
```

Listing 2.3: Example of a Twist Message being published.

```
std_srvs/SetBool.srv
  bool data
  ---
  bool success
  string message
```

Listing 2.4: Example of a service: SetBool.

```
drone/DroneImage.action
  int32 duration
  ---
  geometry_msgs/Point[] PointPose
  ---
  float64 distance
```

Listing 2.5: Example of an action: DroneImage with Goal, Feedback and Result, respectively.

Actions provide a callback-style asynchronous two-way communication pattern, and can also be described using message types. In Listing 2.5, the three parts of the action are separated by horizontal lines: Goal, Feedback and Result. In this case, the Goal includes the duration that the robot should move, the Feedback an array of positions in which the Robot has been, and the Result is the distance travelled by the robot.

2.1.4 ROS Packages

ROS developers create their packages to make their programs. These packages have the *src* folder with the Python programs, but this is not the only *src* folder because in ROS, programmers usually other packages that have their *src* folders.

A robotic system in ROS is composed of directories and files. Some directories and files are essential for the work. Specially in the *src* folder, stores Python programs that developers create, the *msg* folder, developers write their ROS Message.

The relevant files that ROS system includes are:

- **CMakeLists.txt [7] :** is the input to CMake build system for building software packages;
- **package.xml [8] :** defines properties about the package, such as the package name, version numbers, dependencies, etc;
- **file_name.launch [10] :** defines a specific XML format used to run ROS projects and ROS components;
- **file_name.msg :** defines simplified messages with a description language to represent data that ROS nodes publish;

- **file_name.srv :** defines simplified messages with a description language to define ROS service types;
- **file_name.action :** defines simplified messages with a description language in which the Server and the Client communicate.

The relevant directories that ROS system includes are:

- **launch/ :** includes .launch files;
- **src/ :** includes .py source files;
- **scripts/ :** includes .py scripts that can be used by users;
- **msg/ :** includes .msg files;
- **srv/ :** includes .srv files;
- **action/ :** includes .action files.

2.2 Related Work

This section describes other works on bugs, modelling and verification of Robotics Systems. This section also provides information about other verification tools outside of the robotic world that are relevant for this work.

2.2.1 Robot Programming Studies

This subsection describes studies on robot programming, especially bug analysis in robot programs and gaps that make programming robots more difficult.

It Takes a Village to Build a Robot [41] is a study by Kolak et al. on ROS ecosystem examining all ROS packages on GitHub along with their dependencies. This study found that ROS keeps attracting new applications, developers and end-users and identified that a small group of foundational working groups with 25 packages were exclusively used by 82% of ROS applications. The ROS ecosystem stagnated since these packages are growing at a slower rate than the other packages. RobotFix can use these packages to create conditions on their messages that will be used by multiple developers.

A Survey on End-User Robot Programming [25] tries to understand how end-users program robots, using rule-based programming, natural language programming, and visual programming as examples. The authors review the end-user robot programming space and its needs for improvement. The authors study people without experience in programming, believing that the user makes multiple errors and tends to misjudge their

programming abilities. The author states that the end-user robot programmers need assistance identifying errors and help developing programs more correctly. This study shows that static analysis tools like RobotFix will be necessary for robot programmers.

The Forgotten Case of the Dependency Bugs On the Example of the Robot Operating System [27] is a study by Anders Fischer-Nielsen about dependency bugs with Robot Operating System projects. The evaluation was made on 406 ROS packages, 1354 ROS bug reports, and 19553 bug reports and stated that dependency bugs appear in 15% of all issue reports (53% of projects). The authors also stated that dependency linters could be built quickly for specific cases but not all cases, making it hard to find all dependency bugs without extensive analysis. This work shows most ROS projects have errors that static analysis can find but still have flaws. With this information, RobotFix lets developers write annotations to make the analysis more efficient.

A Study on Challenges of Testing Robotic Systems [24] identified 12 practices and nine challenges in robotic testing. This study was based on an inquiry about the programmer experience with testing robot programs. The test practises were field tests, simulations or simple tests on parts of the program. However, it is hard to predict all cases that can happen in robot behaviours, and it is complex to build and maintain end-to-end harnesses for robotic systems. The cost and resources necessary for robot testing are also an issue. Finally is hard to make simulations of the real world. It has a lot of differences, mainly because the real world has multiple complexities in the environment. This study shows that testing with robot programs is difficult. RobotFix will be a static analysis tool to test and find errors without real robots being tested or in simulations.

An Empirical Study on Type Annotations: Accuracy, Speed, and Suggestion Effectiveness [47] is a study on type annotations in robot programs. The study addressed studies with programmers. The study verified that 51% of the annotations made by the participants were correct, and the time to make the proper annotation was around 2 minutes. However, type annotations have room for improvement. They also studied other plugins that add annotations to variable definitions, Phys [40] and Phriky-Units [46] and claimed that these plugins should make three suggestions instead of one because multiple suggestions were wrong. Nevertheless, combining both approaches seems the best approach because it will reduce the time that type annotations take for developers. RobotFix will use annotations, which this study express that are more efficient than tools that do not.

Sharma et al. studied how code changes can impact the rate at which data is produced or consumed on **Rate Impact Analysis in Robotic Systems** [51]. These changes can impact the robot behaviour and performance and are complex to identify the problems of the changes. They implemented a tool that creates a graph with edges from a publisher/subscribers to a topic and labels them as "rate-dependent or rate-independent and then analyses the code and tries to find some patterns in code for labelling the edges.

They count it as "rate-dependent" if it does not find any pattern. The tool was analysed in three case studies and reduced by 41% the impact set that developers should process. However, the application had some limitations. For example, it could not handle or reach some files and these files were not tested. These limitations led to limited good results (low false positives and low false negatives) that cannot be generalised. This approach helps developers find errors that can affect the robot performance by changes in the code, and the application got some inspiring results. Static analysis is efficient in finding errors after changes in the code. RobotFix will be helpful for cases with changes in the code, specifically when multiple developers use the same variable with different objectives.

Synthesis for Robots: Guarantees and Feedback for Robot Behavior [42] is a study on formal synthesis for robot control. Synthesis provides more advantages when doing complex tasks with different constraints. Some jobs like going to a goal in a cluttered workspace are better served using motion planners or learned controllers. However, complex robotic systems like humanoid robots are challenging to model and abstract. Robot programs are complex, and it is challenging to work with them. RobotFix aims to simplify some issues developers will have with the complexities of robot programs.

These studies identified essential things for RobotFix. Developers introduce errors in their programs that linters can not recognise. Although ROS developers usually use the same packages. These packages can receive improvements in their usability that will improve the programming experience of programmers. For robot programmers is important to have help finding bugs since they make multiple errors and tend to misjudge their programming abilities. Also static analysis on robots are important in robot programming due to the disadvantages of simulations. However, it is not easy to create plugins that can be consistent in finding the mistakes, one of these examples in on tools that make type annotations to find errors on unit variables.

From these studies, we can conclude that static analysis are important in the robotic world and exist space for new tool that provide more assurance for the users.

2.2.2 Modeling and Verification Tools for Robotics Systems

The section describes other tools functionalities and the gaps that RobotFix could fill to improve the programming of ROS developers.

Non ROS-Specific Approach

ROBOSYNTH [44] is a new approach to solve integrated task and motion planning, a challenging class of planning problems that involve complex combinations of high-level task planning and low-level motion planning. ROBOSYNTH requires three inputs. The first is the scene description that specifies the robot and the physical space. Secondly, a plan outline is given describing information about credible plans. The last one is the

requirements that the plan generated by ROBOSYNTH must follow. The program produces a placement graph with all the possible locations of the robot and robot feasible actions. Then ROBOSYNTH computes a logical formula representing all plans that the robot should follow that satisfies the requirements. Finally, the program uses a SMT Solver to compute a satisfying solution between all plans. This approach requires a developer help to work on the plan outline but shows us that SMT Solvers can be used to assist developers to find plans for robot behaviours. ROBOSYNTH have a different goal than RobotFix since that it aims to solve planning problems in robotics.

ROS-specific Approaches

The High-Assurance ROS Framework (HAROS) [50] is a framework that focuses on the quality assurance of ROS-based code and static analysis. The HAROS analyser takes an input that is a user-provided YAML project file with specific configurations and with the ROS packages and Plug-ins to analyse the code and do an output. The output has an Issue Report, Runtime Models, Runtime Monitor and Json Data Files. This framework requires a specific input file and results on an external page. The evaluation had good results and found safety bugs in one of the industrial case studies but had some limitations regarding plug-ins based on behavioural properties. These limitations exist because HAROS gives a good frontend for other static analysis tools not made for ROS (that have complex specific code). RobotFix is complementary to HAROS because these static analysis tools do not address the same functionalities, and RobotFix addresses ROS specific code.

Applying Software Static Analysis to ROS: The Case Study of the FASTEN European Project [45] describes the H2020 Flexible and Autonomous Manufacturing Systems case study for Custom-Designed Products (FASTEN) project. The authors Neto et al. used HAROS to find issues on the project. This study analysed the code in four iterations, solving the issues between iterations. In the first iteration, they had 28043 issues, and in the last iteration, they had 2603 issues. Between iterations, the average issue severity and the average effort to solve the issues tend to increase. In the last iterations, they had more difficulties solving the issues and solved the same percentage of previous iterations. This work shows that Static Analysis can help developers detect issues on real projects, but they need help fixing them.

PHYSFRAME [40] is a tool that detects frame inconsistencies and convention violations using automated type inference and checking techniques. PhysFrame reports seven types of errors, two types of warning and three types of violations of implicit conventions. These errors, warnings and violations came from the rules defined by the authors. This approach studied common practices and checked violations of those practices. The authors evaluated this tool on 180 ROS projects band got 81% of true positives and reported that to the developers, they got some responses saying that they fixed and was a real bug

but also got few responses saying that it was not a problematic issue. RobotFix can find the same errors that PHYSFRAME with annotations, developers can make annotations with the frame type, and RobotFix can find inconsistencies between frame assignments if the frames are correctly assigned.

Statik [54] is a plugin-based tool for the Robot Operating System that allows programmers to run multiple plugins that do static analyses and linting tools just in one plugin. The developers configure the "config.yaml" with the analysis level they want on their projects, allowing them to ignore types of detail errors they do not need. Developers can configure the "profile_objetive.yaml" to specify the analysis level for each package or "excepetions.yaml" to remove false positives. The output is printed on the console or an XML file which can be parsed by another tool called Jenkins. Statik helps developers find issues in their code by combining multiple static analysis tools with their strengths and weaknesses. This tool is different from RobotFix because it does not provide help while programming. Statik is a complementary work with RobotFix since it uses other linting tools in this plug-in that can not find the same errors that RobotFix finds.

Phriky-Units [46] is a tool that detects physical unit inconsistencies in robot systems by static analysis. Phriky-Units works as follows: first, they take a program and map the attributes in shared libraries to units. After, they use the mapping to decorate code with physical units and then propagate units in expressions. Finally, they find inconsistencies by using the dimensional analysis. The was done in 213 open-source robotic projects, and they got 87% precision finding on errors found. However, just 11% of the projects had this type of errors.

Phys [39] is a tool to find bugs on inconsistencies between the units of variables on C++ programs. Phys infers the variable units with a probabilistic approach based on the evidence in the code, like variable names and expressions, making them initial probabilities to model uncertainty. The authors compared Phys with Phryki-Units and found 103% more unit inconsistencies than Phriki-Units. However, they had 15% of false positives and did not have information on the false negatives.

Phys and **Phriky-Units** find errors that RobotFix can also find, the difference is that RobotFix uses annotations that the study made by Ore et al. says that should be more efficient.

SOTER [52] is a framework that grants the developers a run-time assurance framework for building safe distributed mobile robotic systems on Robot Operating System (ROS). In this System, the developer can program a Task Planner that is implemented as a state machine and makes some, programmed by the developers, events happen in the execution of the robot and sends it to a motion planner that computes a plan to do the task and gives them to the plan executer. The plan executer consults the decision module, and when it can, executes the plan, notifies the Motion Planner and wait for the next plan. This framework does not provide help for programming like RobotFix but helps the user

to do programs for their goals.

Roswire [23] is a Python library for static and dynamic analysis of Robot Operating System applications. Roswire can provide the users with all information inside ROS applications with the respective docker image. Roswire can also let the users use an interface that allows them to generate and interact with instances of that application in the form of docker containers. For example, the users can do service calls. RobotFix will use Roswire code because it has useful functions for programmers that work with ROS.

ROS does not have a tool similar to RobotFix, where developers can make annotations, and it will find inconsistencies between the code and annotations. Phys and Phikry-Units do annotations automatically, but that approaches brings false positives and an unknown number of false negatives. RobotFix approach should be more precise because it does not need assumptions to define the unit.

2.2.3 Type Annotations

Jhala and Vazou made a study about **Refinement Types** [38]. They refer that refinement types allow developers to enrich their type system with predicates that restrict the values of their variables. In RobotFix, we want to use Refinement types, allowing developers to limit variable values for robotics. In this study, the authors also proclaim that developers can inform the type system about invariants and correctness properties they care about using Refinements. This study shows that refinement types are an excellent option for our tool, allow developers to restrict the values of variables and create properties for the variables.

Pytype [19] is a static analyzer made by Google that looks for Python code errors involving variable types. Pytype can flag mistakes like misspelt attribute names, incorrect function calls and others. With Pytype, programmers can use type annotations, and Pytype will check them. However, the use of annotated types is optional. Pytype traces through the bytecode of the program, simulating the effects of the Python compiler interpreter but following the types and ignoring the values. Google uses Pytype on thousands of projects. Pytype helps these projects to be well-types and error-free. Since annotations are already used in python tools, I want to improve that for more usabilities.

2.2.4 Type annotations tools with LiquidTypes

Liquid Types allow developers to use logical predicates with specifications about variables in order to verify the semantic properties of the code. With liquid types, developers can create rich conditions like " x > 0" in this example, the values of x should be more significant than 0, and this property will be verified in the code.

LiquidJava [32, 33] is a tool made by Gamboa et al. to help programmers to find bugs on Java. LiquidJava allows programmers to add refinements to, for example, limit the

variable values. When the developer introduces values that do not respect the refinements, LiquidJava detects and warns the user. LiquidJava results show that its users can find more errors in less time using LiquidJava compared to programmers without refinements.

LiquidHaskell [55] is a work by Vazou et al. that is a refinement type checker that can specify and verify Haskell's various properties, such as variable conditions (that I want to introduce in RobotFix) or others like proving the termination of recursive calls. The authors said also said that the boundaries of LiquidHaskell are constantly expanding. The fact that they already use Liquid Types to create conditions on the variables with success shows that Liquid Types are a good path for verifying the errors on unexpected values in variables.

Flux: LiquidRust [43] uses Rust ownership mechanisms to make type-based verification for low-level safe Rust code. Flux turns the values in the refinements in mutable locations and uses the ownership mechanism to track strong updates. Flux checks if the rust code does what it should with refinements. Liquid typing specifications are more efficient than the others in rust. It reduced lines to half compared to other specification tools in Rust like Prusti [29].

Liquid Types refinements are already used efficiently in other languages like Java, Haskell and Rust. The results in other languages made using it in ROS programs promising for RobotFix approach.

Chapter 3

Approach

This chapter explains how the approach allows programmers to add annotations to variables and ROS messages and finds inconsistencies between code and annotations. First, Section 3.1 introduces how I want to use LiquidTypes in RobotFix, afterwards Section 3.2 describes how the system will work for the users with examples. Then, Section 3.3 explains the requirements that the system should respect. Finally, Section 3.4 briefly introduces the architecture of the system.

3.1 Liquid Types in Python

In the approach, I want to use Liquid Types to add restrictions to variables. With Liquid Types, I can allow developers to introduce new restrictions on Python variables that are not possible now. Developers will be able to add restrictions to variables like "x > 0" , where the value x should be bigger than 0. These restrictions are robust, finding unexpected values in the program of developers.

I also implemented a typechecker that will find inconsistencies between the code and the annotations.

3.2 Base Example

This sections introduces how RobotFix should work for developers, with one example of Robot Operating System code.

The code example on Listing 3.1 describes a program that sends a message to the robot with the linear velocity vector of the robot with the values (200, 0, 0) each time the robot receives a ROS message called `LaserScan` from the topic `/kobuki/laser/scan`. This topic is responsible for saving the current values of the robot scans.

The code on Listing 3.1 was tested on a simulation with a Turtlebot [22], and the result of the simulation was that the robot went forward, then swung sideways, and ended up falling. Line 3 is responsible for the misbehaviour of the system. Some programmers that

```
def callback(msg : LaserScan):
    move = Twist()
    move.linear.x = 200
    pub.publish(move)


rospy.init_node("pub_sub_node")
sub = rospy.Subscriber("/kobuki/laser/scan", LaserScan, callback)
pub = rospy.Publisher("/cmd_vel", Twist, queue_size=1)

rospy.spin()
```

Listing 3.1: Example of Robot Operating System Code.

```
def callback(msg : LaserScan):
    move = Twist()
    x : Annotated[float,"Unit('m/s') and x < 20 and x > -20"]
    x = 200
    move.linear.x = x
    pub.publish(move)


rospy.init_node("pub_sub_node")
sub = rospy.Subscriber("/kobuki/laser/scan", LaserScan , callback)
pub = rospy.Publisher("/cmd_vel", Twist, queue_size=1)

rospy.spin()
```

Listing 3.2: Example of Robot Operating System Code Annotated.

use Twist messages do not know the unit of the values [20] or the threshold values that the robot can reproduce [21]. In this case, the robot received the information to go forward at 200 m/s, and could not handle it.

With RobotFix, developers can introduce annotations on messages or in their Python code. These annotations can help future programmers that will use that messages or code, or even the developer that wrote that annotations and by mistake introduced code that does not respect it.

On Listing 3.2 the programmer used RobotFix. In line 3 , the developer defines the type *Annotated*, that is imported from *typing_extensions*. In *Annotated* types, developers can define the type on the first argument, and on the second, they can write one annotation that RobotFix will use. In this annotation, developers can define the Unit or write one condition for this variable value.

RobotFix will reproduce an error on line 4 with this code. The error identified by RobotFix was the inconsistency between the annotation "x < 20" and the assignment of x

```
geometry_msgs/Twist.msg
  Vector3 linear
  #RobotFix# Unit(linear.x) == "m/s"
  #RobotFix# Unit(linear.y) == "m/s"
  #RobotFix# Unit(linear.z) == "m/s"
  #RobotFix# linear.x < 20 and linear.x > −20
  Vector3 angular
  #RobotFix# Unit(angular.x) == "rad/s"
  #RobotFix# Unit(angular.y) == "rad/s"
  #RobotFix# Unit(angular.z) == "rad/s"
```

Listing 3.3: Example of a message Twist with annotations.

```
def callback(msg : LaserScan):
  move = Twist()
  x : Annotated[float," Unit('km/h') and x < 20 and x > −20" ]
  x = 10
  move.linear.x = x
  pub.publish(move)


rospy.init_node("pub_sub_node")
sub = rospy.Subscriber("/kobuki/laser/scan", LaserScan , callback)
pub = rospy.Publisher("/cmd_vel", Twist, queue_size=1)

rospy.spin()
```

Listing 3.4: Example of Robot Operating System Code Annotated with wrong units.

with the number 200.

Developers can write annotations on ROS Message. Listing 3.3 presents one example of one annotated message. These annotations assign the unit to the variables of Twist and restrict values of *linear.x* to be between -20 and 20. The annotated messsage allows the code on Listing 3.1 to reproduce an error with RobotFix analysis on line 4 since the assignment of the variable does not respect the annotation on the message “linear.x < 20”.

We have another code example on Listing 3.4 that uses the annotated message on Listing 3.3. RobotFix will find an error in the code because the variable “x” and “move.linear.x” have different units.

Finally, we have an example of code on Listing 3.5 that represents a program that uses the ROS Message annotated on Listing 3.3 and does not have errors identified by RobotFix.

```
from typing_extensions import Annotated
import rospy
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist

def callback(msg : LaserScan):
    move = Twist()
    x : Annotated[float," Unit('m/s') and x < 20 and x > -20" ]
    x = 10
    move.linear.x = x
    pub.publish(move)


rospy.init_node("pub_sub_node")
sub = rospy.Subscriber("/kobuki/laser/scan", LaserScan , callback)
pub = rospy.Publisher("/cmd_vel", Twist, queue_size=1)

rospy.spin()
```

Listing 3.5: Example of Robot Operating System Code Correct for RobotFix.

3.3 System Requirements

This section focuses on the requirements to make the system helpful for ROS developers. These requirements are inspired by LiquidJava [32, 33].

- **R.1 - The verification should work on top of the Python and ROS compiler:** The purpose of the system is to find errors statically in ROS programs, aiming that the developer does not have unexpected problems while running the code. The annotations that the developers should write for the system to work should not affect the Python or ROS. The annotations provided by the developer do not affect the Python and ROS compiler and should not affect other versions of Python and ROS released in the future;

- **R.2 - Annotations should be optional:** The system should work on programs without annotations and successfully validate the input;

- **R.3 - The Annotations syntax should be similar to Python:** The system aim to help programmers that use the Robot Operating System framework with Python. It should be easy for them to write annotations with a syntax close to Python instead of learning a new syntax to write the annotations;

- **R.4 - The verification should be fast and consistent:** The system aims to help developers find errors statically preventing time-consuming and expensive task, such

as simulations. The verification must check the annotations fast and consistently, not giving different results for the same code;

R.5 - Annotations should work in programs without ROS-specific code: The system aims to help Robot Operating System developers and have specific adaptations for the framework functions and messages, but the system should work in programs without ROS.

3.4 System Design

The system needs to get the information about the variable, this is possible by parsing the Python code. For parsing Python programs, the system uses the static analyzer Pytype [19]. In addition to parsing code, Pytype also let us find errors in inconsistent types of variables. It checks if the expected type is a sub-type of the one defined in the code, and if it is not, it produces an error message.

ROS also uses messages to retain information that works like Python classes. One message has arguments that developers can set and access. These arguments can be other messages. Therefore RobotFix need to process ROS messages beyond Python code. To be able to parse the message information, the system used functions already made by Roswire [23]. Pytype also parses the annotations in Python code.

In ROS, the information given by the ROS messages is also essential. ROS programs use these messages to create a type of message and use the variables inside these messages. The system also needs the annotations to set the variables units and the refinements to limit the values. For the annotations in messages, was implemented a parser for the annotations for the messages.

The system prepares the inputs for the SMT Solver with the information given by the parser. The refinements written by the developers have converted into implications that the SMT Solver verifies. The SMT Solver is responsible for deciding if all the implications it received (from annotations and python code) are valid and do not exist inconsistencies. For example, suppose one variable with the implication that has the unit **m/s** is assigned with another variable with the unit **rad/s**. In that case, the SMT Solver will decide that the program is not valid because of these unit inconsistencies.

The ROS verification process is summarized in Figure 3.1.

3.5 Refinable Targets

In this work the annotations can restrict the values and possible assignments of variables. Annotations are introduced in the following code elements:

- **Variable declarations:** These refinements ensure that any variable assignment

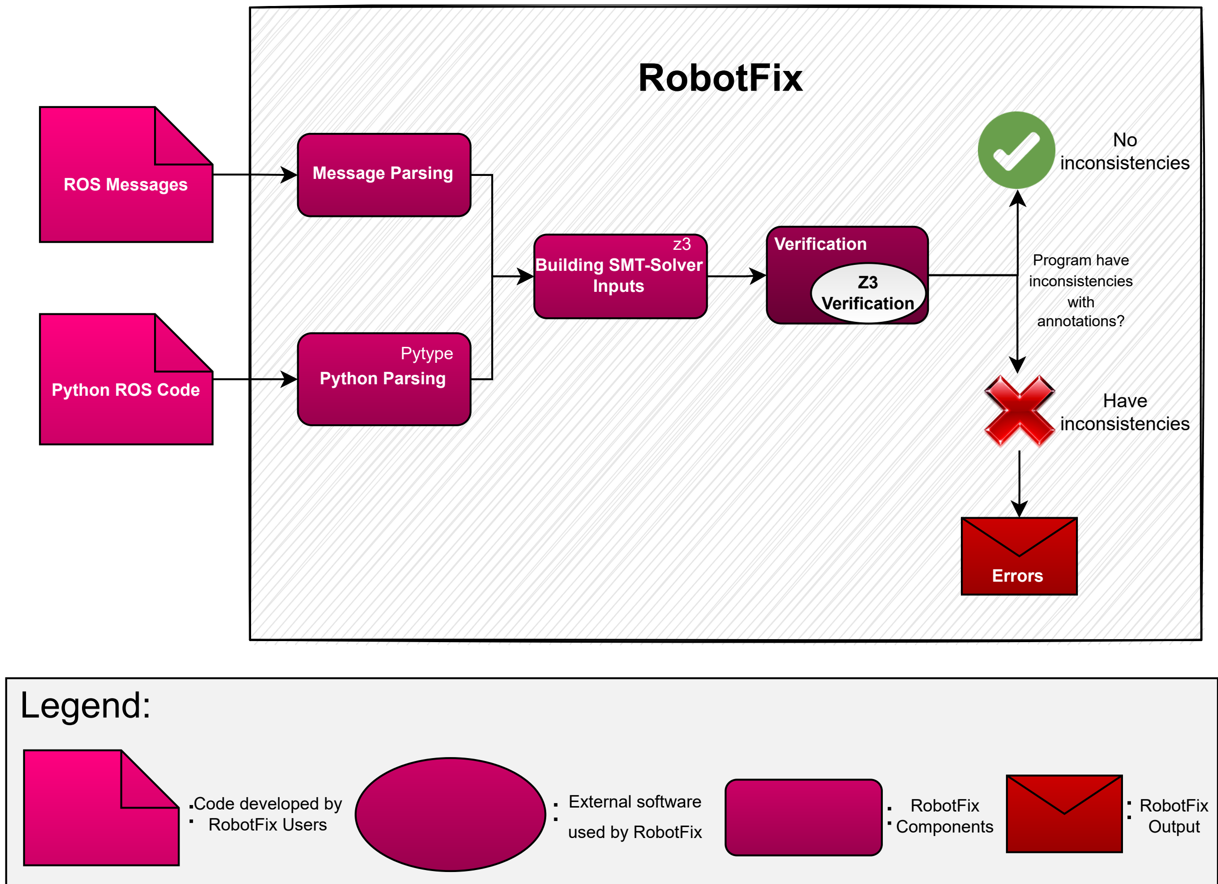


Figure 3.1: Architecture of RobotFix.

must respect the annotation of the variable. For example, given a variable A that is assigned to a variable B, they must have the same unit Z.

- **ROS Messages:** These refinements ensure that any variable of a message must respect the annotations on the message. For example, given a variable A of a message type M, if A is assigned as a variable B, variable B must comply with the restrictions of M.

Chapter 4

Implementation

This chapter explains in detail the implementation of the system briefly introduced in the previous chapter and now in a more complete diagram in Figure 4.1.

4.1 Parsing

The code written by the code is transformed into one sequence of tokens by the lexer, and this result is transformed into an Abstract Syntax Tree (AST) by the Parser. The system includes parsing Python code with annotations and ROS Messages with annotations. This section corresponds to the part of the diagram specified in Figure 4.2.

4.1.1 Grammar

The Robot Operating System framework uses messages to make the communication between the robot nodes possible. These messages contain the variables for the information that the ROS messages should store. One functionality that developers can use with the tool is making annotations inside these messages to restrict the values and the units that values of the variables of that specific ROS message should store as shown in the Listing 3.3.

Making those annotations on ROS messages should be beyond the code that is compiled by Python and the Robot Operating System framework. The system has a specific grammar that developers should follow when they want to make their annotations. This grammar aims to be as close as possible to Python so that developers do not need to learn a new language.

The grammar allows developers to make annotations with predicates that include logical operators (i.e.,"and or "or"), with the ternary expression (i.e,"if"-"then"-"else"), boolean operators (e.g., "<"), linear arithmetics (e.g.,"+"), and function of the system (e.g., Unit() that allows programmers to choose the unit of the variable).

The system reads these annotations if they respect two conditions. Firstly they should follow the grammar presented in Figure 4.3. Secondly, they should have the prefix

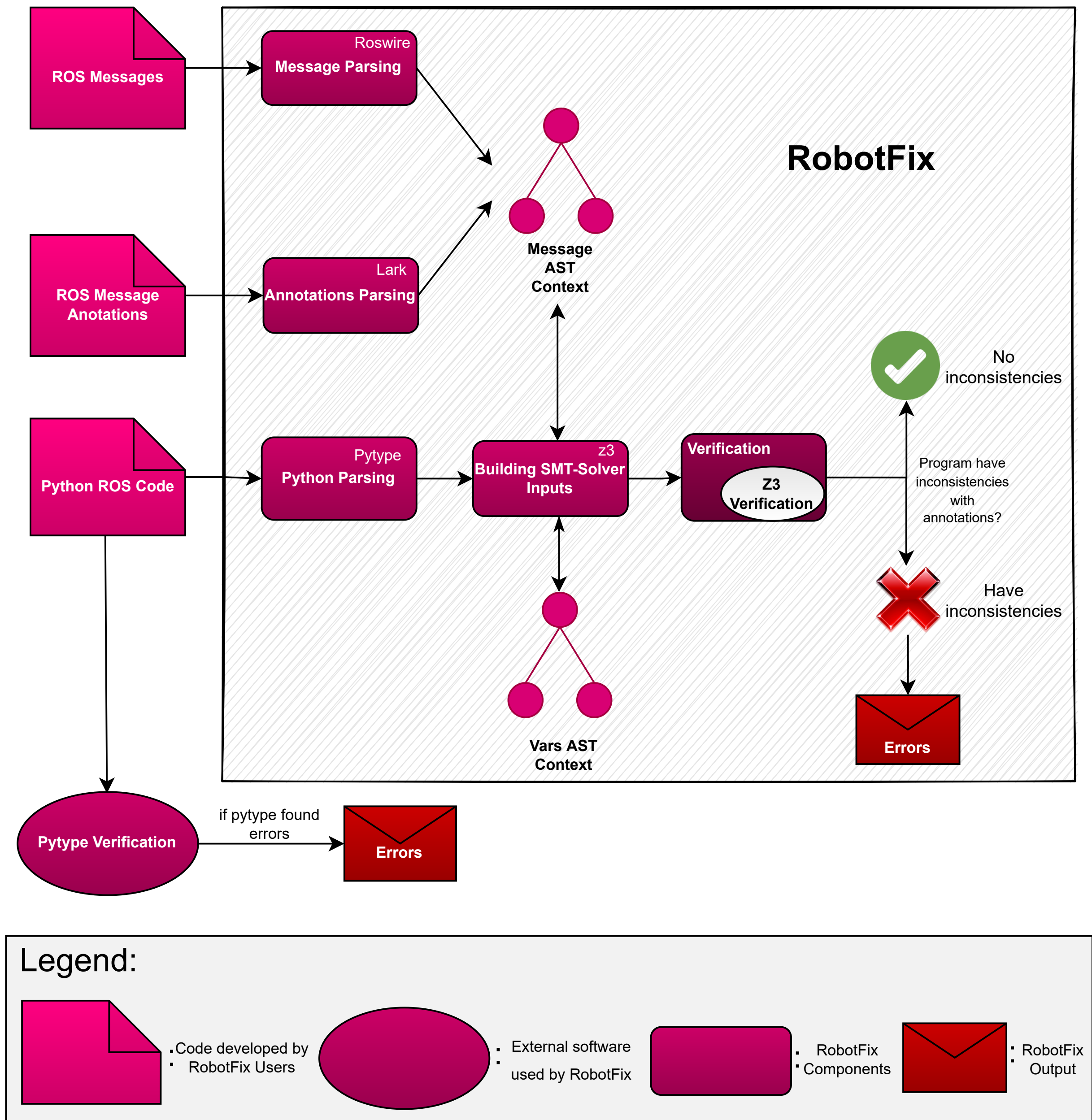
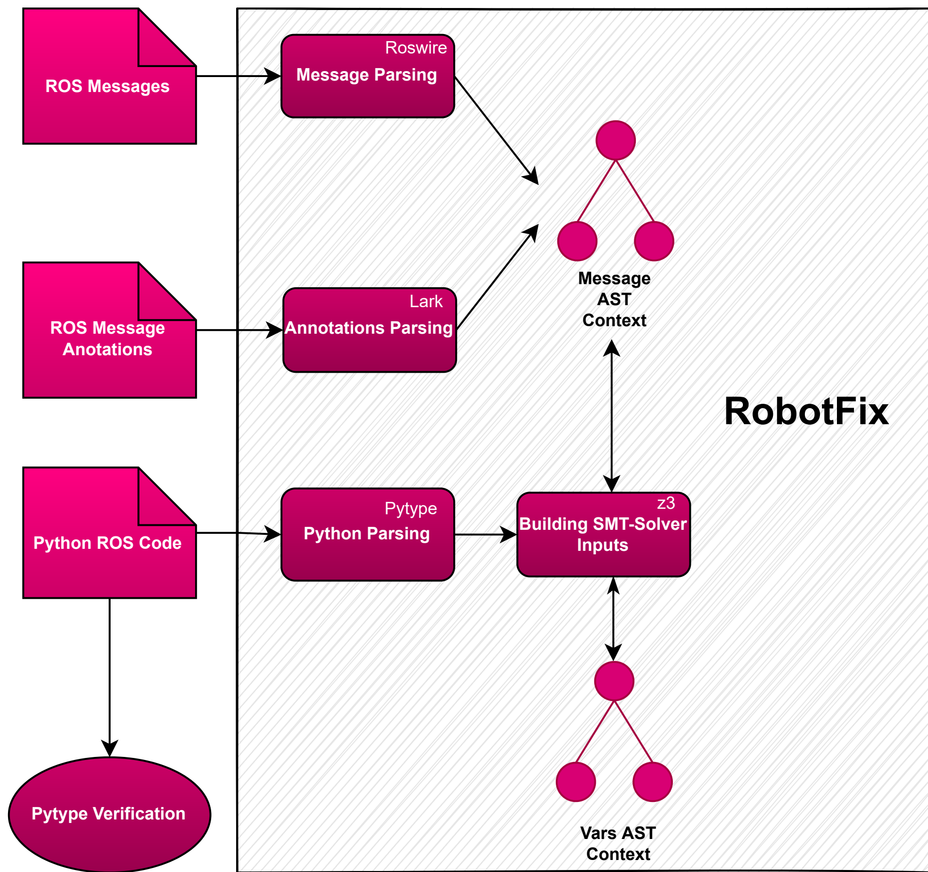


Figure 4.1: Complete architecture of RobotFix.



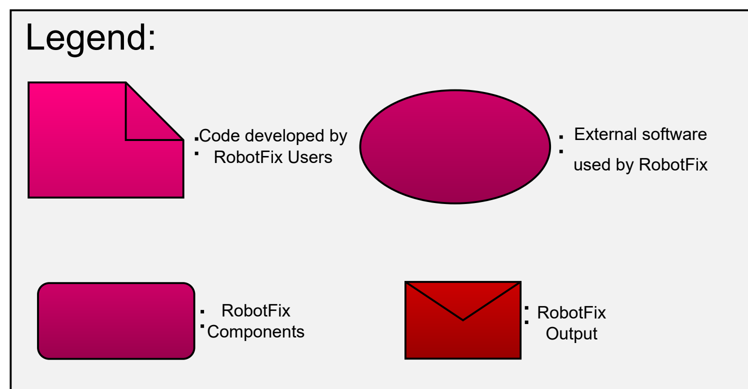


Figure 4.2: Architecture of RobotFix - Parse.

```
start        ::=  expr_log
expr_log     ::=  expr_if ( AND | OR ) expr_log
              |   expr_if
expr_if      ::=  "if" expr_fun "then" expr_log "else" expr_fun
              |   expr_fun
expr_fun     ::=  FUN"("expr_bool(","expr_bool)*")"BOOL_OPS expr_bool
              |   expr_bool
expr_bool    ::=  expr_arit1 BOOL_OPS expr_bool
              |   expr_arit1
expr_arit1   ::=  expr_arit2 (MULT | QUOT) expr_arit2
              |   expr_arit2
expr_arit2   ::=  expr_unop (PLUS | MINUS ) expr_arit2
              |   expr_unop
expr_unop    ::=  MINUS expr_final
              |   expr_final
expr_final   ::=  ( BOOL_LIT | INT_LIT | FLOAT_LIT | STRING_LIT)
              |   VAR ("." VAR)*
              |   "(" expr_log ")"
AND          ::=  "and"
OR           ::=  "or"
PLUS         ::=  "+"
MINUS        ::=  "-"
MULT         ::=  "*"
QOUT         ::=  "/"
LT           ::=  "<"
LTE          ::=  "<="
GT           ::=  ">"
GTE          ::=  ">="
EQUAL        ::=  "=="
DIFF         ::=  "!="
NOT          ::=  "!"
BOOl_OPS     ::=  LT | LTE | GT | GTE | EQUAL | DIFF
VAR          ::=  ("a".."z" /[a-zA-Z0-9_]/*)
              |   "_"
FUN          ::=  "A".."Z" /[a-zA-Z0-9_]/*
BOOL_LIT     ::=  "true" | "false"
INT_LIT      ::=  INT
FLOAT_LIT    ::=  FLOAT
STRING_LIT   ::=  ESCAPED_STRING
```

Figure 4.3: Grammar for the Annotations Language.

"#RobotFix#" before the annotations. The prefix is necessary for the compiler to distinguish general comments from the annotations that developers want to use on the tool.

4.1.2 Pytype

Pytype is essential because in this work, in addition to helping with parsing, the approach also uses Pytype to find inconsistencies between types of variables. In the approach, I changed the Pytype code to use the parser already implemented for ROS.

Pytype works like a "shadow bytecode interpreter" that uses the bytecode of a program and mimics the effects of the CPython interpreter but tracks types instead of values.

Inside Pytype, we have names to reference types that have an essential job for their program. The bytecode interpreter of Pytype is called **Virtual Machine** and reads through the program building a **Typegraph**. The **Typegraph** maps the flow of types during the program.

The **Typegraph** is composed of nodes that save the single statements in the program. We also have the type **Variable** that keeps type data about the variables when they are initialized. The **Variable** is composed by **Bindings** that save the value of the variable for one node (The **Variable** saves all the **Bindings** containing all the information about the variables on these). The association between **Bindings** and **Nodes** represents the types in the program and can be saved in the type **Abstract Values** or the type **Data**.

```
1  def callback(msg : LaserScan):
2    move = Twist()
3    x : Annotated[float,"Unit('m/s') and x < 20 and x > −20 "]
4    x = 10
5    move.linear.x = x
6
7
8  rospy.init_node("pub_sub_node")
9  sub = rospy.Subscriber("/kobuki/laser/scan", LaserScan, callback)
10 pub = rospy.Publisher("/cmd_vel", Twist, queue_size=1)
11
12 rospy.spin()
```

Listing 4.1: Example of Robot Operating System Code to be parse.

Pytype works by popping the **Variables** from the **Virtual Machine** data stack to do the verification and push back to the stack the result. While Pytype does the verification, it also uses the **Typegraph** to know the type of the variable at the current node.

4.1.3 Python Code Parse

Listing 4.1 presents a code example that will be parsed by Pytype.

The code example on Listing 4.1 should be parsed by Pytype. The parsing also reads the variables with the type annotated with *typing.Annotated*. Inside of the *typing.Annotated*, developers can introduce the type and one string with the desired unit or condition that the variable should respect. One example is line 3 of Listing 4.1 where the variable *x* has the type *int*, the unit "**m/s**" and should always be more than -20 and less than 20.

The lines 2, 4 and 5 on Listing 4.1 are also important. Pytype parses those lines, and the system uses the information about the values and variables created in that program.

The approach uses Pytype to parse ROS code, but Pytype was not ready to be used with ROS and it needed some changes in the Pytype code. First, Pytype needed ROS stubs to understand how ROS functions work because it did not have information about ROS complex functions and methods and ROS packages. Then, Pytype code had to have changes to read Subscriber callback functions that Python does not call but are consistently being used in ROS. Pytype did not parsed these functions because it was waiting for a function call that usually does not exist in ROS. In order to make Pytype read callback functions, we added opcodes relative to the call of these functions on the data stack.

Pytype also finds errors in the types of the variables. The system uses that functionality and also returns errors found by Pytype.

4.1.4 ROS Message Parsing

ROS messages are essential for developing ROS systems because these are used in the communication between nodes. One of the most used messages is the *Twist* message. The Twist is used to send the linear and angular velocity to the robot.

Vector3 is another message representing a vector of 3 values, *x*, *y* and *z*.

The message is created with the line 2 on Listing 4.1. When the developers create a Twist message inside Python code, the variables are initialized with an object of type Twist. In the example in the Listing 4.1, Python will create fields "move.linear" and "move.angular". The "linear" and "angular" are also messages, in this case, Vector3 Message, and they also will initialize their variables. The system will initialize "move.linear.x", "move.linear.y", and "move.linear.z" on the object type *Twist*.

The approach saves ROS messages that developers want to use in one folder. ROS developers usually use messages imported with ROS packages and do not need to create these ROS messages in their ROS folder. Still, if these developers want to create conditions on these messages, they can create the message in this folder and make their annotations on the ROS messages. ROS messages are parsed into an Abstract Syntax Tree that keeps the annotations and the fields of the ROS message (for example, for *Twist* the *linear* and the *angular* will be the two fields). These fields will also have a type like the initial ROS message. The parse of the fields is done with Roswire, a Python library for static and dynamic analysis of Robot Operating System applications. This library has functions already made for this purpose on other systems. The fields can correspond to other ROS messages. These messages will be parsed by Roswire even if they are not in the folder. Developers can annotate variables that are initialized in the type of their fields. This functionality allows developers to make all annotations in one file with the specifications and not have to make annotations in multiple files that will be used at the same time and purpose. Listing 4.2 shows an example.

The file on Listing 4.2 has annotations. However, some of them are not being used by the tool. RobotFix reads the annotations after the "#RobotFix#" prefix and not the annotations on lines 12 and 13.

In this case, Twist Message is annotated with conditions on the unit of variables and values. In this example, the developers set the unit of "message.linear.x" , "message.linear.y" and "message.linear.z" as "m/s" and the units of "message.angular.x" , "message.angular.y" and "message.angular.z" as "rad/s". The developers also set that the values of "message.linear.x" should be between "-20" and "20".

The approach parses the annotations in the message with Lark [17]. Lark is a tool made for developers to make parsers for their own grammars.The annotations are written in the grammar in Figure 4.3. The result of the parsing is sent to the message context.

```
geometry_msgs/Twist.msg
  Vector3 linear
  #RobotFix# Unit(linear.x) == "m/s"
  #RobotFix# Unit(linear.y) == "m/s"
  #RobotFix# Unit(linear.z) == "m/s"
  #RobotFix# linear.x < 20 and linear.x > −20
  Vector3 angular
  #RobotFix# Unit(angular.x) == "rad/s"
  #RobotFix# Unit(angular.y) == "rad/s"
  #RobotFix# Unit(angular.z) == "rad/s"

  # move.linear.y > 0
  #Annotation# move.linear.z > 0
```

Listing 4.2: Example of a message Twist with bad annotations.

4.2 Building Contexts

This section will explain how RobotFix holds the information from the parsing process in the contexts and how it creates the inputs for the SMT Solver. This section corresponds to the part of the diagram specified on the picture Figure 4.4

Before parsing the Python Files, RobotFix already has the ROS message information in a specific context for this type of information.

While Pytype runs, another context is built with the variables information. This section will be divided into three parts, one for each context and one explaining how the approach creates inputs for the SMT Solver.

4.2.1 Messages Context

The message context is used to help the system to keep information given by ROS Message and their annotations. One Python dictionary has all information obtained by the parsing explained in the Section 4.1.4 for every message.

The message context holds a function with pre-conditions and post-conditions that are ready for the SMT Solver to use when one message is created in the system. The SMT Solver function results from the message parsing, containing variables that will be generated when that type of message is generated and the annotations written by the developer. However, only unit conditions are used on this SMT Solver function. For example, in the Twist Message, it will include the variable x, y and z for *linear* and *angular*.

The other conditions are saved in another SMT Solver function. The system can use this function to check if the variables in the SMT Solver context are respecting them. For example, if the *Twist* message has one annotation that *move.linear.x* should be bigger than

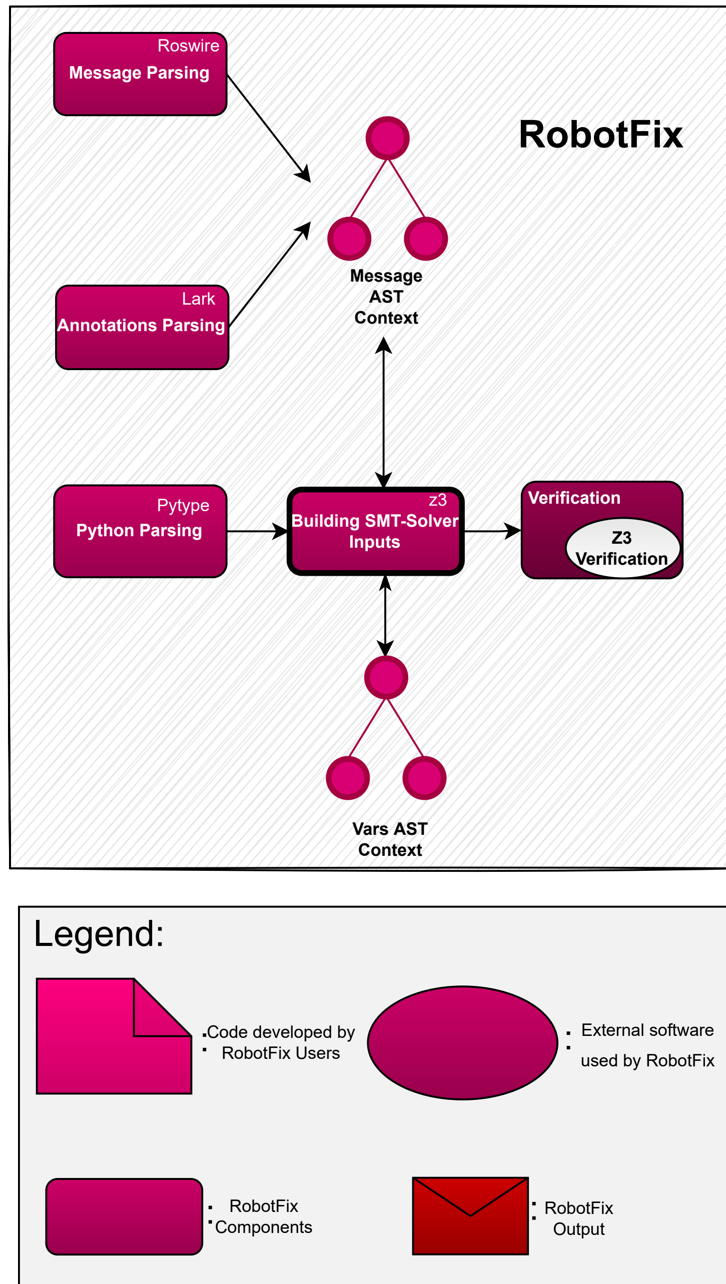


Figure 4.4: Architecture of RobotFix - Context and SMT Solver Inputs.

0, every time that *move.linear.x* will be used, RobotFix checks if the value of *move.linear.x* respect the annotation of being greater than 0.

The message context also contains a third SMT Solver function. This function keeps the values and units from one message type variable from one instance to another (the system creates a new instance of the message each time one variable of message changes). This function is used when one type of message receives updates on one variable and not on the others. It will also update the other variables on the new instance with the old values. For example, if the variable x of linear receives a new update, a new Twist instance should be created with the new value on the variable x of linear. This function will update the other values (y and z from linear and x, textity, and z from angular) with the values on the previous instance.

4.2.2 Variable Context

The variable context has other objectives compared to the message context. The variable context keeps track of the variables created in the Python code.

The variable context also saves the conditions of the variables in a string list.

The variable context holds all instances of a variable, the system creates a new instance of a variable each time the variable changes. When one variable is generated, the system creates the first instance. When one variable is used in the system, it generates a new variable instance, and the variable context saves all of them in the context.

The variable context is essential because each time one variable is called, the system gets the last instance from the context (with the previous value attributed) and saves a new one with the latest information. For example, the first time one variable is defined with one value, it will create the first instance with that value. Then when the variable is determined again, the system will create a new variable instance with the new value while the previous instance will keep the earlier value.

4.2.3 Building SMT Solver Inputs

The SMT Solver chosen for the system was the Z3 solver for its wide feature support, and past success in liquid types Gamboa et al..

The RobotFix approach prepares the input for the SMT Solver in functions. These functions are organized in one context and are responsible for making operations like creating a variable in the Z3 context, adding the unit to a variable, making calculations such as adding a value to a value, assigning one variable to another variable, and finally, adding and minus operations between variables and values.

The outputs from *Pytype* comes with output variables, function names and input variables. These inputs and outputs are built as Z3 objects. Afterwards, the system can run the pre-conditions and post-conditions of the Z3 functions corresponding to the given func-

```
context["assign"] = VFunction(
    outputs=[TVar],
    inputs=[TVar,TVar],
    pre_condition=[lambda ins: var_unit(ins[0]) == var_unit(ins[1])],
    post_condition=[lambda ins, outs: z3.And(var_unit(outs[0]) == var_unit(ins[0]),
                                        var_unit(outs[0]) == var_unit(ins[1]),
                                        var_value(outs[0]) == var_value(ins[1]))],
)
```

Listing 4.3: Example of the "assign" function.

tion name. These conditions are added to the Z3 solver conditions, and Z3 checks if there are inconsistencies.

For example, when x is assigned to y, the output will be x, the function name will be “assign”, and then the input will be “y”. The function “assign” correspond with the Z3 function responsible for the assignments between variables. This function receives one output corresponding to the new instance of x, and two outputs, the previous instance of x and the last instance of y. The Z3 function has pre-conditions to ensure the previous x instance and y had the same unit. The post-conditions define the new instance of x attributes, with the unit of the prior instance of x and the value of y. The code example of this function is represented in Listing 4.3.

The SMT Solver will receive expressions with the assignments made to variables, for example, if x is assigned to the unit *"m/s"* and with the condition “x > 20” the solver will receive the expressions “var_unit(x) == "m/s"” and “"var_value(x) > 20 ”.

However, there are specific cases that need special treatment. One example is when the programmers create a message variable, they initialize more than one variable. For example in Twist, the system needs to initialize the variable x, y and z for linear, and for angular. These messages are not created with the functions in the function context, but they use the functions in the message context that are prepared, that creates all variables in the message type with their units.

4.3 Verification

This section corresponds to the part of the architecture diagram specified in Figure 4.4.

The solver mainly uses the Z3 functions introduced in the last section. The SMT Solver runs the pre-conditions to check if the operation makes inconsistencies in the context. The system runs the contrary of these conditions to check if exists one case when that can be true (where the operation gives an error). If it happens, the system gives back an error; if it is the opposite, it removes these conditions and adds the post-conditions to the Z3 context.

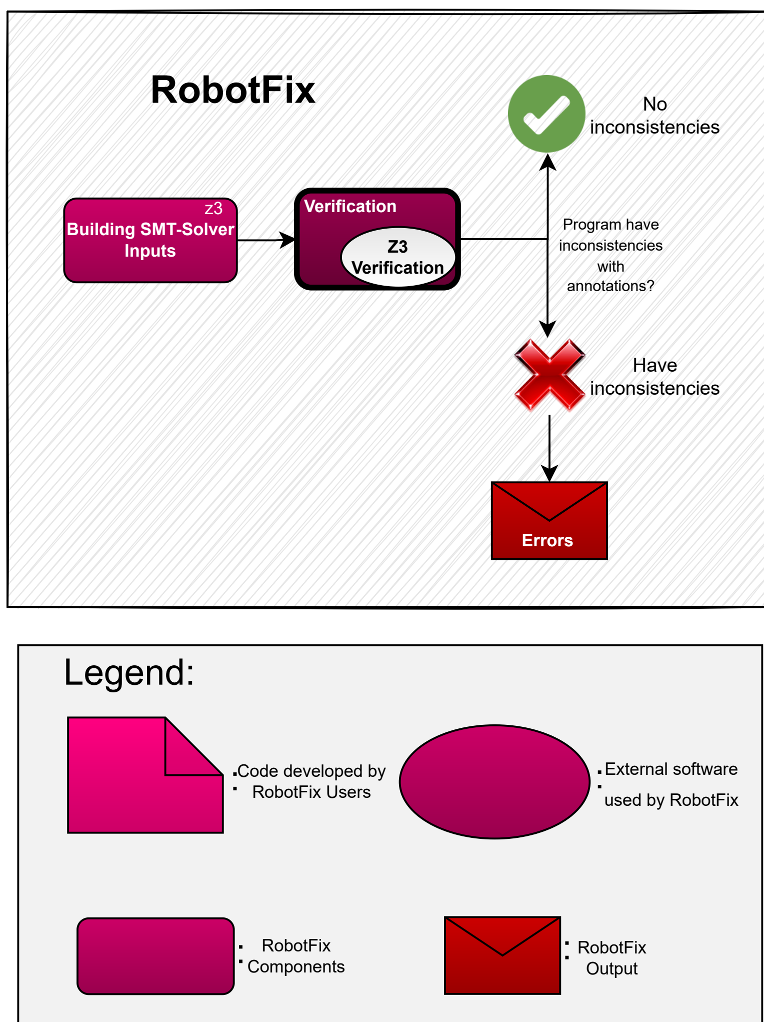


Figure 4.5: Architecture of RobotFix - Verification.

```
1 File "file.py", line 4, in <module>: Type annotation for x does not match type of assignment [
      annotation-type-mismatch]
2 Annotation: int
3 Assignment: float
```

Listing 4.4: Example of a error message provided by Pytype.

The post-conditions stay in the solver context. The system also checks if they gave inconsistencies, and if it happens, it provides errors.

The system also runs the conditions of the used variables in the Z3 solver. These conditions are added to the solver to check if it does not have any inconsistencies. For example, we can have one variable x with the Refinement **x > 10** and the variable assignment value 5. The system needs to check if x is still greater than 10, and in this example, when the condition is added to the solver, it will raise an inconsistency and detect a wrong assignment of the value of x. The behaviour is similar to the Z3 function pre-conditions. If the solver finds inconsistencies, the system will give an error. If the conditions were fulfilling, the system would remove these conditions.

When one value of a message receives an update, it will generate a new message instance and not just a variable instance. Hence, the system also runs other Z3 conditions when one message variable is updated. This function updates the extra values of a message instance to the new one.

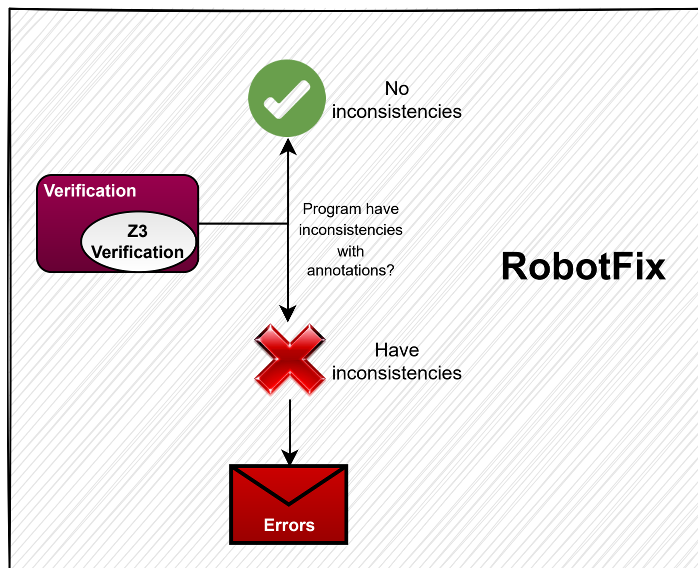
4.4 Errors

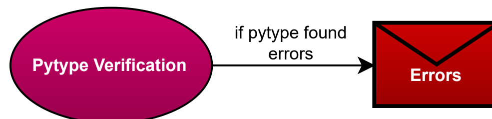
This section corresponds to the part of the architecture diagram specified on the picture Figure 4.6.

The system emits two types of errors, the ones that Pytype found and the ones that Z3 found.The errors found by Pytype give back the error messages that Pytype already emmited. Listing 4.4 represents one example of an error message provided by Pytype. This error says that one variable with int type was assigned with a value in float.

The errors found by the Z3 solver return errors message, these error messages specify the conditions that caused the error and the solver state. With this information, the developers can understand and recover from the mistakes.

Listing 4.5 describes one example of a error provided by RobotFix. This error says that the condition "*timeout* < 200" is inconsistent because *timeout* was assigned with the value 200.





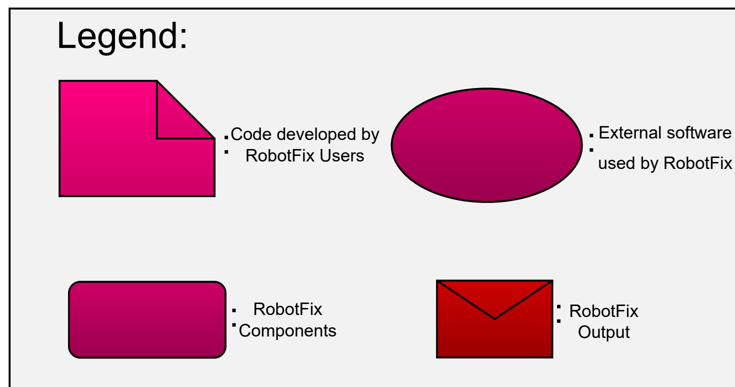


Figure 4.6: Architecture of the ROS Verification System - Errors.

```
ValueError: Condition  var_value( timeout_2 ) < 200 is not satisfied. Example is: [timeout_2
      = 3,
timeout_1 = 2,
var_unit = [else –> "None"],
var_value = [else –> 200]]
```

Listing 4.5: Example of a error message provided by RobotFix.

4.5 Current Limitations of Implementation

Currently, RobotFix does not read all the Python operations. One example is the "if-expression" that is not implemented yet. Pytype can do the sum or the minus operations, but the other numeric expressions are not implemented yet (e.g., division or multiplication).

Although Pytype helped find inconsistencies in the types of the variables. At the same time, I used the already implemented parsing, which gave us some limitations on the work because Pytype was implemented for simple Python code and not for frameworks like ROS, with specific pieces of code. Pytype struggled with ROS code, and I had to make changes in the code to read simple ROS functions correctly. However, ROS has more complex functions that should work easily with RobotFix verification, but Pytype can not parse the code correctly, whitout more changes to Pytype Code.

Pytype also has one issue with importing stubs. This happens because they decided not to use an Ordered dictionary. After all, it will take 40 times longer to make a list. In ROS, we have a lot of packages with dependencies between them, and the order that Pytype ignores is important for ROS Stubs. Pytype outputs errors on the imports, but these errors are on their part and do not exist because of RobotFix.

Chapter 5

Evaluation

This chapter describes the evaluation process made for the approach.

Since Pytype has problems with ROS code, the evaluation was made with simple ROS programs that reproduce errors. These programs are inspired by questions on ROS Answers, bugs identified by ROBUST [27], and bugs identified by Phys [39].

ROS Answers is a website where ROS programmers can ask questions when they have doubts about ROS. Other ROS developers can answer the questions if they have the answer. Basically ROS Answers is the Stack Overflow for ROS.

ROBUST [27] is a ROS bug study that includes a database with ROS Bugs with a description in other information about the bug. Most of the bugs used in the evaluation came from ROBUST.

Phys [39] is a plugin that identifies unit inconsistencies in variables. I use the bugs in the projects identified by Phys. To show that RobotFix can locate the same bugs as Phys with the difference that the programmer has to make the annotations for the units. However, RobotFix should find more bugs than the bugs found by Phys.

5.1 Validity

These tests were created with the limitations of the implementation explained in the Section 4.5. However, I had some requirements for making the tests.

- **R.1 - The test code should have the same objective as the code with the bug:** The tests will be more straightforward than the bug code examples related to them, however, the tests should have the same purpose as the bug code shown that RobotFix should be used to find the bug related to the test.
- **R.2 - The test code should be valid for Python and ROS compiler**: The tests should be proper programs that users can run without RobotFix.
- **R.3 - The test should have a version with an error and one without errors:** The tests with a mistake should show examples of bugs that RobotFix finds and proves

that RobotFix can find those bugs. On the other hand, tests without errors show that if the program is correct, RobotFix does not output unexpected error messages and indicates a change that programmers can make to solve the error on the version with the error.

5.2 Test Examples

The approach can represent the bugs that will be described in the Table 5.1. The table has the test name with the related bug, a simple bug description, and a simple description of the bug instance in the tests.

Test Name	Bug	Bug description	Bug instance
T1 - Appendix A.1	Bug1	Programmer has issues limiting the velocity of the robot	Twist Message Annotated limits the velocity
T2 - Appendix A.2	Bug2	Question about the units for cmd_vel	Message annotated with units
T3 - Appendix A.3	Bug3	One variable has multiple values	Variable assigned with variable with different unit
T4 - Appendix A.4	Bug4	Calculation with wrong type	Sum with different units gives errors
T5 - Appendix A.5	Bug5	Message argument should be one of predefined values	Use of a message that should have one of 3 values
T6 - Appendix A.6	Bug6	Different PI values	Variable annotated with PI and assigned with PI value
T7 - Appendix A.7	Bug7	Added negative seconds	Variable annotated with unit seconds and values bigger or equal to 0
T8 - Appendix A.8	Bug8	Programmer did not initialize important variable on message	Message with annotation used before programmer adds a value
T9 - Appendix A.9	Bug9	Variable should be between 0 and PI	Variable with the condition that his value should be between 0 and PI
T10 - Appendix A.10	Bug10	Missing Frame Conversion to correct type	Variable with frame type assigned to other with different frame type
T11 - Appendix A.11	Bug11	Missing type conversion	Value with one type assigned with other type
T12 - Appendix A.12	Bug12	Wrong Frame assigned	Assignment between variables with different frames
T13 - Appendix A.13	Bug13	Two variables should have different values	Variables assigned with the condition that should have an additional value than the other

Test Name	Bug	Bug description	Bug instance
T14 - Appendix A.14	Bug14	Variable contains undesired values	Variable assigned with the condition that should not have certain values
T15 - Section 5.2.1	Bug15	Timeout Value very big	Timeout with maximum value restriction in the condition
T16 - Section 5.2.2	Bug16	Robot should move forward but moves backwards with wrong calculations	Added the calculations on Message variable that have an annotation that should be bigger than 0
T17 - Appendix A.15	Bug17	Uninitialised variable assigned to timestamps	Value assigned to timestamps have one annotation
T18 - Appendix A.16	Bug18	Computation makes angle wrong	Calculations on a variable with the angular annotated with minimum and maximum value
T19 - Appendix A.17	Bug19	Vector used in Twist Message is inverted	Twist Message annotated with the condition that does not let linear.x being negative
T20 - Appendix A.18	Bug20	ROS message variable is not initialized	ROS Message with condition on variable that make it should be initialized

Table 5.1: Summary of the tests with their related bugs

This section will introduce two of the twenty tests for the reader. These two tests will present two different bugs. The Section 5.2.1 is a simple test representing a bug that is easy to understand. However, the bug on Section 5.2.2 is more complex and showcases a bug relative to calculations. The other tests are in the appendix.

5.2.1 Test 15 - Timeout value is too long

The fifteenth test is inspired on a bug [1] found by ROBUST [27]. In this bug, developers set the timeout too long.

RobotFix can make conditions to restrict these types of values. With RobotFix, developers can add a need to limit the timeout maximum.

The code example on Listing 5.1 has an error found by RobotFix on line 4.

Listing 5.2 represents a code example that respect the annotations on variable *timeout*.

[1] https://github.com/robust-rosin/robust/blob/4ec40a25641317d991486a3a7a65036083e621ce/kobuki/d9aa656/d9aa656.bug

```
from typing_extensions import Annotated

timeout : Annotated[int,"timeout < 200"]
timeout = 200
```

Listing 5.1: Test 15 with error on line 4.

```
from typing_extensions import Annotated

timeout : Annotated[int,"timeout < 200"]
timeout = 20
```

Listing 5.2: Test 15 without errors.

5.2.2 Test 16 - Calculations lead to a unexpected value

The sixteenth test is inspired on a bug [2] found by ROBUST [27]. In this bug, wrong calculations led to unexpected behaviour by the robot.

With RobotFix, developers can make restrictions on messages, removing the hypotheses of having unexpected values.

The message on Figure 5.1a is annotated with conditions that do not allow random values on linear.x. With these annotations, linear.x has to be bigger than 0 and smaller than 20.

The code example on Figure 5.1b has an error found by RobotFix on line 11. This code does not respect the annotation on the Twist message. The result of $x + z$ is -3 that is smaller than 0.

Listing 5.3 represents a code example that respect the annotations on variable *move.linear.x*. The calculation result on line 11 is 7, between the limits 0 and 20.

5.3 Discussion

I have different types of bugs in the list of these 20 bugs. RobotFix is suitable for variables that can not accept all the values. For example, in Test 7 (Appendix A.7), one variable with seconds can not have negative values because negative seconds do not exist. With RobotFix, developers can annotate the variables to exclude negative values. RobotFix finds errors when the programmers do not respect the annotations. These annotations are helpful in other bugs, like in Test 9 (Appendix A.9), where the variable value should be between 2 different values, or in Test 1 (Appendix A.1), where the values inside Twist Message should have a maximum value accordingly with the maximum velocity of the

[2] https://github.com/robust-rosin/robust/blob/4ec40a25641317d991486a3a7a65036083e621ce/kobuki/1c141a5/1c141a5.bug

```
1  geometry_msgs/Twist.msg
2  Vector3 linear
3  #RobotFix# Unit(linear.x) == "m/s" and
    linear.x > 0 and linear.x < 20
4  #RobotFix# Unit(linear.y) == "m/s"
5  #RobotFix# Unit(linear.z) == "m/s"
6  Vector3 angular
7  #RobotFix# Unit(angular.x) == "rad/s"
8  #RobotFix# Unit(angular.y) == "rad/s"
9  #RobotFix# Unit(angular.z) == "rad/s"
10
```

(a) Example of a message Twist annotated for Test 16.

```
1  from typing_extensions import
    Annotated
2  import rospy
3  from geometry_msgs.msg import Twist
4
5  pub = rospy.Publisher('/cmd_vel', Twist
    , queue_size=1)
6  x : Annotated[float, "Unit('m/s')"]
7  z : Annotated[float, "Unit('m/s')"]
8  x = 2
9  z = −5
10 move = Twist()
11 move.linear.x = x + z
12 pub.publish(move)
13
14 rospy.spin()
15
```

(b) Test 16 with error on line 11.

Figure 5.1: Test 16 - Calculations lead to a unexpected value : ROS Message and Python Code.

```
from typing_extensions import Annotated
import rospy
from geometry_msgs.msg import Twist

pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
x : Annotated[float, "Unit('m/s')"]
z : Annotated[float, "Unit('m/s')"]
x = 2
z = 5
move = Twist()
move.linear.x = x + z
pub.publish(move)

rospy.spin()
```

Listing 5.3: Test 16 without errors.

robot.

I found some bugs where programmers write the PI value with different decimal cases(Test 6 - Appendix A.6). PI is used in radians that are used in robotics, and developers write the value with different decimal cases and provoke unexpected behaviours. RobotFix annotations can restrict the correct PI value, causing errors when programmers write the PI value with more decimal cases.

In some ROS programs, ROS message variables should be initialized before the message is used. Test 20 (Appendix A.18) represents one bug caused by one ROS message variable not being initialized. When developers use RobotFix annotations, the values used in the annotations need to be initialized before the message is used. If it is not initialized, RobotFix detects and alerts the programmer.

Tests 16 (Section 5.2.2) and 18 (Appendix A.16) represent bugs where calculations led to unexpected values. With RobotFix, developers can annotate the variable that has the result of the measures that can define an interval of values that the calculations should be inside. RobotFix was successful in finding errors with unexpected values inside variables.

Other types of tests were related to bugs related to Unit or Frame types. Some programmers do not know what units are used in the variables (Test 16 - Section 5.2.2) or do assignments with variables with different units (Test 4 Appendix A.4). RobotFix let developers assign the unit and give an error if the unit is not respected in the code. The same happens with frame types, and RobotFix can be used similarly for Units to Frame Types (Tests 10 - Appendix A.10 and 12 - Appendix A.12).

5.4 Conclusion

RobotFix proved to be useful for robot programmers that want their variables to be used in the right way. Using variables with unexpected values, including initialized values or faulty units, can lead to unexpected behaviours. RobotFix is good at finding this type of errors in different situations. Calculations are difficult to predict, but developers usually have one idea of the result value or at least values that should not be attributed to the variable. RobotFix can check if the result value is in a valid interval. The result was promising because RobotFix, with its actual limitations, could find various types of errors.

Chapter 6

Future Work

This chapter presents future directions for continuing the work and improving the tool.

6.1 Create a parser for Python and C++

During the work, Pytype had a lot of trouble with ROS, and Pytype is not ready to be used in complex ROS environments. There is space to improve and create one parser for Python and make RobotFix address the verification of the variable types. Since developers also use C++ with ROS, there is space to create the parser for C++ and use the rest of the system with it, not restricting the language that the users of the tool can use.

6.2 Implement other Python functionalities

RobotFix have only simple and essential Python operations implemented. There is space to implement to model other Python and ROS functionalities. With this improvement, developers can write more annotations, and RobotFix will find errors in parts of the code that it can not access right now.

6.3 Modeling unit transformations

RobotFix have only sum and minus operations implemented, but when we add multiplication and division, the variable resulting from these operations will have a different unit than the variables that make the operation. For example, when we multiply one variable in meters by one in seconds, the result will be in meters per second. There is space to implement this functionality in RobotFix to keep track of the unit after developers make calculations.

6.4 Let developers create their functions

Developers can only use the Unit function, which is used to define the variable unit. In the future, developers could create their functions and use them the way they want. One example is that developers can create a function to register the values of the frame type instead of using the Unit function.

6.5 Create suggestions for developers while they program

Developers need to create the annotations and set the unit for them selfs. In the future, RobotFix can use other tools that try finding the variable unit, like Phys [39], to make suggestions for the users. These implementations would be made in a plugin for an IDE to make suggestions while developers program and reduce their work doing annotations.

6.6 Improve error messages

In the future, there is space to improve error messages to reduce unnecessary information and make it easier and faster for developers to understand where the error is and the reason for the error.

Chapter 7

Conclusion

ROS is a widely used framework for programming robots. However, programming is hard, more so in complex and uncertain scenarios like robotics. It is difficult and expensive for companies to make tests for robotic programs, and it can be even more costly if the robot produces wrong behaviours while working.

I developed RobotFix to help developers by finding mistakes in their programs when unexpected values or assignments are attributed to variables. With RobotFix, programmers can specify conditions of variables restricting their possible values or assign the variable unit. RobotFix will try to find inconsistencies between variables and their values or assignments.

RobotFix introduces Liquid Types in Python. Liquid Types were already successful in other languages and fitted very well with the Robotic world. Robotic uses numbers for the time, velocity, angles, etc. that should be realistic. With Liquid Types, we can restrict the values of these variables and restrict unexpected and unrealistic values, avoiding bugs.

RobotFix parses the code of developers, and ROS messages are used to have the code information. Then the data is transformed into a query to an SMT Solver, responsible for deciding if all the implications it received (from annotations and python code) are valid and do not exist inconsistencies.

If the SMT Solver says that the program is invalid, RobotFix will give the programmers an error message with the information to find the error.

The evaluation was made with 20 simple tests inspired by 20 real bugs in robotics software. RobotFix was successful in finding all the errors related to variables. The bugs in the evaluation showed the flexibility that RobotFix has to find unexpected values or assignments with wrong units.

In the future, RobotFix should involve a tool that is used with all Python and ROS complexities and also work with C++. RobotFix should have an IDE integration that gives suggestions on annotations to the users.

Appendix A

Test Description

A.1 Test 1 - Defining limits in the robot velocity

The first test is inspired in one question on ROS Answers, the question [3] is related to the fact that the programmer has problems setting the minimum velocity of the robot in one YAML file.

With RobotFix, developers can annotate the messages like in the Figure A.1a. With these annotations, the values for linear.x in the type Twist should always be greater than 0.

The code example on Figure A.1b has an error found by RobotFix on line 10. The correct version of this code is on Listing A.1

A.2 Test 2 - Defining the variable units on Twist Message.

The second test is inspired by other questions on ROS Answers, the question [4] is related to the fact that the programmer does not know the units that Twist message use.

With RobotFix, developers can annotate the messages like in the Figure A.2a. And later, when inexperienced programmers use the message already annotated, they can check which unit they should use. In this case Twist uses **"m/s"** in the linear vector, and **"rad/s"** in the angular vector.

The code example on Figure A.2b has an error found by RobotFix on line 11.

This bug is related to this question because the unit introduced on the *move.linear.z* variable is wrong. The error message says that z and move.linear.z have different units. With this information the programmer can change the unit of *z* to **"rad/s"** and the programs runs without RobotFix errors. The correct version of this code is on Listing A.2

[3]https://answ ers.ros.org/question/376679/cmd_vel-values-is-lower-than-my-limits/

[4]https://answers.ros.org/question/55977/units-for-cmd_vel/

```
geometry_msgs/Twist.msg
    Vector3 linear
    #RobotFix# Unit(linear.x) == "m/s"
    #RobotFix# Unit(linear.y) == "m/s"
    #RobotFix# Unit(linear.z) == "m/s"
    #RobotFix# linear.x > 0
    Vector3 angular
    #RobotFix# Unit(angular.x) == "rad/s"
    #RobotFix# Unit(angular.y) == "rad/s"
    #RobotFix# Unit(angular.z) == "rad/s"
```

(a) Example of a message Twist annotated for defining limits in the robot velocity.

```
from typing_extensions import Annotated
import rospy
from geometry_msgs.msg import Twist

pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
x : Annotated[float, "Unit('m/s')"]
x = -10

move = Twist()
move.linear.x = x
pub.publish(move)

rospy.spin()
```

(b) Test 1 with error on line 10.

Figure A.1: Test 1 - Defining limits in the robot velocity: ROS Message and Python Code.

```python
from typing_extensions import Annotated
import rospy
from geometry_msgs.msg import Twist

pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
x : Annotated[float, "Unit('m/s')"]
x = 5

move = Twist()
move.linear.x = x
pub.publish(move)

rospy.spin()
```

Listing A.1: Test 1 without errors.

```python
from typing_extensions import Annotated
import rospy
from geometry_msgs.msg import Twist
pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
x : Annotated[float, "Unit('m/s')"]
z : Annotated[float, "Unit('rad/s')"]
x = 5
z = 6
move = Twist()
move.linear.x = x
move.angular.x = z
pub.publish(move)

rospy.spin()
```

Listing A.2: Test 2 without errors.

```
1  geometry_msgs/Twist.msg
2    Vector3 linear
3    #RobotFix# Unit(linear.x) == "m/s"
4    #RobotFix# Unit(linear.y) == "m/s"
5    #RobotFix# Unit(linear.z) == "m/s"
6    Vector3 angular
7    #RobotFix# Unit(angular.x) == "rad/s"
8    #RobotFix# Unit(angular.y) == "rad/s"
9    #RobotFix# Unit(angular.z) == "rad/s"
10
11
```

(a) Example of a message Twist annotated for Test 2.

```
1      from typing_extensions import
        Annotated
2      import rospy
3      from geometry_msgs.msg import Twist
4      pub = rospy.Publisher('/cmd_vel', Twist
        , queue_size=1)
5      x : Annotated[float, "Unit('m/s')"]
6      z : Annotated[float, "Unit('degree/s')"]
7      x = 5
8      z = 6
9      move = Twist()
10     move.linear.x = x
11     move.angular.z = z
12     pub.publish(move)
13
14     rospy.spin()
15
```

(b) Test 2 with error on line 11.

Figure A.2: Test 2 - Defining the variable units on Twist Message : ROS Message and Python Code.

```
1  from typing_extensions import Annotated
2
3  z : Annotated[float, "Unit('quaternion')"]
4  z = 6
5  a : Annotated[float, "Unit('meter')"]
6  b : Annotated[float, "Unit('quaternion')"]
7  a = 10
8  b = 20
9  z = b
10 a = z
```

Listing A.3: Test 3 Code with error on line 10.

A.3 Test 3 - Assignment between two variables with different units

The third test is based on a bug on Python File [5] found by Phys [39]. In this bug, the variable textivec_out.z should have the unit *meters* and be assigned one variable with the unit *quartenion*.

The code example on Listing A.3 has an error found by RobotFix on line 10.

With RobotFix, the variables should not be assigned to others with different units. RobotFix finds an error on line 10 because it set a variable with the unit **quarternion** to one variable with the unit **meter**. To assign one value to *a*, it should be with a variable with the same unit, like in the case of line 9.

RobotFix also finds these type of errors on ROS Messages. Appendix A.2 is one example of that.

Listing A.4 represents an example of a code for Test 3 that developers can make so that RobotFix does not find errors.

A.4 Test 4 - Calculations lead to an unexpected value

The fourth test is inspired on a bug [6] found by ROBUST [27]. In this bug, wrong calculations made the robot have unexpected behaviour.

With RobotFix, developers can restrict the values of variables. With this functionality, developers can limit a result of a calculation on the result variable. This way, RobotFix will check if the result is within a margin that makes sense for the developer avoiding unexpected behaviours.

The code example on Listing A.5 has an error found by RobotFix on line 11.

[5] https://github.com/ashish-kb/bayesian_compact_ware/blob/master/ard/src/ard_node.cpp
[6] https://github.com/robust-rosin/robust/blob/mas ter/geometry2/da141a5/da141a5.bug

```
from typing_extensions import Annotated

x : Annotated[float, "Unit('meter')"]
z : Annotated[float, "Unit('quaternion')"]
x = 5
z = 6
a : Annotated[float, "Unit('meter')"]
b : Annotated[float, "Unit('quaternion')"]
a = 10
b = 20
x = a
z = b
```

Listing A.4: Test 3 Code without errors.

```
from typing_extensions import Annotated
import rospy
from geometry_msgs.msg import Twist

pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
x : Annotated[float, "Unit('m/s')"]
z : Annotated[float, "Unit('m/s')"]
y : Annotated[float, "Unit('m/s') and _ > 0 and _ < 20"]
x = 10
z = 20
y = x + z
move = Twist()
move.linear.x = y
pub.publish(move)

rospy.spin()
```

Listing A.5: Test 4 with error on line 11.

```
from typing_extensions import Annotated
import rospy
from geometry_msgs.msg import Twist

pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
x : Annotated[float, "Unit('m/s')"]
z : Annotated[float, "Unit('m/s')"]
y : Annotated[float, "Unit('m/s') and _ > 0 and _ < 20"]
x = 10
z = 5
y = x + z
move = Twist()
move.linear.x = y
pub.publish(move)

rospy.spin()
```

Listing A.6: Test 4 without errors.

The error happens because the sum resulted in a bigger value than the restriction on the annotation, the result was 100, and the variable had a condition that the variable value should be smaller than 90. RobotFix tells the user that he found an inconsistency between the annotation and the variable value, and the programmer can fix the sum.

Listing A.6 presents the same code with different values that lead to a correct result consistent with the variable annotation.

A.5 Test 5 - Variable assigned with a unexpected value

The Fifth test is inspired on a bug [7] found by ROBUST [27]. In this bug, a programmer gave a unexpected value to a message variable. This message variable should have one of 39 pre-defined strings.

RobotFix is really helpful with this type of messages that have pre-defined values. A developer can add a annotation with that information and it will lead to a error when someone breaks that annotation.

Figure A.3a is a example of a message annotated with a condition that restrict the values into specific ones. This case is numbers because strings are not implemented yet.

The code example on Figure A.3b has an error found by RobotFix on line 12. RobotFix finds a inconsistency on line 5 because the *v.data* does not have one of the 3 pre-defined values.

Listing A.7 presents the same code with one of the 3 pre-defined values. It gives no RobotFix errors.

[7] https://github.com/robust-rosin/robust/blob/master/mavros/4fb6e7e/4fb6e7e.bug

```
[std_msgs/Int32]:

  int32 data #RobotFix# data == 1 or data
   == 2 or data == 3


```

(a) Example of a message Twist annotated for Test 2.

```
from std_msgs import Int32

v = Int32()

v.data = 10

```

(b) Test 5 with error on line 5.

Figure A.3: Test 5 - Variable assigned with a unexpected value : ROS Message and Python Code.

```
from std_msgs import Int32

v = Int32()

v.data = 1
```

Listing A.7: Test 5 without errors.

```
from typing_extensions import Annotated

x : Annotated[float, " _ >= 3.141592653589793238 "]
x = 3.14
```

Listing A.8: Test 6 with error on line 4.

```
from typing_extensions import Annotated

x : Annotated[float, " _ >= 3.141592653589793238 "]
x = 3.141592653589793238
```

Listing A.9: Test 6 without errors.

A.6 Test 6 - Different values for PI

Two bugs found by ROBUST [27] inspires the Sixth test: bug 1 [8] and bug 2 [9]. These two bugs have one issue in common. The programmer wrote the value of PI in a different way than was expected. In Robot programs, the PI value is usually set with varying decimal cases. The inconsistent value of PI is an issue and can make the robot do an unexpected behaviour. For example, one *if* condition can be false and change the behaviour of the robot.

RobotFix allows the programmer to make annotations restricting the values to a specific PI value. And if the programmer introduces a value that does not respect that annotation RobotFix will have an error message with that information.

The code example on Listing A.8 has an error found by RobotFix on line 4.

RobotFix finds an inconsistency on line 4. The written PI value on line 4 is different and smaller than the one on the restriction in *x* condition.

Listing A.9 presents the same code with the correct PI value. It gives no RobotFix errors.

A.7 Test 7 - Variable assigned with a negative value for seconds

The seventh test is inspired on a bug [10] found by Phys [39]. In this bug, the programmer assigned a negative value to a variable with the unit **"seconds"**. One variable with the unit **"seconds"** should always have positive values.

[8] https://github.com/robust-rosin/robust/blob/master/motoman/2d42582/2d42582.bug

[9] https://github.com/robust-rosin/robust/blob/master/motoman/2d42582/2d42582.bug

[10] https://github.com/RobotnikAutomation/summit_xl_sim/blob/hydro-multi/summit_xl_robot_control/src/summit_xl_robot_control.cpp #L576

```
[std_msgs/Float64]:
float64 data #RobotFix# Unit(data) == "
 seconds" and data >= 0

```

(a) Example of a message Float annotated for Test 7.

```
from std_msgs import Float64

time = Float64()

time.data = −1

```

(b) Test 7 with error on line 5.

Figure A.4: Test 7 - Variable assigned with a negative value for seconds : ROS Message and Python Code.

```
from std_msgs import Float64

time = Float64()

time.data = 1
```

Listing A.10: Test 7 without errors.

With RobotFix, programmers can add to a variable the unit and **"seconds"** and the annotation that values should be positive.

Figure A.4a is an example of a message that saves the value of time in seconds. The message is annotated with the condition that it should receive one value in **seconds** and that value should be greater or equal to 0.

The code example on Figure A.4b has an error found by RobotFix on line 5.

RobotFix finds an inconsistency on line 5 because the *time.data* received a value negative that does not respect the condition of the message.

Listing A.10 presents the same code with one positive value on *time.data*. RobotFix does not find inconsistencies in this code.

```
from typing_extensions import Annotated
import rospy
from geometry_msgs.msg import Twist

pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
y : Annotated[float, "Unit('m/s')"]
y = 5
move = Twist()
move.linear.x = 1
move.linear.y = y
pub.publish(move)

rospy.spin()
```

Listing A.11: Test 8 without errors.

A.8 Test 8 - Variable was not initialized

The eigth test is inspired on a bug [11] found by ROBUST [27]. In this bug, the developers did not initialize one variable.

Variable with the type of a Message with RobotFix annotations on their values should be initialized before the Message is used, or RobotFix will find an error.

Figure A.5a is an example of a message annotating textitlinear.x. With this annotation, textitlinear.x has to be initialized.

The code example on Figure A.5b has an error found by RobotFix on line 9.

RobotFix finds an inconsistency on line 9 because the *move.linear.x* was not initialized yet.

Listing A.11 presents the same code with *move.linear.x* initialized first. RobotFix found zero inconsistencies.

A.9 Test 9 - Unexpected range value assigned to one variable

The ninth test is inspired on a bug [12] found by ROBUST [27]. In this bug, the developers allowed every range when the range should be between 0 and PI to make sense for the program.

With RobotFix, developers can add annotations to variable definitions restricting their values between 0 and PI. The annotated message represented on Figure A.6a has this type of annotation.

[11] https://github.com/robust-rosin/robust/blob/master/geometry2/d50d51b/d50d51b.bug
[12] https://github.com/bulletphysics/bullet3/commit/e80bafdf6639429193ca5601d195c54f7793a416

```
1  geometry_msgs/Twist.msg
2      Vector3 linear
3      #RobotFix# Unit(linear.x) == "m/s" and
        linear.x > 0
4      #RobotFix# Unit(linear.y) == "m/s"
5      #RobotFix# Unit(linear.z) == "m/s"
6      Vector3 angular
7      #RobotFix# Unit(angular.x) == "rad/s"
8      #RobotFix# Unit(angular.y) == "rad/s"
9      #RobotFix# Unit(angular.z) == "rad/s"
10
11
```

(a) Example of a message Twist annotated for Test 8.

```
1      from typing_extensions import
        Annotated
2      import rospy
3      from geometry_msgs.msg import Twist
4
5      pub = rospy.Publisher('/cmd_vel', Twist
        , queue_size=1)
6      y : Annotated[float, "Unit('m/s')"]
7      y = 5
8      move = Twist()
9      move.linear.y = x
10     pub.publish(move)
11
12     rospy.spin()
13
```

(b) Test 8 with error on line 9.

Figure A.5: Test 8 - Variable was not initialized : ROS Message and Python Code

```
1  [geometry_msgs/Quaternion]:
2  # This represents an orientation in free
     space in quaternion form.
3
4  float64 x #RobotFix# x >= 0 and x <=
     3.14
5  float64 y
6  float64 z
7  float64 w
8
```

(a) Example of a message Quarternion annotated for Test 9.

```
1  from geometry_msgs import Quaternion
2
3  qt = Quaternion()
4  qt.x = 4
5
6
```

(b) Test 9 with error on line 4.

Figure A.6: Test 9 - Unexpected range value assigned to one variable: ROS Message and Python Code.

```
from geometry_msgs import Quaternion

qt = Quaternion()
qt.x = 1
```

Listing A.12: Test 9 without errors.

```
from typing_extensions import Annotated

variableFrame : Annotated[int,"Unit('NED')"]
x : Annotated[int,"Unit('ENU')"]
x = 1
send_attitude_target = x
```

Listing A.13: Test 10 with error on line 6.

The code example on Figure A.6b has an error found by RobotFix on line 4.

RobotFix finds an inconsistency on line 4 because the *qt.x* is bigger than 3.14.

Listing A.12 presents the same code with *qt.x* initialized with a value between 0 and 3.14. RobotFix found zero inconsistencies.

A.10 Test 10 - Assignment between variables with different type of frames

The tenth test is inspired on a bug [13] found by ROBUST [27]. In this bug, the developers tried to do an assignment between two variables with different types of frames.

With RobotFix, developers can define the unit of frame variables with the frame type. If they make an assignment with two variables with different frames, RobotFix will find inconsistencies and return an error.

The code example on Listing A.13 has an error found by RobotFix on line 6.

RobotFix finds an inconsistency on line 6 because the *x* is assigned to *send_attitude_target* and they have an different frame type.

Listing A.14 presents the same code with *x* and *send_attitude_target* with the same frame type as unit. RobotFix found zero inconsistencies in this code example.

[13]https://github.com/robust-rosin/robust/blob/master/mavros/248cb38/248cb38.bug

```
1 from typing_extensions import Annotated
2
3 variableFrame : Annotated[int,"Unit('NED')"]
4 x : Annotated[int,"Unit('NED')"]
5 x = 1
6 variableFrame = x
```

Listing A.14: Test 10 without errors.

```
1 from typing_extensions import Annotated
2
3 x : Annotated[int,""]
4 x = 2.0
```

Listing A.15: Test 11 with error on line 4.

A.11 Test 11 - Assignment between variables with different types

The eleventh test is inspired on a bug [14] found by ROBUST [27]. The developers assigned values with different types on the same variable in this bug.

With RobotFix, Pytype verifies if the program has inconsistencies with the variable types. If pytype finds inconsistencies like this bug, it will return an error.

The code example on Listing A.15 has an error found by RobotFix on line 4.

RobotFix finds an inconsistency on line 4 because the x is set with a value of type float.

Listing A.16 presents the same code with x with one value that respect the annotated type.

[14] https://github.com/robust-rosin/robust/blob/master/mavros/a67d81d/a67d81d.bug

```
1 from typing_extensions import Annotated
2
3 x : Annotated[int,""]
4 x = 1
```

Listing A.16: Test 11 without errors.

```
from typing_extensions import Annotated

variableFrame : Annotated[int,"Unit('ENU')"]
x : Annotated[int,"Unit('NED')"]
x = 1
variableFrame = x
```

Listing A.17: Test 12 with error on line 6.

```
from typing_extensions import Annotated

variableFrame : Annotated[int,"Unit('ENU')"]
x : Annotated[int,"Unit('ENU')"]
x = 1
variableFrame = x

```

Listing A.18: Test 12 without errors.

A.12 Test 12 - Assignment between variables with different type of frames

The twelfth test is inspired on a bug [15] found by ROBUST [27]. In this bug, the developers tried to do an assignment between two variables with different types of frames. This bug is similar to the one on Appendix A.12.

With RobotFix, developers can define the unit of frame variables with the frame type. If they make an assignment with two variables with different frames, RobotFix will find inconsistencies and return an error.

The code example on Listing A.17 has an error found by RobotFix on line 6.

RobotFix finds an inconsistency on line 6 because the x is assigned to $variableFrame$ and they have an different frame type.

Listing A.18 presents the same code with x and $variableFrame$ with the same frame type as unit. RobotFix found zero inconsistencies in this code example.

A.13 Test 13 - Two variables should have different values

The thirty tenth test is inspired on a bug [16] found by ROBUST [27]. In this bug, developers introduced a wrong condition to check if two Messages have the same value.

[15] https://github.com/robust-rosin/robust/blob/master/mavros/b96bf67/b96bf67.bug

[16] https://github.com/robust-rosin/robust/blob/041df50682ae49a32b2100330de84ce6f3a3d9e5/geometry2/c81fa72/c81fa72.bug

```
from typing_extensions import Annotated
import rospy
from geometry_msgs.msg import Twist

move = Twist()

move.linear.x = 1

move2 = Twist()
move2.linear.x = 2

x : Annotated[float,"Unit('m/s')"]
x = move.linear.x

y : Annotated[float,"Unit('m/s') and y == x"]
y = move2.linear.x
rospy.spin()
```

Listing A.19: Test 13 with error on line 16.

With RobotFix, developers can make conditions to check or double-check if the ROS messages have the same value on the variable.

The code example on Listing A.19 has an error found by RobotFix on line 16.

RobotFix finds an inconsistency on line 16 because the y have a different value of x. The y has the value of move2.linear.x and x has the value of *move.linear.x*. The condition on the y implies that x and y have the same value, therefore move.linear.x and move2.linear.x will also have the same value

Listing A.20 presents the same code where *move.linear.x* and *move2.linear.x* have the same value . RobotFix found zero inconsistencies in this code example.

A.14 Test 14 - Variable with unexpected value

The fourteenth test is inspired on a bug [17] found by ROBUST [27]. In this bug, developers had a variable that should not have some characters.

RobotFix can make conditions to restric values that they do not want to have in their variables.

The code example on Listing A.21 has an error found by RobotFix on line 4. This example is with *int variable* instead of *strings*

RobotFix finds an inconsistency on line 4, because x have one condition that restriction that x should not be 1 , and textitx is set with the value 1.

[17] https://github.com/robust-rosin/robust/blob/041df50682ae49a32b2100330de84ce6f3a3d9e5/confidential/81d0a3c/81d0a3c.bug

```python
from typing_extensions import Annotated
import rospy
from geometry_msgs.msg import Twist

move = Twist()

move.linear.x = 1

move2 = Twist()
move2.linear.x = 1

x : Annotated[float,"Unit('m/s')"]
x = move.linear.x

y : Annotated[float,"Unit('m/s') and y == x"]
y = move2.linear.x
rospy.spin()
```

Listing A.20: Test 13 without errors.

```python
from typing_extensions import Annotated

x : Annotated[int,"x != 1"]
x = 1
```

Listing A.21: Test 14 with error on line 4.

```
1 from typing_extensions import Annotated
2
3 x : Annotated[int,"x != 1"]
4 x = 2
```

Listing A.22: Test 14 without errors.

```
1 from typing_extensions import Annotated
2
3 timeout : Annotated[int,"timeout < 200"]
4 timeout = 200
```

Listing A.23: Test 15 with error on line 4.

Listing A.22 represents a code example that respect the annotations on variable *x*

A.14.1 Test 15 - Timeout value is too long

The fifteenth test is inspired on a bug [18] found by ROBUST [27]. In this bug, developers set the timeout too long.

RobotFix can make conditions to restrict these types of values. With RobotFix, developers can add a need to limit the timeout maximum.

The code example on Listing A.23 has an error found by RobotFix on line 4.

Listing A.24 represents a code example that respect the annotations on variable *timeout*.

A.14.2 Test 16 - Calculations lead to a unexpected value

The sixteenth test is inspired on a bug [19] found by ROBUST [27]. In this bug, wrong calculations led to unexpected behaviour by the robot.

[18]https://github.com/robust-rosin/robust/blob/4ec40a25641317d991486a3a7a65036083e621ce/kobuki/d9aa656/d9aa656.bug

[19]https://github.com/robust-rosin/robust/blob/4ec40a25641317d991486a3a7a65036083e621ce/kobuki/1c141a5/1c141a5.bug

```
1 from typing_extensions import Annotated
2
3 timeout : Annotated[int,"timeout < 200"]
4 timeout = 20
```

Listing A.24: Test 15 without errors.

```python
from typing_extensions import Annotated
import rospy
from geometry_msgs.msg import Twist

pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
x : Annotated[float, "Unit('m/s')"]
z : Annotated[float, "Unit('m/s')"]
x = 2
z = 5
move = Twist()
move.linear.x = x + z
pub.publish(move)

rospy.spin()
```

Listing A.25: Test 16 without errors.

With RobotFix, developers can make restrictions on messages, removing the hypotheses of having unexpected values.

The message on Figure A.7a is annotated with conditions that do not allow random values on linear.x. With these annotations, linear.x has to be bigger than 0 and smaller than 20.

The code example on Figure A.7b has an error found by RobotFix on line 11. This code does not respect the annotation on the Twist message. The result of $x + z$ is -3 that is smaller than 0.

Listing A.25 represents a code example that respect the annotations on variable *move.linear.x*. The calculation result on line 11 is 7, between the limits 0 and 20.

A.15 Test 17 - Variable not being initialized

The seventeenth test is inspired on a bug [20] found by ROBUST [27]. In this bug, one variable has not initialized, leading to incorrect timestamps being produced.

With RobotFix, developers can make annotations on messages. These annotations require the variable to be initialized or create a condition that requires the variable to receive an initialized value.

The Listing A.26 represent the Test 17. In this test x is annotated with an empty condition. This variable is used to define the value of *timestamp* but was not initialized. RobotFix finds an inconsistency because line 7 does not respect the condition that *timestamp* should be smaller than 10.

[20]https://github.com/robust-rosin/robust/blob/54ec3344dda74b19a8dcb856d5e89617f676e9bf/mavros/fcf9cd9/fcf9cd9.bug

```
geometry_msgs/Twist.msg
Vector3 linear
#RobotFix# Unit(linear.x) == "m/s" and
 linear.x > 0 and linear.x < 20
#RobotFix# Unit(linear.y) == "m/s"
#RobotFix# Unit(linear.z) == "m/s"
Vector3 angular
#RobotFix# Unit(angular.x) == "rad/s"
#RobotFix# Unit(angular.y) == "rad/s"
#RobotFix# Unit(angular.z) == "rad/s"

```

(a) Example of a message Twist annotated for Test 16.

```
from typing_extensions import
 Annotated
import rospy
from geometry_msgs.msg import Twist

pub = rospy.Publisher('/cmd_vel', Twist
 , queue_size=1)
x : Annotated[float, "Unit('m/s')"]
z : Annotated[float, "Unit('m/s')"]
x = 2
z = −5
move = Twist()
move.linear.x = x + z
pub.publish(move)

rospy.spin()

```

(b) Test 16 with error on line 11.

Figure A.7: Test 16 - Calculations lead to a unexpected value : ROS Message and Python Code.

```
from typing_extensions import Annotated

x : Annotated[int,""]

timestamp : Annotated[int, "_ < 10"]

timestamp = x
```

Listing A.26: Test 17 with error on line 7.

```
from typing_extensions import Annotated

x : Annotated[int,""]

timestamp : Annotated[int, "_ < 10"]

x = 5

timestamp = x
```

Listing A.27: Test 17 without errors.

Listing A.27 represents a code example that respect the annotations on variable *timestamp*. The variable *x* was defined on line 7 and *timestamp* will have this value.

A.16 Test 18 - Calculations lead to a unexpected value

The eighteenth test is inspired on a bug [21] found by ROBUST [27]. In this bug, wrong calculations made the robot move backwards.

With RobotFix, developers can annotate the result of calculations restricting unexpected results of calculations.

The Listing A.28 has a variable with conditions and one error on calculations. The conditions on line 12 prevent unexpected results in line 13. In this line, the calculation resulted in a -1 that does not respect the annotation. This code leads to an inconsistency found by RobotFix on line 12.

Listing A.29 represents a code example that respects the annotations on line 11, and the result of the calculation on line 12 respect the annotation being 2, which is between 0 and 3.

A.17 Test 19 - Calculations lead to a unexpected value

The nineteenth test is inspired on a bug [22] found by ROBUST [27]. In this bug, developers set the vector with the inverted angle. The robot went in the opposite direction of what the developers expected.

With RobotFix, developers can restrict messages, removing the hypotheses of having unexpected values that will lead to unexpected behaviours.

[21] https://github.com/robust-rosin/robust/blob/4ec40a25641317d991486a3a7a65036083e621ce/geometry2/439e235/439e235.bug

[22] https://github.com/robust-rosin/robust/blob/4ec40a25641317d991486a3a7a65036083e621ce/kobuki/e964bbb/e964bbb.bug

```
from typing_extensions import Annotated
import rospy
from geometry_msgs.msg import Twist

pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
x : Annotated[float, "Unit('rad/s')"]
z : Annotated[float, "Unit('rad/s')"]
x = 1
z = -2
move = Twist()
move.linear.x = 1
ang : Annotated[float,"Unit('rad/s') and _ > 0 and _ < 3 "]
ang = x + z
move.angular.z = ang
pub.publish(move)

rospy.spin()
```

Listing A.28: Test 18 with error on line 7.

```
from typing_extensions import Annotated
import rospy
from geometry_msgs.msg import Twist

pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
x : Annotated[float, "Unit('rad/s')"]
z : Annotated[float, "Unit('rad/s')"]
x = 1
z = 1
move = Twist()
move.linear.x = 1
ang : Annotated[float,"Unit('rad/s') and _ > 0 and _ < 3 "]
ang = x + z
move.angular.z = ang
pub.publish(move)

rospy.spin()
```

Listing A.29: Test 18 without errors.

```
from typing_extensions import Annotated
import rospy
from geometry_msgs.msg import Twist

pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
move = Twist()
move.linear.x = 1
move.linear.y = 0
move.linear.z = 0
pub.publish(move)

rospy.spin()
```

Listing A.30: Test 19 without errors

The message on Figure A.8a is annotated with conditions that do not allow unexpected values on linear.x. With these annotations, linear.x has to be bigger than 0 and smaller than 20. With these conditions, the robot can not go backwards.

The code example on Figure A.8b introduces the vector (-1,0,0) on *move.linear*. This vector makes the robot move backwards. This behaviour was not expected, and the annotations on Twist Message made these values produce an inconsistency found by RobotFix.

Listing A.30 represents a code example that respect the annotations on variable *move.linear.x*. The vector introduced on *move.linear* is (1,0,0) and will move forward as expected.

A.18 Test 20 - ROS Message variable used but not initiliazed

The twentieth test is inspired on a bug [23] found by ROBUST [27]. In this bug, developers used a ROS message variable that was not initialized.

With RobotFix, developers can make annotations on Messages. If the developers use the message with conditions on one variable and the variable has not been initiliazed, it will find an inconsistency in the program.

Figure A.9a is an example of Twist Message with annotations. In this test, the annotations are not important. The important part is that they exist.

The Figure A.9b have an inconsistency found by RobotFix on line 8. This inconsistency exists because Twist Message have annotations, and none of the values of Twist has been initiated.

Listing A.31 represents a code example that *move.linear.x* has been initialized with a value in agreement with the condition on Twist message. RobotFix found zero inconsistencies in this code example.

[23] https://github.com/robust-rosin/robust/blob/67c4b9c3aee2060409e88c27c0e93e798fc43d9f/geometry2/001fca6/001fca6.bug

```
geometry_msgs/Twist.msg
Vector3 linear
#RobotFix# Unit(linear.x) == "m/s" and
 linear.x > 0 and linear.x < 20
#RobotFix# Unit(linear.y) == "m/s"
#RobotFix# Unit(linear.z) == "m/s"
Vector3 angular
#RobotFix# Unit(angular.x) == "rad/s"
#RobotFix# Unit(angular.y) == "rad/s"
#RobotFix# Unit(angular.z) == "rad/s"

```

(a) Example of a message Twist annotated for Test 19.

```
from typing_extensions import
 Annotated
import rospy
from geometry_msgs.msg import Twist

pub = rospy.Publisher('/cmd_vel', Twist
 , queue_size=1)
move = Twist()
move.linear.x = -1
move.linear.y = 0
move.linear.z = 0
pub.publish(move)

rospy.spin()

```

(b) Test 19 with error on line 11.

Figure A.8: Test 19 - Calculations lead to a unexpected value : ROS Message and Python Code.

```
1   geometry_msgs/Twist.msg
2   Vector3 linear
3   #RobotFix# Unit(linear.x) == "m/s"
   and linear.x > 0 and linear.x < 20
4   #RobotFix# Unit(linear.y) == "m/s"
5   #RobotFix# Unit(linear.z) == "m/s"
6   Vector3 angular
7   #RobotFix# Unit(angular.x) == "rad/s"
8   #RobotFix# Unit(angular.y) == "rad/s"
9   #RobotFix# Unit(angular.z) == "rad/s"
10
```

(a) Example of a message Twist annotated for Test 20.

```
1   from typing_extensions import
    Annotated
2   import rospy
3   from geometry_msgs.msg import Twist
4
5   pub = rospy.Publisher('/cmd_vel', Twist
    , queue_size=1)
6   move = Twist()
7   x : Annotated[float, "Unit('m/s')"]
8   x = move.linear.x
9   pub.publish(move)
10
11  rospy.spin()
12
```

(b) Test 20 with error on line 8.

Figure A.9: Test 20 - ROS Message variable used but not initiliazed : ROS Message and Python Code.

```
from typing_extensions import Annotated
import rospy
from geometry_msgs.msg import Twist



pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
move = Twist()
x : Annotated[float, "Unit('m/s')"]
move.linear.x = 1
x = move.linear.x
pub.publish(move)

rospy.spin()
```

Listing A.31: Test 20 without errors.

Bibliography

[1] Carmen robot navigation toolkit. http://carmen.sourceforge.net/, 2021.

[2] Moos. https://www.robots.ox.ac.uk/~mobile/MOOS/wiki/pmwiki.php, 2021.

[3] Orca: Components for robotics. http://orca-robotics.sourceforge.net/, 2021.

[4] The orocos project. https://orocos.org/, 2021.

[5] The player project. http://playerstage.sourceforge.net/, 2021.

[6] actionlib - ros wiki. http://wiki.ros.org/actionlib, 2021.

[7] catkin/cmakelists.txt - ros wiki. http://wiki.ros.org/catkin/CMakeLists.txt, 2021.

[8] catkin/packages.xml - ros wiki. http://wiki.ros.org/catkin/package.xml, 2021.

[9] sensor_msgs/laserscan message documentation. http://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/LaserScan.html, 2021.

[10] roslaunch/xml - ros wiki. http://wiki.ros.org/roslaunch/XML, 2021.

[11] Nodes - ros wiki. http://wiki.ros.org/Nodes, 2021.

[12] Packages - ros wiki. http://wiki.ros.org/Packages, 2021.

[13] Ros - robot operating system. https://www.ros.org/, 2021.

[14] geometry_msgs/pose message documentation. http://docs.ros.org/en/noetic/api/geometry_msgs/html/msg/Pose.html, 2021.

[15] Services- ros wiki. http://wiki.ros.org/Services, 2021.

[16] Topics - ros wiki. http://wiki.ros.org/Topics, 2021.

[17] Welcome to lark's documentation. https://lark-parser.readthedocs.io/en/latest/, 2022.

[18] The origin story of ros, the linux of robotics. https://spectrum.ieee.org/the-origin-story-of-ros-the-linux-of-robotics, 2022.

[19] Pytype - a static tipy analyzer for python code. https://google.github.io/pytype/, 2022.

[20] Ros answers - units for cmd_vel. https://answers.ros.org/question/55977/units-for-cmd_vel/, 2022.

[21] Ros answers - cmd_vel values is lower than my limits. https://answers.ros.org/question/376679/cmd_vel-values-is-lower-than-my-limits/, 2022.

[22] Turtlebot. https://www.turtlebot.com/, 2022.

[23] Roswire. https://github.com/rosqual/roswire, 2022.

[24] Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. A study on challenges of testing robotic systems. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*, pages 96–107. IEEE, 2020. doi: 10.1109/ICST46399.2020.00020. URL https://doi.org/10.1109/ICST46399.2020.00020.

[25] Gopika Ajaykumar, Maureen Steele, and Chien-Ming Huang. A survey on end-user robot programming. *ACM Comput. Surv.*, 54(8):164:1–164:36, 2022. doi: 10.1145/3466819. URL https://doi.org/10.1145/3466819.

[26] Mark Allan, Uland Wong, Padraig M. Furglong, Arno Rogg, Scott McMichael, Terry Welsh, Ian Chen, Steven Peters, Brian Gerkey, Morgan Quigley, Mark Shirley, Mathew Deans, Howard Cannon, and Terry Fong. Planetary rover simulation for lunar exploration missions. IEEE Aerospace Conference, 2019. URL https://ntrs.nasa.gov/api/citations/20190027571/downloads/20190027571.pdf.

[27] Andrzej Wąsowski Anders Fischer-Nielsen, Zhoulai Fu. The forgotten case of the dependency bugs : On the example of the robot operating system. https://ieeexplore.ieee.org/abstract/document/9276573/, 2020.

[28] Renaud Anjoran. The 3 most common challenges with manufacturing robots. https://www.cmc-consultants.com/blog/the-3-most-common-challenges-with-manufacturing-robots, 2021.

[29] Vytautas Astrauskas, Aurel Bílý, Jonás Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. The prusti project: Formal verification for rust. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, volume 13260 of *Lecture Notes in Computer Science*, pages 88–108. Springer, 2022. doi: 10.1007/978-3-031-06773-0_5. URL https://doi.org/10.1007/978-3-031-06773-0_5.

[30] Paulo Canelas, Miguel Tavares, Ricardo Cordeiro, Alcides Fonseca, and Christopher S. Timperley. An experience report on challenges in learning the robot operating system. In *4th International Workshop on Robotics Software Engineering (RoSE'22), May 9, 2022, Pittsburgh, PA, USA. ACM, New York, NY, USA*. IEEE, 2022. URL https://rose-workshops.github.io/files/rose2022/papers/RoSE22_paper_11.pdf.

[31] Fulano, Cicrano, and Beltrano. A paper on something. In *The 7th Conference on Things and Stuff (CTS 2009)*, Lisbon, Portugal, May 2009. Accepted for publication.

[32] Catarina Gamboa, Paulo Alexandre Santos, Christopher Steven Timperley, and Alcides Fonseca. User-driven design and evaluation of liquid types in java. *CoRR*, abs/2110.05444, 2021. URL https://arxiv.org/abs/2110.05444.

[33] Catarina Gamboa, Paulo Alexandre Santos, Christopher Steven Timperley, and Alcides Fonseca. Liquidjava: Adding lightweight verification to java. 2021.

[34] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is still so hard. *IEEE Softw.*, 26(4):66–69, 2009. doi: 10.1109/MS.2009.86. URL https://doi.org/10.1109/MS.2009.86.

[35] Pablo Garrido, Antonio Cuadros, and Maria Merlan. micro-ros for renesas e2 studio. https://github.com/micro-ROS/micro_ros_renesas2estudio_component, 2021.

[36] Ajay Harish. When nasa lost a spacecraft due to a metric math mistake. https://www.simscale.com/blog/2017/12/nasa-mars-climate-orbiter-metric/, 2021.

[37] Gabriel Hartmann, Zvi Shiller, and Amos Azaria. Autonomous head-to-head racing in the indy autonomous challenge simulation race. *CoRR*, abs/2109.05455, 2021. URL https://arxiv.org/abs/2109.05455.

[38] Ranjit Jhala and Niki Vazou. Refinement types: A tutorial. *Found. Trends Program. Lang.*, 6(3-4):159–317, 2021. doi: 10.1561/2500000032. URL `https://doi.org/10.1561/2500000032`.

[39] Sayali Kate, John-Paul Ore, Xiangyu Zhang, Sebastian G. Elbaum, and Zhaogui Xu. Phys: probabilistic physical unit assignment and inconsistency detection. In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 563–573. ACM, 2018. doi: 10.1145/3236024.3236035. URL `https://doi.org/10.1145/3236024.3236035`.

[40] Sayali Kate, Michael Chinn, Hongjun Choi, Xiangyu Zhang, and Sebastian G. Elbaum. PHYSFRAME: type checking physical frames of reference for robotic systems. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 45–56. ACM, 2021. doi: 10.1145/3468264.3468608. URL `https://doi.org/10.1145/3468264.3468608`.

[41] 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 *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*, pages 430–440. IEEE, 2020. doi: 10.1109/ICSME46990.2020.00048. URL `https://doi.org/10.1109/ICSME46990.2020.00048`.

[42] Hadas Kress-Gazit, Morteza Lahijanian, and Vasumathi Raman. Synthesis for robots: Guarantees and feedback for robot behavior. *Annual Review of Control, Robotics, and Autonomous Systems*, 2018.

[43] Nico Lehmann, Adam Geller, Gilles Barthe, Niki Vazou, and Ranjit Jhala. Flux: Liquid types for rust. *CoRR*, abs/2207.04034, 2022. doi: 10.48550/arXiv.2207.04034. URL `https://doi.org/10.48550/arXiv.2207.04034`.

[44] Srinivas Nedunuri, Sailesh Prabhu, Mark Moll, Swarat Chaudhuri, and Lydia E. Kavraki. Smt-based synthesis of integrated task and motion plans from plan outlines. In *2014 IEEE International Conference on Robotics and Automation, ICRA 2014, Hong Kong, China, May 31 - June 7, 2014*, pages 655–662. IEEE,

2014. doi: 10.1109/ICRA.2014.6906924. URL https://doi.org/10.1109/ICRA.2014.6906924.

[45] Tiago Neto, Rafael Arrais, Armando Sousa, André Santos, and Germano Veiga. Applying software static analysis to ROS: the case study of the FASTEN european project. In Manuel F. Silva, José Luís Lima, Luís Paulo Reis, Alberto Sanfeliu, and Danilo Tardioli, editors, *Robot 2019: Fourth Iberian Robotics Conference - Advances in Robotics, Volume 1, Porto, Portugal, 20-22 November, 2019*, volume 1092 of *Advances in Intelligent Systems and Computing*, pages 632–644. Springer, 2019. doi: 10.1007/978-3-030-35990-4_51. URL https://doi.org/10.1007/978-3-030-35990-4_51.

[46] John-Paul Ore, Carrick Detweiler, and Sebastian G. Elbaum. Phriky-units: a lightweight, annotation-free physical unit inconsistency detection tool. In Tevfik Bultan and Koushik Sen, editors, *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 352–355. ACM, 2017. doi: 10.1145/3092703.3098219. URL https://doi.org/10.1145/3092703.3098219.

[47] John-Paul Ore, Carrick Detweiler, and Sebastian G. Elbaum. An empirical study on type annotations: Accuracy, speed, and suggestion effectiveness. *ACM Trans. Softw. Eng. Methodol.*, 30(2):20:1–20:29, 2021. doi: 10.1145/3439775. URL https://doi.org/10.1145/3439775.

[48] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. *ICRA workshop on open source software*, 3(4.2):5, 2009. doi: 10.1080/09511920601160262. URL http://robotics.stanford.edu/~ang/papers/icraoss09-ROS.pdf.

[49] Hirohisa Sakai and Kakuro Amasaka. Development of a robot control method for curved seal extrusion for high productivity in an advanced toyota production system. *Int. J. Comput. Integr. Manuf.*, 20(5):486–496, 2007. doi: 10.1080/09511920601160262. URL https://doi.org/10.1080/09511920601160262.

[50] André Santos, Alcino Cunha, and Nuno Macedo. The high-assurance ROS framework. In *3rd IEEE/ACM International Workshop on Robotics Software Engineering, RoSE@ICSE 2021, Madrid, Spain, June 2, 2021*, pages 37–40. IEEE, 2021. doi: 10.1109/RoSE52553.2021.00013. URL https://doi.org/10.1109/RoSE52553.2021.00013.

[51] Nishant Sharma, Sebastian G. Elbaum, and Carrick Detweiler. Rate impact analysis in robotic systems. In *2017 IEEE International Conference on Robotics and Automation, ICRA 2017, Singapore, Singapore, May 29 - June 3, 2017*, pages 2089–2096. IEEE, 2017. doi: 10.1109/ICRA.2017.7989240. URL https://doi.org/10.1109/ICRA.2017.7989240.

[52] Sumukh Shivakumar, Hazem Torfah, Ankush Desai, and Sanjit A. Seshia. SOTER on ROS: A run-time assurance framework on the robot operating system. In Jyotirmoy Deshmukh and Dejan Nickovic, editors, *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings*, volume 12399 of *Lecture Notes in Computer Science*, pages 184–194. Springer, 2020. doi: 10.1007/978-3-030-60508-7_10. URL https://doi.org/10.1007/978-3-030-60508-7_10.

[53] Niaz Sharif Shourov, Masnur Rahman, Mohammad Zahirul Islam, Ali Ahsan, Syed Md Kamruzzaman, Saifur Rahman, Md Sakiluzzaman, Intisar Hasnain, Ekhwan Islam, Saiful Islam, and Md. Khalilur Rhaman. BRACU mongol tori: Next generation mars exploration rover. *CoRR*, abs/2111.02004, 2021. URL https://arxiv.org/abs/2111.02004.

[54] Mark Tjersland. Statick. https://github.com/sscpac/statick, 2021.

[55] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. Liquidhaskell: experience with refinement types in the real world. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 39–51. ACM, 2014. doi: 10.1145/2633357.2633366. URL https://doi.org/10.1145/2633357.2633366.