**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

Ricardo Ribeiro Pereira

# An HAROS Extension for Variability Aware ROS Code Analysis

December 2021

**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

Ricardo Ribeiro Pereira

# An HAROS Extension for Variability Aware ROS Code Analysis

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
**Alcino Cunha**

December 2021

# ACKNOWLEDGEMENTS

I would like to thank Professor Alcino Cunha for guiding me in such an important journey, for transmitting all the necessary knowledge to help solve the challenges that appeared along the way, and for being consistently available to help me to carry on with this thesis.

To Professor Nuno Macedo and also to André Santos for being such a regular and valuable help and for sharing their expertise on the discussed matters.

I would also like to express my gratitude to Professor José Nuno Oliveira, Professor José Creissac Campos and again, Professor Alcino Cunha for being such important references in formal methods field.

And finally to my family, my grandparents, my father and mother, my siblings and last but not least, to my girlfriend, for always being there.

This thesis was developed in the context of the SAFER project, financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project PTDC/CCI-INF/29583/2017 (POCI-01-0145-FEDER-029583).

# STATEMENT OF INTEGRITY

# ABSTRACT

Human kind has proven how challenging and volatile the technological market can be, growing at an exponential rate. The benefits of such evolution are directly reflected in many ways in our everyday life. Robots are a clear example of an advanced technology that may be completely integrated in our societies in a near future, hopefully in such a way that their actions will be considered as trustable as human actions are. These machines are permanently relying on software, which has a development process that many times cannot be considered trustworthy. This may cause the final product to have multiple malfunctions, which in turn may result in tremendous economic losses or even harm human lives.

Bearing this in mind, software industry and academia have been trying to establish new standards and techniques that considerably lower the occurrence of the latter problems. The solution is to apply certain formal methodologies and tools when developing software, namely when developing critical software that controls machinery used, for example, in healthcare sector, aeronautical industry, or in military operations.

The present dissertation aims to explore and improve techniques and tools to help developers in the process of building robotic systems, namely those developed with the *Robot Operating System* (ROS). The focus will be on a specific framework named HAROS, which performs different types of analyses of ROS-based code. Although it has a solid set of useful features, some need to be upgraded to enhance efficiency and also to promote a better experience to their users, in particular when the the software has many variants, as is often the case with robotic applications.

The proposed extension offers ROS and HAROS users a practical methodology that, by merging existing ROS and *Software Product Line* (SPL) development tools and concepts, considerably improves the understanding of the variability in a robotic application, without requiring a steep learning curve.

KEYWORDS Formal Methods, High Assurance ROS (HAROS), JavaScript, NodeJS, Robot Operating System (ROS), Software Product Line (SPL), Text-based Variability Language (TVL), Variability.

# RESUMO

A humanidade tem provado o quão desafiador e volátil consegue ser o mercado tecnológico, que cresce a um ritmo exponencial. Os benefícios dessa evolução refletem-se diretamente de várias formas no nosso quotidiano. Os robôs são um exemplo evidente de uma tecnologia vanguardista que num futuro próximo poderá estar totalmente integrada nas nossas sociedades, de tal forma que as suas ações serão consideradas tão confiáveis quanto as dos seres humanos. Não obstante, estas máquinas estão constantemente dependentes de software cujo processo de desenvolvimento é muitas vezes pouco credível. Isto leva a que o produto final tenha muitas anomalias, que por sua vez podem resultar em prejuízos económicos avultados ou até pôr em perigo vidas humanas.

Tendo isso em conta, a indústria de software e a academia têm tentado estabelecer novos padrões e técnicas que reduzem consideravelmente a ocorrência de problemas futuros. A solução passa por aplicar determinadas metodologias e ferramentas formais durante o desenvolvimento de software, nomeadamente no desenvolvimento de software sensível que controla aparelhos usados, por exemplo, no setor da saúde, na indústria aeronáutica, ou até em operações militares.

O propósito desta dissertação é explorar e melhorar técnicas e ferramentas que auxiliem os técnicos no processo de construção de sistemas robóticos, nomeadamente os desenvolvidos com o *Robot Operating System* (ROS). O foco vai para uma ferramenta chamada HAROS, que exerce diferentes tipos de análises em código ROS. Apesar da mesma já ter um conjunto consistente de funcionalidades, algumas precisam de ser otimizadas para melhorar a eficiência e também para promover uma melhor experiência aos seus utilizadores, em particular quando o software tem muitas variantes, como é frequentemente o caso nas aplicações robóticas.

A extensão desenvolvida oferece aos utilizadores do ROS e do HAROS uma metodologia prática que, combinando ferramentas e conceitos já usados no ROS e no desenvolvimento de *Linhas de Produtos de Software* (LPSs), melhora consideravelmente a compreensão da variabilidade existente numa aplicação robótica, não requerendo uma curva de aprendizagem muito elevada.

PALAVRAS-CHAVE High Assurance ROS (HAROS), JavaScript, Linha de Produtos de Software (LPS), NodeJS, Métodos Formais, Robot Operating System (ROS), Text-based Variability Language (TVL), Variabilidade.

# CONTENTS

Contents . . . iv
1 Introduction . . . 3
2 The Robot Operating System . . . 5
2.1 ROS Concepts . . . 6
2.1.1 Filesystem level . . . 6
2.1.2 Computation graph level . . . 7
2.2 First configuration: a keyboard controlled turtle . . . 10
2.2.1 Launch file . . . 11
2.2.2 Analysis . . . 12
2.3 Second configuration: a randomly controlled turtle . . . 13
2.3.1 Launch file . . . 15
2.3.2 Analysis . . . 17
2.4 Third configuration: adding a multiplexer . . . 17
2.4.1 Launch file . . . 20
2.4.2 Analysis . . . 22
3 The HAROS Framework . . . 23
3.1 Setup stage . . . 24
3.1.1 Commands . . . 24
3.1.2 Project file . . . 25
3.2 Analysis stage . . . 27
3.2.1 Model extraction . . . 27
3.3 Report stage . . . 29
3.3.1 Dashboard . . . 29
3.3.2 Packages . . . 29
3.3.3 Issues . . . 31
3.3.4 Models . . . 32
3.4 Shortcomings . . . 33
4 Managing Software Variability . . . 35
4.1 Sources and challenges of variability in robotics . . . 35
4.1.1 Sources . . . 35
4.1.2 Challenges . . . 37
4.2 Software product lines . . . 38
4.3 Feature models . . . 38
4.3.1 Feature diagrams . . . 40

4.3.2 Feature diagram example . . . 41
4.4 TVL: a text-based variability language . . . 42
4.4.1 Features . . . 43
4.4.2 Attributes . . . 44
4.4.3 Other keywords and operators . . . 45
4.4.4 Constraints . . . 46
4.4.5 Modularisation . . . 46
4.4.6 TVL feature model example . . . 46
4.4.7 The TVL reference implementation . . . 48
5 Handling Variability in ROS and HAROS . . . 52
5.1 Handling variability in ROS . . . 53
5.1.1 Describing the variability with a TVL file . . . 53
5.1.2 Implementing a generic Launch File . . . 54
5.1.3 Towards a variability aware ROS launcher . . . 61
5.2 Handling variability in HAROS . . . 61
5.2.1 HAROS-TVL server . . . 62
5.2.2 Project configuration file . . . 63
5.3 Summary . . . 64
6 HAROS-TVL: A Variability Aware HAROS Extension . . . 65
6.1 Executing Java code within a JavaScript environment . . . 65
6.2 Converting the TVL feature diagram into a D3 JSON structure . . . 66
6.3 Implementing the HAROS-TVL server . . . 66
6.3.1 Initialization . . . 67
6.3.2 Selecting or deselecting a feature . . . 67
6.3.3 Refreshing the server . . . 68
6.3.4 The server web page . . . 68
6.3.5 Generating configurations . . . 69
6.4 Implementing the HAROS-TVL extension . . . 70
6.4.1 Computation graph file . . . 71
6.4.2 Runtime issues files . . . 74
6.4.3 Variational computation graph . . . 74
7 HAROS-TVL Evaluation . . . 76
7.1 Applying HAROS-TVL to `turtlerand` . . . 76
7.1.1 Defining a TVL file . . . 76
7.1.2 Defining a generic launch file . . . 77
7.1.3 Defining a project configuration file . . . 77
7.1.4 Evaluation . . . 78
7.1.5 Conclusion . . . 80

7.2 Applying HAROS-TVL to `turtlebot` . . . . . . . . . . 81
7.2.1 Defining a TVL file . . . . . . . . . . 81
7.2.2 Defining a generic launch file . . . . . . . . . . 83
7.2.3 Defining a project configuration file . . . . . . . . . . 83
7.2.4 Evaluation . . . . . . . . . . 84
7.2.5 Conclusion . . . . . . . . . . 86
7.3 Applying HAROS-TVL to `lizi` . . . . . . . . . . 87
7.3.1 Defining a TVL file . . . . . . . . . . 87
7.3.2 Defining a generic launch file . . . . . . . . . . 88
7.3.3 Defining a project configuration file . . . . . . . . . . 88
7.3.4 Evaluation . . . . . . . . . . 89
7.3.5 Conclusion . . . . . . . . . . 91
8 Conclusion . . . . . . . . . . 92
8.1 Future work . . . . . . . . . . 93

A Listings . . . . . . . . . . 96

# LIST OF FIGURES

1 Master name service example. . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2 turtleX moved using turtle_teleop_key node. . . . . . . . . . . . . . . . . . 11
3 First configuration computation graph. . . . . . . . . . . . . . . . . . . . . . 13
4 turtleX moving randomly through the virtual environment. . . . . . . . . . . . 16
5 Second configuration computation graph. . . . . . . . . . . . . . . . . . . . . 17
6 turtleX moving both randomly and straightly through the virtual environment. 22
7 Third configuration computation graph. . . . . . . . . . . . . . . . . . . . . . 22
8 Dashboard tab. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
9 Packages tab. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
10 Issues tab. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
11 Computation graphs of the three example configurations. . . . . . . . . . . . 32
12 HAROS pointing random_controller node as an answer to the query. . . . . . 33
13 Conditioned element (dashed) in the computation graph. . . . . . . . . . . . . 33
14 turtlerand feature model. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
15 Example of a feature model. . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
16 Example of a feature model configuration. . . . . . . . . . . . . . . . . . . . 42
17 Mock of an interactive TVL feature diagram. . . . . . . . . . . . . . . . . . . 61
18 Proposed interface for the new HAROS visualiser with an interactive TVL feature diagram. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
19 Evolution of a feature model until a configuration is selected. . . . . . . . . . 69
20 Different appearances of the computation graph according to the respective different states of the Figure 19 feature diagram. . . . . . . . . . . . . . . . . . 74
21 Initial state of Models tab for the turtlerand application. . . . . . . . . . . . 78
22 Initial state of Issues tab for the turtlerand application. . . . . . . . . . . . . 79
23 Resulting state of Models tab after selecting Turtlerand feature (in Figure 21). . 79
24 Resulting state of Models tab after deselecting Random and Teleop features (in Figure 23). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
25 Resulting state of Issues tab after deselecting Random and Teleop features (in Figure 23). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
26 Resulting state of Models tab after selecting Random and Teleop features (in Figure 23). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
27 Initial state of Models tab for turtlebot application. . . . . . . . . . . . . . . 84
28 Resulting state of Models tab after selecting Battery feature (in Figure 27). . . . 85

29 Resulting state of `Models` tab after selecting four features (in Figure 28). . . . . . . 85
30 Resulting state of `Models` tab after selecting `Logitech` and `Keyboard` features (in Figure 29). . . . . . . 86
31 Initial state of `Models` tab for `lizi` application. . . . . . . 89
32 Resulting state of `Models` tab after selecting `Amcl` feature and deselecting others (in Figure 31). . . . . . . 89
33 Resulting state of `Models` tab after collapsing `Navigation` and expanding `Hw` features (in Figure 32). . . . . . . 90
34 Resulting state of `Models` tabs after deselecting `Depth_cam`, `Diagnos` and `Cam` features (in Figure 33). . . . . . . 90

# LIST OF LISTINGS

2.1 First configuration launch file. . . . . 11
2.2 The list of topics (and respective types) to publish to and subscribe to. . . . . 12
2.3 Message sent from turtle_teleop_key to turtlesim_node over cmd_vel topic after pressing an up arrow key. . . . . 12
2.4 Content of random_controller.cpp. . . . . 13
2.5 Second configuration launch file. . . . . 15
2.6 Second configuration alternative launch file. . . . . 16
2.7 Content of multiplexer.cpp. . . . . 17
2.8 Third configuration launch file. . . . . 20
2.9 Third configuration parameterized launch file. . . . . 21
3.1 turtlerand project file. . . . . 25
4.1 turtlerand TVL feature model. . . . . 44
4.2 A variant of Listing 4.1 turtlerand TVL feature model. . . . . 44
4.3 Example of a TVL feature model. . . . . 47
5.1 turtlerand generic launch file. . . . . 54
5.2 turtlerand generic launch file simplified (in relation to Listing 5.1). . . . . 56
5.3 turtlerand alternative TVL feature model. . . . . 58
5.4 turtlerand alternative generic launch file. . . . . 58
5.5 turtlerand generic launch file with <group> tag technique. . . . . 60
5.6 Generic turtlerand project configuration file. . . . . 63
5.7 HAROS-TVL turtlerand project configuration file. . . . . 64
6.1 turtlerand launch file corresponding to the configuration selected in Figure 19. 70
6.2 Snippet of configurations.json file. . . . . 71
6.3 configurations.json file (Listing 6.2) after selecting Random and deselecting Teleop features. . . . . 73
7.1 turtlerand project configuration file. . . . . 77
7.2 turtlebot TVL feature model. . . . . 82
7.3 turtlebot project configuration file. . . . . 83
7.4 lizi TVL feature model. . . . . 87
7.5 lizi project configuration file. . . . . 88
A.1 turtlebot generic launch file. . . . . 96
A.2 lizi generic launch file. . . . . 102

# 1 INTRODUCTION

Every year, significant progress happens in the technological world. Nowadays, there are machine learning algorithms running in our smartphones, cloud computation power just a click away, autonomous vehicles that communicate with each other, and many other technologies that were unthinkable some years ago. Robotic systems (of which autonomous vehicles are an example) are a key part of the current technological boom, but despite all their advantages, there are some obstacles preventing their widespread use, in particular related to safety when operating in critical contexts. In fact, several tragic accidents have already occurred with autonomous vehicles. The key question is then how to make safe and reliable robotic systems? It is not an easy task, as building a robot requires many different interdisciplinary techniques and tools.

Software is a key ingredient of modern robots, and many of the tools and techniques that are used in traditional software development to ensure quality can also be applied in this context. Among those techniques there are, for example, formal methods, mainly used in the early stages of the development process to ensure the quality of the design, and testing or static analysis to ensure the quality of the final implementation. The *Robot Operating System* (ROS) [10] is a framework that provides a set of tools and libraries that all together allow a developer to easily build a robotic system. A ROS application comprises a distributed group of nodes (processes), each one with a certain function, that communicate via message passing in a publish-subscribe paradigm. The large community that exists around it, a result of being an open source tool, and its flexibility, turned ROS into one of the most popular frameworks for developing robotic systems nowadays.

HAROS[1] [12] is a framework that promotes the quality of a ROS application, mainly by applying static analysis techniques to its source code. Among other capabilities, HAROS allows its users to extract ROS computation graphs (an abstract model of the system architecture) directly from source code [13], a powerful feature that opens doors to perform several advanced analyses, for example using model checking [2].

Although there is a lot of work already done in HAROS, it still has certain aspects that can be improved. First, it is important to understand that many different configurations may

1 https://github.com/git-afsantos/haros

exist for the same robot, each supporting a different set of features (for example, different hardware components) [5]. This means that for a single robot we may end up developing not only one software application, but a full *Software Product Line* (SPL) [9]. In ROS these different configurations are typically managed in an ad-hoc manner, by resorting to different (or parameterized) launch files to set up different features, without an explicit variability model (e.g. a feature model) that describes the configuration space of a robot (namely which features are compatible or dependent on each other). In ROS, launch files are used to define which nodes and topics exist in an application. So far, HAROS can only handle one configuration at a time, defined by a specific set of launch files. To analyse the full SPL it is necessary to analyze one configuration at a time, which in many cases is unfeasible due to the high number of possible configurations. Several work already exists on variability-aware static analysis, where the goal is to somehow analyze all variants at once, thus increasing the efficiency of the process [14]. The overall goal of this thesis is to add explicit variability support to HAROS, to enable such variability aware analyses in the future. This will require searching/defining a new configuration language that captures in a systematic way the variability of a ROS application, and a new variability aware computation graph visualisation and exploration technique.

The first four chapters of this thesis describe the state of the art of the present research. After this brief introduction, Chapter 2 summarizes the most important and useful ROS concepts and describes how can an application be built with this framework. Chapter 3 presents HAROS, giving the reader an overview of what this framework consists of by using it to analyze the ROS application built in the previous chapter. Then, Chapter 4 uncovers some of the existing techniques and tools that can be applied to manage variability.

The two following chapters focus on the contribution of the thesis. Chapter 5 describes a proposal to handle variability in HAROS, and ROS in general. Chapter 6 describes HAROS-TVL, a new variability aware HAROS extension. Lastly, the final two chapters evaluate HAROS-TVL, pointing the pros and cons, as well as what can be improved. In particular, Chapter 7 evaluates HAROS-TVL with three case studies, and Chapter 8 concludes the thesis by depicting a general overview of what has been done and pointing out some ideas for future work.

# 2 THE ROBOT OPERATING SYSTEM

As stated before, the world is changing and evolving at an astounding pace and new technologies are largely responsible for this progression. Robots are also part of this technological progress and they are increasingly operating closely to humans. They can be used in healthcare, industry, or in military operations, and in this kind of critical domains it is essential to ensure their safety. Building and programming a robot requires a different approach than that commonly used by developers. To mitigate this problem some researchers started building frameworks that help developing robotic software.

The *Robot Operating System* (ROS) is precisely a framework that assembles libraries and other existing tools, that all together simplify the building of robotic systems. To develop software with ROS, one must understand some fundamental concepts that are key parts of this framework, such as *nodes*, *topics*, *messages*, *launch files*, and others. This framework has another important aspect that must be understood - its runtime architecture - a set of distributed nodes that communicate with each other with a publish-subscribe paradigm using message topics.

In order to introduce the main features of ROS to its new users, there are some tutorials online[1] that teach how to develop applications with some available robot simulators. These simulators are able to represent part of the behaviour of a real robot in a very simple and virtual environment. After some experiments following these tutorials and interacting with some simulators, it was decided to illustrate the central concepts of ROS by building three different configurations of an application with the `turtlesim` simulator[2], namely:

- A simple configuration with a simulated turtle controlled manually with the keyboard;
- Another simple configuration with simulated turtle controlled automatically with a random controller.
- A more complex configuration where both controllers can be used to move the simulated turtle.

1 http://wiki.ros.org/turtlesim/Tutorials
2 http://wiki.ros.org/turtlesim

Some standard ROS tools are also executed in each of the three configurations, to explain their usage and to obtain important information that will be useful later. This chapter documents how ROS works and despite not being the central point of this research, it highlights essential elements that will be necessary to fully understand the following chapters. As side note, the ROS version used in this chapter was *Noetic Ninjemys*.

## 2.1 ROS CONCEPTS

ROS concepts are divided into three main levels[3]: the *Filesystem Level*, the *Computation Graph Level*, and the *Community Level*. However, the latter is not relevant for this thesis, as well as a few concepts from the other levels, and for this reason they will be omitted.

### 2.1.1 *Filesystem level*

A filesystem keeps everything manageable by establishing proper locations to store files. Usually, they are grouped by their type or function in folders and sub folders inside the system, making it easier to access any desired file. Some ROS filesystem elements are described below.

*Catkin workspace*

Starting from the top, there is a *workspace* folder (`catkin_ws`) where users must create their packages under a *build system* named *catkin*[4] (the successor of *rosbuild*, the previous build system used in earlier versions of ROS). This folder can contain up to three or more subdirectories of which `/build`, `/devel` and `/src` are part of. A build system automates the manipulation of all object files needed for a program to execute, which eases the compilation process when there are many source code files. Once `catkin_make` command is invoked, the build system reads a file containing a set of directives to generate object files.

*Package*

Inside `catkin_ws/src` are all the packages a user created in the workspace. Everything that is related with a certain package must be inside the respective folder, which is grouped by the type of files it has (ROS runtime processes, ROS-dependent libraries, launch files, etc.). In addition, the package has a *manifest* and a `CMakeList.txt` file.

3 `http://wiki.ros.org/ROS/Concepts`
4 `http://wiki.ros.org/catkin`

*Package manifest*

The manifest file (`package.xml`) has essential information about the package it is related to, like its name, description, maintainers information, license, dependencies, etc.

*CMakeList*

`CMakeList.txt` is the file mentioned before that contains references to source and object files. Every time a file that instantiates a node is created, it must be declared in this file along with its location and the corresponding name for the object file. The same condition is valid for ROS runtime services and messages.

### 2.1.2 *Computation graph level*

ROS has a particular runtime architecture where processes communicate with each other through a publish-subscribe paradigm. Every process executes independently from the others, forming a distributed architecture with a *peer-to-peer* configuration. In essence, the ROS computation graph represents a general view of what is happening during runtime execution and it is possible to depict it using appropriate tools that will be explored in sections ahead. Understanding the computation graph is crucial to the following stages of this research.

*Nodes*

Typically, a robot comprises a set of hardware elements that are programmed to execute specific actions and these elements are controlled by nodes. For example, proximity sensors, electronic arms, cameras and wheel motors may be managed by nodes. Other purely computational nodes may exist, for example to serve as multiplexers, to collect and save important and specific data, to handle GPS coordinates, etc. The reason why nodes are so flexible is because they are built with ROS *client libraries* coded in some of the programming languages most applied in software development (namely Python and C++).

Nodes are independent processes and the availability of one should not directly interfere with others, which means that a process may be killed for some reason, but the others continue to work. Each node has a well defined purpose in the whole system, making it easier to isolate the source code of each functionality and therefore to reduce code complexity [10], a major problem in software development. This aspect also allows developers to easily reuse code from other projects.

### *Topics*

The reason why the ROS computation graph has a peer-to-peer configuration, lies on the fact that sensors and identical hardware devices generate a lot of data that is constantly sent to the network and having a central entity processing all this data would cause its overflow. So, communication between nodes in this peer-to-peer network is done using topics.

A topic is literally a channel that nodes can subscribe and where they can also publish on. For example, a node representing a sensor will publish messages on a certain topic and the node responsible for dealing with such data must subscribe the same topic. This approach avoids network flooding. However, nothing prevents a topic from congestion and that is why a ROS developer has to specify the size of the queue for incoming or outgoing messages.

In order for a topic to exist, either a publisher or a subscriber (at least) must have been declared. It is also important to say that topics are necessarily related to the type of messages they carry, which in general, defines their name and promotes code reusability.

### *Master*

The ROS master is a collection of nodes and programs[5] that set up the system to run the computation graph. In order to run nodes and have them communicating with each other, a ROS user must launch the ROS master before using the command `roscore`. Figure 1 shows how the ROS master is used to establish communication between two nodes.

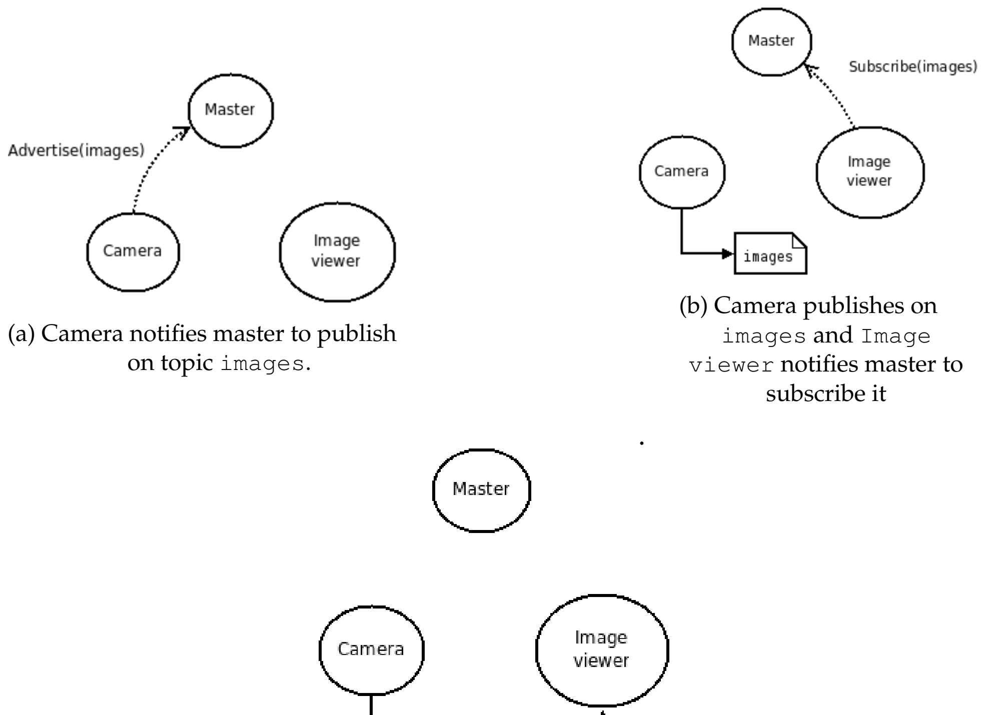
Although it must be always running, the ROS master does not play a primary role in computation graphs and it is usually ignored, that is, it does not appear in the visual representation of computation graphs.

Another important feature of the ROS master is the capability of storing certain *parameters* that may be useful to more than one ROS resource in a parameter server. The same happens with services that may be accessed by nodes too. Overall, the ROS master is in charge of establishing nodes communication and providing and making other components accessible to all computation graph actors.

### *Parameter server*

Parameters are often used when there is a certain value that is needed by several nodes. The parameter server acts as a shared dictionary with a key-value pair structure, where values can be *integers*, *booleans*, *strings*, or even another dictionary. Its main function is to store and retrieve these values at runtime, but it is recommended to use them statically (at the beginning of the execution), since multiple and consecutive accesses to the parameter server during runtime lead to losses in efficiency.[6]

5 `http://wiki.ros.org/roscore`
6 `http://wiki.ros.org/Parameter%20Server`



(a) Camera notifies master to publish on topic `images`.

(b) Camera publishes on `images` and `Image viewer` notifies master to subscribe it

(c) Master establishes a topic between `Camera` and `Image viewer`.

Figure 1: Master name service example.
(Source: `http://wiki.ros.org/Master`)

*Configurations*

The ROS computation graph varies according to the nodes that are running. A package may have ten nodes but a ROS user may want for a particular application to run only five of them. Now, let us imagine that for another application these five nodes are still running, but some of them are publishing/subscribing on/to different topics than the ones they did before. These would be examples of two different configurations where some of the nodes and topics of the computation graph change.

*Launch files*

To run a node it is necessary to invoke the `rosrun` command followed by its package and the corresponding executable. However, doing this repeatedly ends up being a waste of time or even unmanageable when there is a large number of nodes. Launch files solve this issue by allowing ROS users to run multiple nodes at once.

The content of a launch file gathers instructions on how to execute every element of a certain computation graph configuration (including parameters), following an XML syntax. Thus, launching a configuration with a large number of nodes becomes easy, simply by invoking `roslaunch` command followed by the name of the package that contains the launch file and by the actual launch file name.

Running more than one instance of a node in a single computation graph happens quite often, not to stay continuously, but there are some problems that can arise in such situations, namely collisions between the names of topics they publish or subscribe. To prevent that, it is possible in a launch file to *remap* topics, which consists in changing their original name. Another option is to define different namespaces for nodes that have this issue.

Despite the fact that launch files are a set of instructions to launch ROS resources when executed with `roslaunch` command, it does not mean that all of these elements need to be launched together. In another words, supposing that two different nodes are declared in an launch file, it is possible to choose which of these nodes would be executed when running `roslaunch`. That trick would be possible by declaring `arg` (abbreviation of argument) and `group` tags, alongside `value`, `name` and `if` attributes. An example of such situation will be presented in Section 2.4.1.

*Messages*

Before sending data through a topic, the type of message that is used for this purpose must be previously defined and stored in `.msg` files. Doing so, eases the generation of source code for the message type in the different languages used in ROS. Message types may vary from simple standard types to some more complex data structures. A few message types are already pre-defined in ROS and available to be used.

## 2.2 FIRST CONFIGURATION: A KEYBOARD CONTROLLED TURTLE

At the beginning, working in ROS can be quite confusing, so the ROS community tries to smooth this initial impact by providing tools like `turtlesim`, a package to interact with a simulated mobile turtle-shaped robot named `turtleX`. These tools allow beginners to experience ROS without having to deal with the complex problems arising when programming real robots they are not familiar with.

Using `turtlesim`, it is possible to build some simple configurations with pre-built nodes. For instance, `turtlesim_node` is a node representing a simulator capable of showing the behaviour of `turtleX` when it receives `Twist` messages on the subscribed topic `cmd_vel`. `Twist` messages express linear and angular velocities (in this order) using a three dimensional vector for each component. Consequently, these vectors (orientation pointers) are represented by the message type `Vector3`, which includes three `float64` values, the `x`, `y`, and `z` compo-

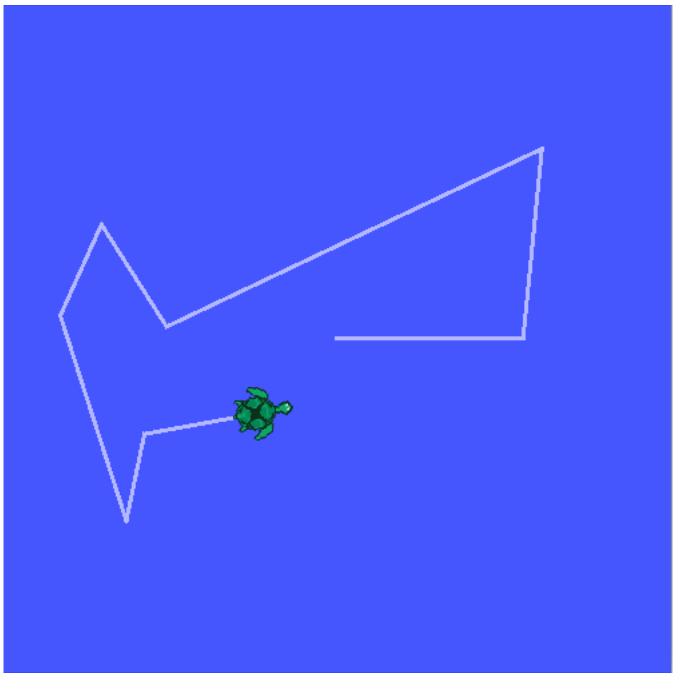


Figure 2: turtleX moved using turtle_teleop_key node.

nents. It also publishes its current position (using Pose messages) on a topic, named pose, that other nodes may subscribe. turtle_teleop_key is another turtlesim node which reads keyboard arrow keys typed by the user, translating them to Twist messages published on the same topic turtlesim_node is subscribing (cmd_vel). Running roscore and both nodes with rosrun in different terminals, will start a new window with a centered image of a turtle, which starts rotating left or right and/or moving forward or backward as the user types the arrow keys in the same terminal which is executing turtle_teleop_key node. A result of this interaction with turtleX can be seen in Figure 2.

### 2.2.1 *Launch file*

As said earlier in this chapter, running nodes one by one ends up being a burden to the user and launch files are an effective solution for this problem. Therefore, the next step is to build a launch file capable of launching turtlesim_node and turtle_teleop_key with a single bash command. This file, with the .launch extension, has an XML syntax. Its outer tag is launch, which embraces two other node tags, each one with the attributes pkg, type and name. These two tags refer to the previous nodes and they specify the package where each node belongs, its object file, and its name in the computation graph, respectively. This simple launch file is shown in Listing **??**. To launch it, the user only has to run roslaunch since it also starts roscore.

```
<launch>
    <node pkg="turtlesim" type="turtlesim_node" name="turtlesim_node"/>
    <node pkg="turtlesim" type="turtle_teleop_key" name="turtle_teleop_key"/>
</launch>
```

Listing 2.1: First configuration launch file.

### 2.2.2 *Analysis*

ROS has a lot of tools that a developer can use to examine what is happening, especially in the background. For instance, it is possible to observe which topics are available (Listing 2.2), to listen to a certain topic (Listing 2.3), and to find out on which topics a node is publishing or which it is subscribing.

```
$ rostopic list
Published topics:
 * /turtle1/color_sensor [turtlesim/Color] 1 publisher
 * /turtle1/cmd_vel [geometry_msgs/Twist] 1 publisher
 * /rosout [rosgraph_msgs/Log] 2 publishers
 * /turtle1/pose [turtlesim/Pose] 1 publisher

Subscribed topics:
 * /turtle1/cmd_vel [geometry_msgs/Twist] 1 subscriber
 * /rosout [rosgraph_msgs/Log] 1 subscriber
```

Listing 2.2: The list of topics (and respective types) to publish to and subscribe to.

```
$ rostopic echo /turtle1/cmd_vel
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

Listing 2.3: Message sent from `turtle_teleop_key` to `turtlesim_node` over `cmd_vel` topic after pressing an up arrow key.

Another important tool is `rqt_graph` which allows users to visualise a dynamic representation of the current computation graph as shown in Figure 3, containing the turtle_teleop_key node, the turtlesim_node, and the topic cmd_vel which both use to send and receive Twist messages, respectively. This tool can be launched by executing the command

```
rosrun rqt_graph rqt_graph
```

and it can also be added to the launch file.

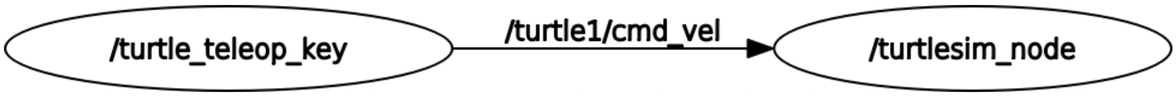


Figure 3: First configuration computation graph.

## 2.3 SECOND CONFIGURATION: A RANDOMLY CONTROLLED TURTLE

Now that the most important basics of ROS for the present research are covered, it is essential to learn how to build a node. In order to take advantage of the existing ROS libraries, the chosen programming language was C++.

The goal is to program a node, named random_controller, that sends turtleX consecutive Twist messages containing random values, making it move randomly through the virtual environment. The code is shown in Listing 2.4.

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include <random>
#include <cmath>

int main(int argc, char **argv){
    ros::init(argc, argv, "random_controller");

    ros::NodeHandle n;

    ros::Publisher publisher = n.advertise<geometry_msgs::Twist>("cmd_vel", 1000)
        ;

    geometry_msgs::Twist msg; //initializes a Twist message with all values to
        zero

    ros::Rate loop_rate(5); //loops at a 5Hz rate

    n.getParam("vel", msg.linear.x); //x component of linear vector is set to vel
         parameter value

    while(ros::ok()){ //loops as long as ros is running
19         msg.angular.z = ((double(rand()) / double(RAND_MAX)) * (M_PI - (-M_PI)))
               + (-M_PI); //z component of linear vector is set to a random value
               between (*$\pi$*) and (*$\pi$*)
20
21         publisher.publish(msg); //publishes the message
22
23         loop_rate.sleep();
24     }
25
26   return 0;
27 }
```

Listing 2.4: Content of `random_controller.cpp`.

*Code explanation*

To have access to ROS C++ functions and other libraries, it is important to include them in the `.cpp` file. Thus, `ros.h` is needed to have access to functions that allow us to create a node, to set a loop rate, to create publishers, etc. The `Twist.h` library allows to instantiate `Twist` messages. The `random` library provides random numbers and `cmath` contains a set of mathematical functions and constants.

In Line 6 a new node is initialised with name `random_controller`. A node handler is also declared in Line 8 and it will allow the current node to subscribe and publish on topics, to access the parameter server, to create timers, etc.

Next, a publisher is instantiated and committed to publish `Twist` messages on `cmd_vel` topic, establishing a queue with a maximum of a thousand outgoing messages (Line 10).

An instance of a `Twist` message is created in Line 12, with the `x`, `y`, and `z` components values being initialised with zero by default. Line 14 declares at which rate the following loop will be working. In this particular case, the loop rate was set to five hertz, meaning that in one second (approximately) five iterations will occur.

The use of parameters was mentioned before and they are useful here to set the `x` component of linear velocity via parameter `vel` (Line 16). Notice that the only modified component of linear velocity is `x`, since changing `y` would make `turtleX` to move in a diagonal direction, which looks unnatural, and changing `z` would not change where it moves, since it only lifts or lowers `turtleX`.

The stopping condition chosen for the main loop is `ros::ok()`, which will return `false` if: a `SIGINT` signal is received, another node with the same name forces the current one to abort execution, `ros::shutdown()` has been called by some element of the computation graph, or if all `ros::NodeHandles` have been destroyed.

The next step is to change the angular velocity of `msg` (Line 19), causing a `turtleX` rotation. This rotation occurs with respect to the `z` axis, since rotations over `x` and `y` axis would point turtleX belly out off the ground, which looks unnatural too. The rotation amplitude is a randomly chosen number from a range of $-\pi$ to $\pi$, rotating clockwise if negative and counter-clockwise if positive. It is important to stress that this change (of msg.angular.z value) is relative to the previous velocity state and not to the initial.

Then, the message containing the linear and the new angular velocity components is published (Line 21). The process enters in sleep mode (Line 23) for a certain period of time determined by the loop_rate specified before, and when it wakes up, these procedures - changing msg.angular.z, publishing, and sleeping - are repeated until the loop condition becomes false.

### 2.3.1 *Launch file*

Having random_controller ready to be executed, it is necessary to build a new launch file with information about the nodes that are part of this second configuration: the turtlesim_node and the random_controller. The two tags referring to both are displayed in Line 3 and Line 5 of Listing 2.5, respectively.

In this launch file it is possible to observe two new tags that were not used in the previous configuration: param and remap. The param tag creates a new entry on the parameter server with a key named vel and a double value of 0.5. This is the same parameter that is accessed by getParam function in Line 16 of Listing 2.4 and it becomes available as soon as roscore starts.

Remaps are also needed because nodes topics names matched with each other in the first configuration, but not in this one. Reading ROS wiki or executing

```
rosnode info turtlesim_node
```

command (while running turtlesim_node), one confirms that the subscribed topic is /turtle1/cmd_vel, while the topic advertised in Line 10 of Listing 2.4 is named cmd_vel. It is necessary to redefine either turtlesim_node or random_controller topic names so that they match, and the adopted solution was to rename cmd_vel topic to /turtle1/cmd_vel using a remap (Line 6 of Listing 2.5).

Another possible approach (Listing 2.6) consists of using name spaces, which will alter the relative name of any element within the group tag (elements declared with absolute names will not be changed). Thus, topics and nodes must be defined with general and relative names, since this measure promotes code reusability.

Figure 4 illustrates a random turtleX path after executing roslaunch with one of these launch files.

```
0 <launch>
1     <param name="vel" type="double" value="0.5" />
```

```

    <node pkg="turtlesim" type="turtlesim_node" name="turtlesim_node"/>

    <node pkg="turtlerand" type="random_controller" name="random_controller">

        <remap from="cmd_vel" to="/turtle1/cmd_vel"/>
    </node>
</launch>
```

Listing 2.5: Second configuration launch file.

```
<launch>
    <node pkg="turtlesim" type="turtlesim_node" name="turtlesim_node"/>

    <group ns="turtle1">
        <param name="vel" type="double" value="0.5" />

        <node pkg="turtlerand" type="random_controller" name="random_controller"
            />
    </group>
</launch>
```

Listing 2.6: Second configuration alternative launch file.

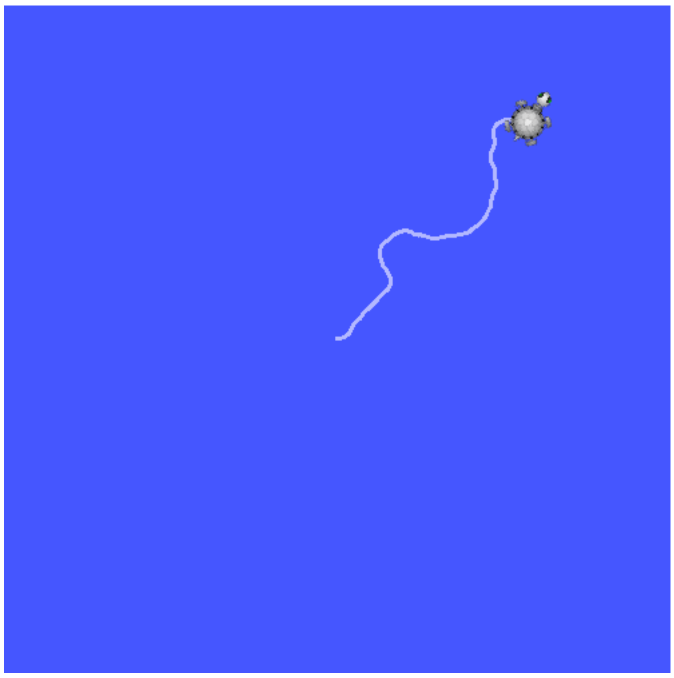


Figure 4: `turtleX` moving randomly through the virtual environment.

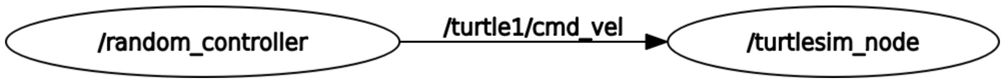


Figure 5: Second configuration computation graph.

### 2.3.2 *Analysis*

This configuration has a computation graph similar to the first one: two nodes, `random_controller` and `turtlesim_node`, publishing and subscribing on and to `/turtle1/cmd_vel` topic, respectively, as depicted in Figure 5.

## 2.4 THIRD CONFIGURATION: ADDING A MULTIPLEXER

The simplicity of the previous configurations cannot express the complexity of a ROS system in real case scenarios. To have a slightly more complex application, it was decided to build a third configuration, which comprises the functionalities of the first and the second. This new configuration also explores topic subscription and timers usage, something that has not been done yet.

This third configuration aims to combine the `turtle_teleop_key` and `random_controller` controllers, having both sending `Twist` messages to `turtlesim_node`. These nodes could easily publish on the same topic `turtlesim_node` is subscribing, although it might not be the best implementation, since the latter node would be receiving messages from two nodes with different purposes. To avoid this problem, `random_controller` messages will be held for a few seconds as soon as `turtle_teleop_key` starts communicating with `turtlesim_node`. This solution gives a higher priority to messages sent from `turtle_teleop_key` than the ones sent from `random_controller`. To distinguish these messages, it was decided that the best approach was to send them through different topics, that would be read by a new node that prioritizes them. This node would forward these messages according to the previous condition, where messages from `random_controller` would hold for at least ten seconds every time messages from `turtle_teleop_key` appear. A suitable name for this node is `multiplexer`, since it acts like one, and its source code is presented in Listing 2.7.

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"

ros::Publisher publisher;

ros::Timer timer;

```

```cpp
int SEMAPHORE = 1; //decides if a message is published (1-green) or not (0-red)

void reset(const ros::TimerEvent& event){
    SEMAPHORE = 1; //the SEMAPHORE is set to green again
}

void lowCallback(const geometry_msgs::Twist& msg){
    if (SEMAPHORE) //publishes a low priority topic's message if the SEMAPHORE is
         green
        publisher.publish(msg);
}

void highCallback(const geometry_msgs::Twist& msg){
    SEMAPHORE = 0; //disables the publication of low priority topic messages

    timer.setPeriod(ros::Duration(10));

    timer.start(); //starts a countdown to change SEMAPHORE

    publisher.publish(msg); //publishes the message
}

int main(int argc, char **argv){
    ros::init(argc, argv, "multiplexer");

    ros::NodeHandle n;

    ros::Subscriber low_subscriber, high_subscriber;

    timer = n.createTimer(ros::Duration(10), reset, true);

    publisher = n.advertise<geometry_msgs::Twist>("cmd_vel", 1000);

    low_subscriber = n.subscribe("cmd_vel_low", 1000, lowCallback); //low
        priority topic subscription

    high_subscriber = n.subscribe("cmd_vel_high", 1000, highCallback); //high
        priority topic subscription

    ros::spin();

    return 0;
}
```

Listing 2.7: Content of `multiplexer.cpp`.

*Code explanation*

The multiplexer is the central node of this configuration because it manages message flow, which means that every message sent in the current computation graph is under its supervision. This leads to a higher code complexity, causing this node to have publishers, subscribers, timers, and a semaphore system to decide which of the two controller nodes can communicate with the turtlesim_node.

To avoid repeating information given in Section 2.3, some explanations will be omitted, which is the case of library inclusions and of some variables declarations. Bearing this in mind, the current file has four functions: main, lowCallback, highCallback, lowCallback and reset, as well as three global variables: publisher, timer, and SEMAPHORE (declared in Lines 3, 5, and 7, respectively). The term *semaphore* is an analogy to traffic signs, and it can have value 1 (green; its default state) or 0 (red).

The main function has the whole logic of the node. After initialising the node and declaring the NodeHandle, two subscribers are declared in Line 33.

Then, the global variable timer is initialised (Line 35) with a timer which has a duration of ten seconds, a handler function (reset), and a boolean value which decides whether the handler is called every ten seconds (false) or if it is called just once (true). The chosen timer settings indicate that once timer start function is invoked, the reset handler will take ten seconds to be deployed and the countdown stops immediately after. If the previous boolean value was set to false, the timer would deploy the reset function at every ten seconds, stopping only when timer stop function is invoked.

Variables low_subscriber and high_subscriber are also initialized (Line 39 and Line 41) with subscribers to the respective topics, cmd_vel_low and cmd_vel_high, triggering their callbacks every time a message is detected on the subscribed topics. Both queues are able to store up to a thousand messages. The handling of the multiple incoming messages, and the invocation of the respective callbacks, occurs inside ros::spin() (Line 43), which does not return until the multiplexer has been shutdown too.

The remaining functions are callbacks, and are better understood if explained alongside a runtime execution of the current configuration. Thus, imagine that random_controller is continuously publishing messages on cmd_vel_low and the multiplexer executes lowCallback (Lines 13 to 16) every time one is received. This callback simply forwards the message to the cmd_vel topic (the one turtlesim_node subscribes) if the SEMAPHORE is not red. Otherwise, the callback is called but nothing is done and the message remains inside the queue.

Now imagine that the turtle_teleop_key just sent a message to the multiplexer, which calls the respective highCallback function (Lines 18 to 26). The SEMAPHORE is set to red (0), the timer duration is reset to 10, the timer is started, and the message is forwarded. At this point, any message coming from random_controller will not be forwarded, because the SEMAPHORE does not allow it. Within ten seconds from the moment timer starts, the reset function will be called (Lines 9 to 11). However, it can happen that the timer has not reached ten seconds yet, but the multiplexer executes highCallback again thanks to an incoming message; this would be a trouble if Line 21 did not exist, since the start function does not reset timer to ten (start function does not change an already started and unfinished countdown). As side note, it is important to remember that these callbacks executions are synchronous, that is, none is executed at the same time.

Proceeding with the above scenario, it is expected that the timer finishes its ten seconds countdown at a certain moment, and that is when timer handler function (reset) is called. Whenever reset function is called, the SEMAPHORE is set to green (1) again and the forwarding of messages coming from random_controller becomes possible again.

### 2.4.1 *Launch file*

The need for a lunch file is even more obvious in this configuration, since that in order to execute every element, one would need to execute rosrun command five times. The third configuration launch file (Listing 2.8) follows the scheme of the previous ones, declaring a parameter named vel and four nodes, of which turtle_teleop_key, random_controller and multiplexer need topic remapping. Figure 6 is a possible result of executing this file. It has some straight lines arising from the interaction with the turtle_teleop_key node and some irregular lines as outcome of the interaction with the random_controller node.

```xml
<launch>
    <param name="vel" type="double" value="0.5" />

    <node pkg="turtlesim" type="turtlesim_node" name="turtlesim_node"/>

    <node pkg="turtlesim" type="turtle_teleop_key" name="turtle_teleop_key">
        <remap from="/turtle1/cmd_vel" to="cmd_vel_high"/>
    </node>

    <node pkg="turtlerand" type="random_controller" name="random_controller">
        <remap from="cmd_vel" to="cmd_vel_low"/>
    </node>

    <node pkg="turtlerand" type="multiplexer" name="multiplexer">
        <remap from="cmd_vel" to="/turtle1/cmd_vel"/>
    </node>
</launch>
```

Listing 2.8: Third configuration launch file.

Earlier, it was mentioned that a launch file is a much more flexible technique than a simple file where the ROS resources in it have all to be launched at the same time. Taking Listing 2.8 as an example, by calling that launch file with `roslaunch`, all of those ROS resources would be put running (if no error occurs until the end of `roslaunch` execution). Yet, let us now suppose that `turtlesim_node` should not be running every time this launch file is executed but only when the user wants to do so - yes, he could delete/add that element from the launch file to do so, but what happens in the case of existing five, ten or even a hundred more features like this one? Quickly, that task becomes onerous and the launch file gets hard to manage.

Fortunately, launch file syntax has `arg` and `group` tags which allow ROS users to specify certain values on which tags such as `param`, `node`, etc. will depend on to determine whether their corresponding ROS resources are launched or not. For instance, let us suppose that `turtle_teleop_key` is a node that should not always be launched. An `arg` with `name` equal to `"launchTeleop"` could be set, as well as a `group` tag with `if` equal to `"$(arg launchTeleop)"` wrapping `turtle_teleop_key` `node` tag. `launchTeleop` is now sort of a variable (an untyped one) and the content of the `group` tag is either parsed or not, depending on the value of that variable. The resulting launch file is presented in Listing 2.9. The downside of `if` attributes is the fact that only one `arg` tag value can be assigned to each one.

```
<launch>
    <arg name="launchTeleop"/>

    <param name="vel" type="double" value="0.5" />

    <node pkg="turtlesim" type="turtlesim_node" name="turtlesim_node"/>

    <group if="$(arg launchTeleop)"
        <node pkg="turtlesim" type="turtle_teleop_key" name="turtle_teleop_key">
            <remap from="/turtle1/cmd_vel" to="cmd_vel_high"/>
        </node>
    </group>

...

</launch>
```

Listing 2.9: Third configuration parameterized launch file.

Then, besides accepting the package and the launch file name, to launch this configuration `roslaunch` will also require a value to be specified for that argument, for example:

```
roslaunch turtlerand third.launch launchTeleop:=true
```

In case launchTeleop is true, the Teleop node will be launched, otherwise it will not. Note that roslaunch would not necessarily require a value if the arg tag was declared with a default value.

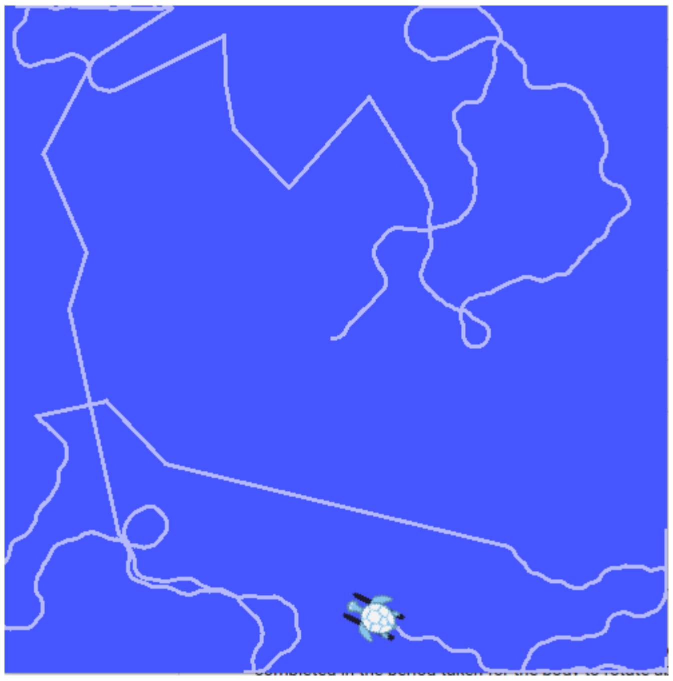


Figure 6: turtleX moving both randomly and straightly through the virtual environment.

### 2.4.2 *Analysis*

This configuration ends up being a merge of everything that was previously explored. It contains new nodes and reuses others, it declares parameters, and uses them in runtime, it has nodes that are just publishers or subscribers, or even both, and it confirms the idea that a node may also be useful to perform tasks other than hardware management. The resulting computation graph is depicted in Figure 7: two nodes publishing on topics with different priorities, and a multiplexer subscribing them and deciding which messages at which time should be forwarded to turtlesim_node.

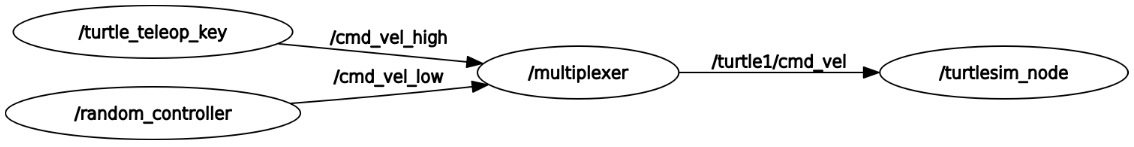


Figure 7: Third configuration computation graph.

# 3 THE HAROS FRAMEWORK

The previous chapter has shown how important ROS is to those working in robotics, due to its features and the large community it engages. However, ROS has room for improvements, namely in what concerns to quality and safety analyses. The need for this type of analyses is already relevant in standard software development, but it is even more critical in ROS where developers are mostly non-software engineers who generally lack knowledge in such important aspects. Other software development life cycle steps that are constantly ignored when programming in ROS are planning and design. Modelling software is one of the most important phases of the entire development process, since such practice significantly prevents the occurrence of faults in the final product.

It is true that ROS projects are no longer small academic projects only, neither simple experiences made by robotic enthusiasts. ROS is widely applied for industrial purposes and large companies are starting to use it on their SPLs. Therefore it would be desirable to have powerful frameworks capable of performing such quality assessment and safety verification. HAROS[1] [12], which stands for *High-Assurance Robot Operating System*, is a framework that can precisely be used to verify the safety and quality of ROS code, and that is able to perform an automatic extraction of architectural models directly from source code. A user may decide whether he performs these tasks with independent specialized tools or with HAROS, noting that the latter gathers many of them (as plugins) in an unified interface. Moreover, the model extraction process is entirely adapted and targeted to ROS.

This framework works in three stages, that will be further detailed in this chapter:

- the *setup stage*, an initial stage where the user has to setup the environment in which HAROS will be executed, alongside the definition of a project file and the commands to execute;
- the *analysis stage*, responsible for source code verification and assessment, for the model extraction process, and for the creation of files containing the resultant data;
- the third and final stage, the *report stage*, which consists in presenting to the user in a comprehensible way the data collected during analysis.

1 https://github.com/git-afsantos/haros

Although HAROS is a framework that suits the previous referred problems, it still has to be improved in some points, particularly in what concerns to efficiency and user interaction, namely when dealing with multiple configurations of the same robot. Improving this aspect is precisely the main goal of the current research. Thereafter, some problems identified during HAROS evaluation will be described in Section 3.4, alongside possible solutions to solve them.

## 3.1 SETUP STAGE

The setup stage covers those initial steps that similar applications typically require. First of all, it is mandatory to understand the commands to execute it and its configuration via project files. Other actions may also be needed but these two are essential, therefore they will be explained below.

### 3.1.1 *Commands*

This framework is launched through a command-line interface which provides some options and parameters that will be interpreted by the analyser.

Among them is `haros analyse`, a command that runs analyses and performs model extraction on certain packages, depending on the chosen options. For instance, without any options, it will scan an existing list of packages and run analyses on each one, while with option `-p`, the user has to specify the path to a project file containing information and instructions about what must be analysed. Additionally, `-n` option triggers model extraction from the source code inside the corresponding packages. The `-r` option is also helpful since it allows HAROS to look for packages in the official distribution, if it is unable to find them with the given information. Other options are possible too.

There is also a command to execute the visualiser alone, `haros viz`, which reports results from previous analyses that left the collected data inside a specific directory. By default, running the visualiser without any option, will cause it to consume the data stored in such directory. Otherwise, the user may invoke `-d` option and specify a local path containing data to feed the visualiser. The `-s` option allows to launch the visualiser at the given host and port, while `-headless` starts the visualiser server without launching it on a browser, like it does by default.

Although both previous commands are partially independent, they are, in most cases, applied after one another, which motivates the existence of the `haros full` command. This command executes `haros analyse` and then `haros viz`, allowing the user to add the options available for each command.

Other commands exist, however they are not relevant for the present research. More information about these can be found on the official distribution site[2].

### 3.1.2 *Project file*

The existence of large quantities of packages and the need for selecting only some to analyse is a very common case scenario in ROS projects. To support this scenario, HAROS can be configured with a project file written in YAML, that details the packages to be analysed. The same happens with model extraction, although the project file is not optional here, being always required. The model to extract is based on a given configuration which may comprise a set of launch files, and this information must be manually specified in the project file. Nonetheless, the model extraction process is not completely automatic and it does not recognise topics subscribed or advertised by third-party nodes, that is, nodes belonging to another package that is not included in the project file. The reason why it cannot extract models from those nodes is due to the lack of their source code. Therefore to specify the connections of those nodes in the extracted computation graph, the user will need to specify them using *hints*. More on that will be detailed in the following paragraphs.

```
project: SAFER
packages: ["turtlerand", "beginner_tutorials"]
configurations:
    teleop_key:
        launch:
            - turtlerand/launch/first.launch
        hints:
            /turtle_teleop_key:
                advertise:
                    /turtle1/cmd_vel: geometry_msgs/Twist
            /turtlesim_node:
                subscribe:
                    /turtle1/cmd_vel: geometry_msgs/Twist
    random_controller:
        launch:
            - turtlerand/launch/second.launch
        hints:
            /turtlesim_node:
                subscribe:
                    /turtle1/cmd_vel: geometry_msgs/Twist
    multiplexer:
        launch:
            - turtlerand/launch/third.launch
        hints:
            /turtle_teleop_key:
```

2 https://github.com/git-afsantos/haros/blob/master/docs/USAGE.md

```
25                  advertise:
26                      /turtle1/cmd_vel: geometry_msgs/Twist
27              /turtlesim_node:
28                  subscribe:
29                      /turtle1/cmd_vel: geometry_msgs/Twist
30  rules:
31      parameters_readers:
32          name: Parameters readers
33          description: "One or more parameters are being read."
34          tags:
35              - custom
36          query: "for n in <configs/nodes>
37                  where len(n.reads) > 0
38                  return n"
```

Listing 3.1: `turtlerand` project file.

YAML is a common format used in configuration files. Data is stored within lists and key-value dictionaries, where values can be lists, strings, or even other dictionaries. A list is denoted by square brackets and its elements are separated by commas or, alternatively, they can also be separated one per line beginning with an hyphen and indented in relation to the list key. Therefore, indentation is also an element of YAML syntax. An example of an HAROS YAML project file is shown in Listing 3.1.

Starting from the top, the `project` key stores the designation of the project and the following `packages` key corresponds to the packages to be analysed. The latter is intended to be a list, which means that it can store more than one package name. From fifth line and on, the content of the project file is predominantly related to model extraction. Thus, `configurations` contains the designation of each configuration the user wants to analyze.[3] In this example, `teleop_key`, `random_controller`, and `multiplexer` (Lines 3, 13 and 20, respectively) represent three configurations, each having inside the `launch` key the paths to the launch files from where the model will be extracted. Since they are similar, only the `multiplexer` configuration will be explained. This one has a single launch file, the same that was presented in Listing 2.8. It is fair to say that, in the present case scenario, the concepts of "launch file" and "configuration" may seem the same, yet a launch file is intended to be much more atomic than a configuration, and a configuration typically comprises several launch files.

Besides `launch`, there is a key named `hints` containing information to help in the process of model extraction. This one is required when HAROS is unable to extract certain elements due to lack of their source code or other reasons. In this case nodes `turtle_teleop_key` and `turtlesim_node` belong to third-party packages, and thus information about the topics

3 https://github.com/git-afsantos/haros/blob/master/docs/USAGE.md#defining-custom-applications

they advertise or subscribe must be manually added to the project file. This information is added inside the `advertise` and `subscribe` keys, where the corresponding values are lists containing the advertised and subscribed topics, as well as the message type they carry. A more efficient way to incorporate elements that for some reason are not automatically included in the computation graph is to instantiate them under `nodes` key, at the same level of `project` key, avoiding the declaration of similar data multiple times.

Lastly, `rules` defines a set of queries to be applied to the extracted model. Listing 3.1 lists only one as an example, but it is possible to define others. In this case, the query identifier is `parameters_readers` which comprises its name, description and tags. The query itself is defined inside the `query` value and, in this case, it will identify nodes that are using the parameter server. Rules here defined, will later be run, and may originate one or more issues every time the query yields some result.

## 3.2 ANALYSIS STAGE

The main goal of the previous stage is to prepare HAROS to the analysis. This stage is heavily dependent on plugins and these are responsible for the great majority of information that is reported afterwards. Essentially, a plugin is a Python package that comprises at least the `plugin.yaml` and `plugin.py` files. All the metadata related to this plugin is stored within the YAML file. For instance, it contains its name, version, computed metrics, etc, and whether the plugin only works with certain programming languages. On the other hand, `plugin.py` contains all the necessary programming logic to run the plugin. An example plugin is the query engine that HAROS applies to the extracted model. This one is based on *PyFlwor*[4] which provides a language to query Python objects.

The choice of Python as the language to develop HAROS and its plugins is mainly due to the fact that ROS itself is also developed in this language, which simplifies the interface between the two frameworks. Python is also a good language to perform scripting tasks, which is typically the case in most analysis plugins, that are usually just wrappers for external tools.

### 3.2.1 *Model extraction*

The model extraction follows a particular algorithm that is optimised to collect as much knowledge as possible from source code. The data collected by this algorithm is stored inside JSON files. In particular, `configurations.json` is one of those files, containing information about the extracted model that will be consumed by the visualiser.

4 https://pypi.org/project/haros-plugin-pyflwor/

The process of extracting an architectural model from source code is not easy, since it must consider projects which may differ a lot in the way they are built. Obviously, bigger projects have more intricate usage of ROS, ending up using some of its aspects that are typically not used in smaller projects. The present section aims to cover the main steps of such algorithm, showing how the extraction process happens.

First of all, there is a function named `extract_metamodel` that is in charge of reading the `project file`, and computing sets with information of the packages, source files, repositories, and nodes found within this file. This function reveals that the algorithm has three important phases: the extraction of source artefacts, the parsing of ROS primitive calls, and the establishment of the final model.

When a ROS developer produces a package, he must build it under certain basic guidelines. Among other rules, a package folder must be inside a *catkin* workspace and they all must contain an XML manifest and a `CMakeList.txt`. With that said, to assess the existence of a package is relatively easy, which is done by traversing the directories inside the workspace having in mind those guidelines. It is also possible to specify directly the package path or to use ROS tools that are tailored for that mission. After the identification of a package, HAROS examines its folders and files, namely the source code files. The parent folder of the package is also analysed, to determine if it is part of a repository.

To extract information regarding nodes, HAROS does more that just a simple parsing of the respective source files. In fact, doing so would be really inefficient since one would have to parse every single file. A much more rational approach is to check the CMake build file which is responsible for generating object files. The `CMakeLists.txt` gathers instructions to do it, and also the location for the source files that feed the compiler. Yet, it is wrong to assume that each file stated in `CMakeLists.txt` is a node. In fact, those might represent libraries or other type of tools that must not be confused with nodes. That problem is solved by looking at the content of each file and tracing the call to `ros::init`, the function that instantiates a node. To locate this function call, HAROS may need to inspect other files that are imported by that node.

Next, HAROS has to link all artefacts to build the application computation graph, and it does that by looking at ROS primitive calls. Functions like `advertise` and `subscribe` are directly related to the existence of topics, but some are particularly hard to parse. Another difficulty that HAROS might have sometimes, is to determine the full name of a ROS topic, service or parameter, for example when it is defined using variables which value can only be known at runtime. In those cases, HAROS replaces the unknown terms with a wildcard `?`.

Finally, HAROS inspects the content of the configurations specified within the project file. For each launch file, the algorithm does what `roslaunch` command also does: reads the `launch` tags and collects the information inside. Finally, HAROS reads the hints inserted by the user, in order to complete the model. Every hint that the user specifies within the project file is considered as a fact, so they should only be given when strictly necessary [11].

## 3.3 REPORT STAGE

The report stage aims to make data generated in earlier stages intelligible, since it is not feasible to read and inspect information stored in JSON files. Most HAROS users do not even expect to see these files, but a practical interface that shows them the results obtained after the analysis. The visualiser works with the help of web technologies and under normal circumstances it opens a page inside a web browser containing statistics, issues, and the architectural model (if the user chose to extract it).

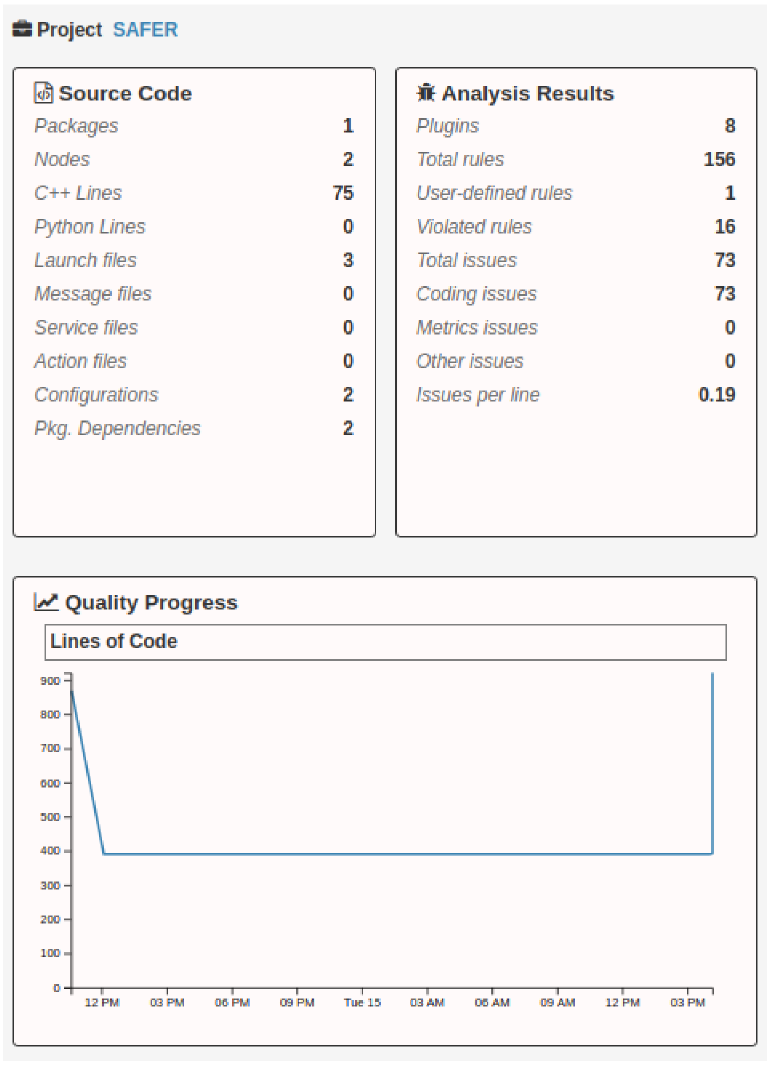
The default web page tab that HAROS users see is named `Dashboard`, and contains an horizontal navigable bar that offers the user the possibility to easily switch between HAROS main features. This bar is stationary, that is, it remains unchanged across the other tabs. Besides `Dashboard`, HAROS provides four more tabs: `Packages`, `Issues`, `Models`, and `Help`. The intention of the present section is to explain the content of each one of them, with exception of `Help`, which is relatively straightforward. This tab provides some general definitions and information about HAROS.

### 3.3.1 *Dashboard*

The `Dashboard` has an initial drop down menu which allows to chose the project to inspect. Sometimes, there will be more than one to chose because HAROS stores previous project analyses and compares them with recent ones. As seen in Figure 8, this tab has some statistics divided into three boxes: `Source Code`, `Analysis Results`, and `Quality Progress`. They report the number of lines of code, ROS files, rules applied in the analysis, issues, etc. However, `Quality Progress` produces a result slightly different from the others, which is a chart emphasising the variation of different metrics through time. Some of them are lines of code, lines of comments, number of issues, etc.

### 3.3.2 *Packages*

The `Packages` tab represents an overview centred in the quantity of issues of all the analysed packages. It consists of a graph where nodes represent packages and edges represent dependencies between them. Nodes will have different colours, varying according to the rate of issues detected in the analysis. The colour scale goes from white, which represents zero issues in the whole package, to red, representing a rate of more than one issue per twelve lines of code. Moreover, HAROS allows to filter graph nodes by a tag. Clicking on a given


Project SAFER

**Source Code**

| | |
|---|---|
| Packages | 1 |
| Nodes | 2 |
| C++ Lines | 75 |
| Python Lines | 0 |
| Launch files | 3 |
| Message files | 0 |
| Service files | 0 |
| Action files | 0 |
| Configurations | 2 |
| Pkg. Dependencies | 2 |

**Analysis Results**

| | |
|---|---|
| Plugins | 8 |
| Total rules | 156 |
| User-defined rules | 1 |
| Violated rules | 16 |
| Total issues | 73 |
| Coding issues | 73 |
| Metrics issues | 0 |
| Other issues | 0 |
| Issues per line | 0.19 |

**Quality Progress**

Figure 8: `Dashboard` tab.

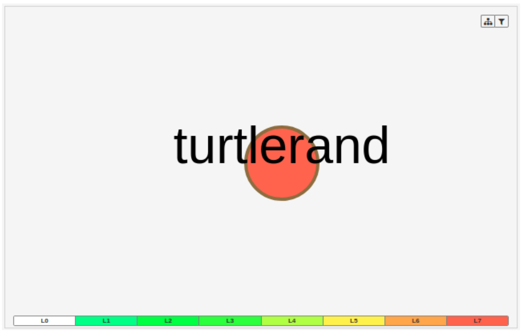
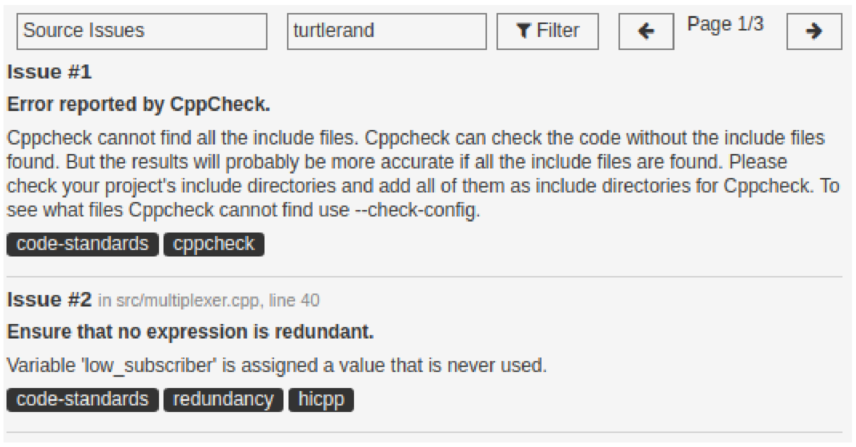


Figure 9: `Packages` tab.



Issue #1

Error reported by CppCheck.

Cppcheck cannot find all the include files. Cppcheck can check the code without the include files found. But the results will probably be more accurate if all the include files are found. Please check your project's include directories and add all of them as include directories for Cppcheck. To see what files Cppcheck cannot find use --check-config.

code-standards cppcheck

Issue #2 in src/multiplexer.cpp, line 40

Ensure that no expression is redundant.

Variable 'low_subscriber' is assigned a value that is never used.

code-standards redundancy hicpp

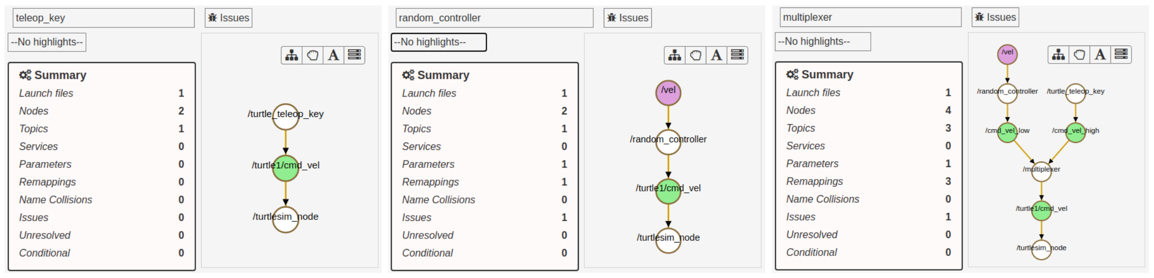
Figure 10: `Issues` tab.

package prompts a few more options: focus the package, get additional information (authors, dependencies, size, etc.), and redirect user to see the issues of the package. Figure 9 depicts this tab for the running example.

### 3.3.3 *Issues*

By selecting the `Issues` tab, developers can easily see the issues reported by the analysis plugins, each showing exactly where the problem is located, what caused it, some tags that characterize it, and sometimes, even how to solve it. In Figure 10, some issues reported by plugins can be seen focusing on metrics and coding standards commonly applied in programming languages, such as Google's C++ Style Guide [5], High-Integrity C++ [6], or MISRA C++ [7]. Filtering is also possible here, whether by selecting the pretended package, the pretended tags, or even both.

5 https://google.github.io/styleguide/cppguide.html
6 https://www.perforce.com/resources/qac/high-integrity-cpp-coding-standard
7 https://www.misra.org.uk/misra-c-plus-plus/



Summary (teleop_key)

| | |
|---|---|
| Launch files | 1 |
| Nodes | 2 |
| Topics | 1 |
| Services | 0 |
| Parameters | 0 |
| Remappings | 0 |
| Name Collisions | 0 |
| Issues | 0 |
| Unresolved | 0 |
| Conditional | 0 |

Summary (random_controller)

| | |
|---|---|
| Launch files | 1 |
| Nodes | 2 |
| Topics | 1 |
| Services | 0 |
| Parameters | 1 |
| Remappings | 1 |
| Name Collisions | 0 |
| Issues | 1 |
| Unresolved | 0 |
| Conditional | 0 |

Summary (multiplexer)

| | |
|---|---|
| Launch files | 1 |
| Nodes | 4 |
| Topics | 3 |
| Services | 0 |
| Parameters | 1 |
| Remappings | 3 |
| Name Collisions | 0 |
| Issues | 1 |
| Unresolved | 0 |
| Conditional | 0 |

(a) `teleop_key` (b) `random_controller` (c) `multiplexer`

Figure 11: Computation graphs of the three example configurations.

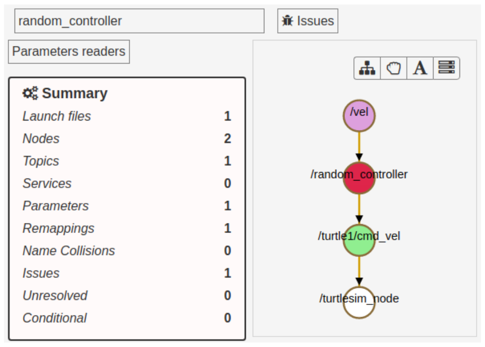
### 3.3.4 *Models*

Alongside `Issues`, the `Models` tab is also one of the most relevant features of HAROS. It displays the extracted architectural models for the configurations previously defined within the project file. The user has a drop down menu where he can switch between the available configurations. The nodes of the depicted computation graph are ROS resources and the edges represent the existing relations between them.

In order to make this feature even more readable, HAROS assigns different colours to those nodes, accordingly to their type in ROS. For instance, white nodes are ROS nodes, green nodes are topics, purple nodes are parameters, and blue nodes are services. Dashed nodes are conditional entities (nodes whose presence in configurations may vary according to a conditional statement). By selecting one of these, HAROS presents a few more details about it, like its location and data type, in the case of parameters, or its conditions and the type of messages they carry, in the case of topics. Other options like exhibition of parameters or graph nodes names are available, as well as a summary with statistics about the extracted model. If for some reason HAROS is unable to find a certain resource name, it will place a question mark instead of its name. Figure 11 shows the extracted computation graphs for the three configurations developed in the previous chapter.

As one can see, the extracted models are very similar to those extracted by `rqt_graph`, but there is a big difference in the way they are obtained. HAROS does it through static analysis while `rqt_graph` does it dynamically at runtime. Furthermore, HAROS provides an essential ingredient to model analysis: the possibility to query these models. As Figure 12 shows, this tab contains a drop down menu where the user can select one of the rules defined in the project file. By selecting one of those, HAROS reacts by highlighting (colouring with red) the graph elements that are the answer to the respective query.

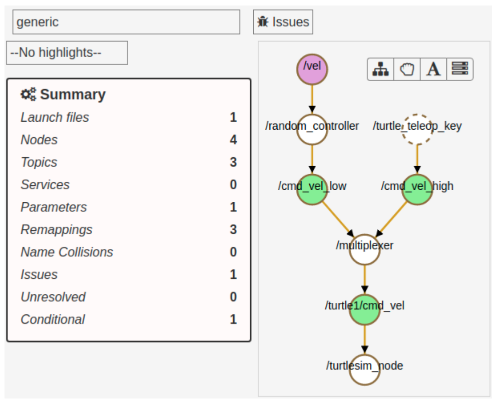
In what concerns to `Models` tab, there is another useful functionality that users can take advantage of to make the computation graph more enriching, that is the ability to parse conditions under which launch file elements are. In the computation graph, these elements are represented in the same circumstances as the unconditional ones, but in spite of being sur-



**Summary**

| | |
|---|---|
| *Launch files* | 1 |
| *Nodes* | 2 |
| *Topics* | 1 |
| *Services* | 0 |
| *Parameters* | 1 |
| *Remappings* | 1 |
| *Name Collisions* | 0 |
| *Issues* | 1 |
| *Unresolved* | 0 |
| *Conditional* | 0 |

Figure 12: HAROS pointing `random_controller` node as an answer to the query.



**Summary**

| | |
|---|---|
| *Launch files* | 1 |
| *Nodes* | 4 |
| *Topics* | 3 |
| *Services* | 0 |
| *Parameters* | 1 |
| *Remappings* | 3 |
| *Name Collisions* | 0 |
| *Issues* | 1 |
| *Unresolved* | 0 |
| *Conditional* | 1 |

Figure 13: Conditioned element (dashed) in the computation graph.

rounded by a thick line, they are surrounded by a dashed line. Listing 2.9 depicts a launch file in which the `turtle_teleop_key node` is conditional to the setting of the `launchTeleop` argument. Therefore, after submitting that launch file to be analyzed by HAROS, in computation graph `turtle_teleop_key` appears dashed, as shown in Figure 13.

However, whether a computation graph element is under one or more conditions, that information is kept inside an array in the element's object which in turn is stored in `configurations.json`. Each element of the array is a value on which that element is depending. For instance, if the array has three values, the corresponding element will be dashed, if it has only two, the element will still be dashed, but if it has none, then the node is not under any condition, appearing with a thick line.

## 3.4 SHORTCOMINGS

Although HAROS has proven to be a very helpful framework in what concerns to quality and safety assurance of ROS applications, it still could benefit from some improvements in order to become more efficient and user friendly. The experiments carried out with HAROS using the running examples have shown that HAROS major problem is related with the inability to handle variability.

Assigning generic names to the topics advertised or subscribed by any ROS resource is a good practice, but that turns out to be a problem to HAROS. At the moment, the HAROS analysis stage is extracting the information of which resources advertise/subscribe which topics from the source code files. Nevertheless, given that these topics identifications are generic, they need to be renamed in order to suit their purposes. That process, also known as remapping, usually takes place inside launch files, where ROS resources are given the actual specific identification of each topic they must advertise/subscribe. Thus, despite the fact that launch files contain such important information, HAROS does not take that into consideration, which results in bad or inexistent relations between ROS resources in the computation graph. The problem can be temporarily fixed by providing hints for the missing information, but it still is a significant limitation of this framework.

Another shortcoming is related to the visualisation of architectural models through a graph. It is true that such feature helps a lot in terms of readability. Nonetheless, it would be very helpful to compare different configurations and HAROS fails to provide a proper way to do it. For instance, when the user switches from one configuration to another, the layout of the graph does not seem to obey to a particular procedure, and two similar configurations can have totally different layouts. Additionally, configurations differing in just one or even a few nodes, should be given the chance to be represented as a single variable structure. In another words, it should be possible to merge configurations in a single annotated graph, something that would save a lot of time to HAROS users.

The HAROS `Models` visualiser is not the only tab where variability needs to be managed; the `Issues` tab may also display some content that is related to some of the computation graph elements. Therefore, it would be desirable to also consider variability in the `Issues` tab, namely allow the user to filter it by somehow selecting features.

To address these shortcomings, it is convenient to add explicit variability management to HAROS, something that can be achieved with variability models such as *feature models* (detailed in the next chapter), which should be somehow encoded in project files or in HAROS itself.

# 4 MANAGING SOFTWARE VARIABILITY

Variability exists in all types of software, but it is more prevalent in some of them, particularly in what concerns to robotics development. This is due to the fact that robots, on the whole, are intended to be autonomous machines capable of interacting with the surrounding environment. The less controlled this environment is, the more variables a robot will need to handle, which increases variability.

The present chapter starts by discussing why variability is intrinsically related with robotics. Then it presents some techniques to manage variability in software, namely *Software Product Lines* (SPLs) and *Feature Models*. It finishes by presenting with detail the *Text-based Variability Language* (TVL), a *Domain-Specific Language* (DSL) that can be used to describe the variability of a system, as well as different useful operations available in TVL library.

## 4.1 SOURCES AND CHALLENGES OF VARIABILITY IN ROBOTICS

Sergio et al. [5] conducted a series of experiences and interviews where they identify four sources of variability in robotics: the customer requirements, the environment, the robot hardware, and the middleware. They also also present some key challenges related to the management of variability in this application domain.

### 4.1.1 *Sources*

#### *Customer requirements*

Robotics can either serve industrial or research purposes. In the case of industrial robots, the set of usage scenarios is very large, as customers interested in this type of machines have many different and disparate needs. Customers vary from healthcare to military or they can even be regular persons seeking for innovative solutions. Inevitably, such fact leads to different hardware and software implementations on robots. Even robots built for the same goal can end up being very different, as customers are able to choose whether they want or not to have as additional features on their machines. As such, this is one of the more direct sources of variability in robotics.

*Environment*

When building robots, developers must be aware of the environment in which they will operate. A robot may work as it should under a controlled environment and become completely unable to operate when this factor is altered. For instance, robots working on assembly lines are designed for a specific task, generally, to pick an exact object and mount it somewhere. Such robots cannot operate out of this well controlled environment simply because they were tailored for a specific task. In those cases, environment variability is easily handled at design time as developers can predict nearly all situations. Yet, in some usage scenarios it is even impossible to capture all of the unpredictability of the environment, as it is the case of autonomous vehicles. In the latter case, agents in the environment are constantly changing and machines cannot always recognize them nor their actions and intentions. All of that leads robots to have multiple features that are specifically made to operate in different environments and a decision has to be made about those who will be launched and those that will not.

*Robot hardware*

Robots hardware has a direct impact in the software to be developed, as different hardware mounted on identical robots require distinct drivers and approaches when programming. Nevertheless, this is not the only aspect in which variability is linked to hardware. For example, robots may have different types of movements and the hardware layout on certain robots can affect them in their locomotion process. This forces developers to rethink software algorithms in order to be adapted to different maneuvers.

*Middleware*

Robotic middlewares are another source of variability, as none is standard in robotic development. In the present research the chosen middleware was ROS but there are more of them in use in robotics development. Different middlewares lead to different software approaches and distinct used libraries. These frameworks are themselves responsible for some variability too, as they change over the time, and new versions with different features may introduce some entropy into the development process.

### 4.1.2 *Challenges*

#### *Multi purpose robots*

Multi purpose robots may be the type of robots where variability is harder to manage, since they have to be prepared for a higher number of different scenarios. In order to build those robots, customers must detail every aspect of the desired outcome as well as the tasks they will be used for.

#### *Generic software*

As said before, software is generally built towards a specific piece of hardware, suppressing any possibility of reusing it latter on another similar component. To cope with this variability, the development process could be targeted to develop generic software that can handle different hardware solutions. Of course this will be more complex and time consuming, but in the long-term, this type of development would be compensated by saving human resources and mostly important, by reducing variability.

#### *Middleware and framework heterogeneity*

Although having slightly different goals, it would be useful if robotics middleware and framework developers joined efforts to define a common architecture to be applied in robotic software development, to avoid heavily disparate differences. The definition of other standards and good practices would also be convenient and helpful. The downside is that developers would have to restructure their software to accommodate older versions.

#### *Dynamic variability*

Variability can be tackled either statically or dynamically. Static solutions have proven to be the most immediate path to handle variability, since it essentially consists of predicting all possible situations and encode them in the software. Typically, this method works fairly well in controlled environments. However it fails when confronted with unfamiliar situations.

In comparison, dynamic variability is much harder to treat as it appears when it is least expected. In those circumstances, software has to find a workaround, one that causes no harm to the surrounding environment. For instance, a straightforward solution to an autonomous car moving in a motorway when it finds something it cannot understand is to stop, yet that behaviour might put in danger other vehicles. Drones are another example where stopping immediately is not an option either. The manufacturers of the previous examples generally opt for the most obvious solution: transfer vehicles' controls to a human. Examples of robots which treat variability dynamically are, for instance, those that automatically activate certain features when they are in the presence of a human being or drones which activate an automatic descent when a problem is detected. In short, static variability does not solve the problem entirely and the solution to manage dynamic variability may be in implementing algorithms where software can make independent decisions.

## 4.2 SOFTWARE PRODUCT LINES

To properly deal with variability, the robotic software development process should apply a software engineering paradigm specifically tailored for that purpose. Developing the robotic software as a *Software Product Line* (*SPL*) [4] might be a possible solution. SPL engineering consists of a set of techniques, practices and tools which are applied in the development of similar software systems. Those systems are composed by smaller software units (denoted *features*) that are designed to be functional in different software products with different needs and missions. These features may be optional or mandatory in the development life cycle. By analogy, SPL development might be compared to an hamburger shop where customers have different hamburger options with distinct ingredients and flavours they can opt for.

In general, SPLs have two main approaches to implement features. One is called *annotative* and the other *compositional* [7]. In the annotative approach, software developers include in the source code annotations, akin to conditional statements, that control which code implements each feature (practical examples of such annotations are the `#ifdef` and `#endif` directives, often applied in *C*-like languages). The compositional approach promotes the development of features in different modules totally independent from each other. These features can later be combined to form configurations/products of that SPL.

Software development can benefit from such paradigm in many ways. Besides improving reusability, it eases the development process, for instance, when changing and upgrading existing software with new features. When software manufacturers decide to use SPLs development they also get significant cost and development life cycle time reductions and improvements of software quality and productivity. In essence, such approach would help tame variability as it provides a method where developers can easily resort to an existing feature collection, that gets even richer as developers build new features and upgrade others. It also allows manufacturers to effectively answer to customer requirements and environment reshapes.

## 4.3 FEATURE MODELS

In 1990, Kang et al. [6] proposed a method for better describing software features, i.e. small software pieces that can be put together to form bigger software systems. The method stands by the name of *Feature-Oriented Domain Analysis* (FODA), and its goal is to establish and emphasize differences and commonalities across software systems in a certain domain. The report uncovers a method to analyse the given software domain and also a technique to record in a so-called *feature model* the existing components of such domain. By doing so, software reuse becomes easier as developers can look at an intelligible model in spite of looking directly to the source code. Such feature models are essential for developers to understand and simplify the reuse of the existing software features.

The development of a feature model requires a previous survey of the domain under analysis, in order to make it precise and to avoid misconceptions. Having this in mind, Kang et al. propose the following steps to perform a proper domain analysis:

1. Gather information about the domain from different sources, for example: books, technical papers, existing applications, domain experts, previous works, etc. This will allow developers to limit the domain to the essential.

2. Analyse the domain of each identified application and assess if it fits in the general domain. Problems and issues must also be declared.

3. Anticipate problems that will occur and may disturb features within the domain. Those can also be new requirements or others subject to changes.

4. The analysis of the variability is probably the most important step to take and also a matter of particular interest for the current research. Here, one should start developing a feature model consisting of features within the identified domain, as well as defining environment variables, feature constraints, domain ground truths, etc. Finally, a feature model should be defined.

5. Compare the model with the different sources of step 1. This step may expose some hidden variability that must be declared in the model.

6. Identify commonalities and differences across applications features and assess their implementation.

7. Calculate improvements in terms of resources brought by context analysis, as well as benefits arising from it. At this phase, the analysis should be done and correctly documented.

8. This final step, aims to test the model against foreign applications that were not considered in the current context analysis. This will allow to improve the model and equip it with more knowledge concerning variability.

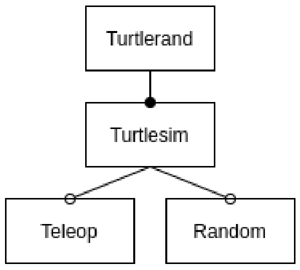


Figure 14: `turtlerand` feature model.

Earlier in Chapter 2, multiple software features were built and others reused in order to build a software system that allowed to control a simulated robot turtle. Not all of them were mandatory, some were optional. These relations can be clearly identifiable in a feature model.

A feature model illustrates and documents the interactions between features present in multiple software applications, all of them with common characteristics. In general, every time one refers to a feature model, one is in fact referring to a *feature diagram* (such as the one in Figure 14, describing the variability in our running example), which is just one of its components. Nevertheless, feature models also have other components such as composition rules (mutual dependency and mutual exclusion relationships), issues and decisions, and a system feature catalogue (a set containing the existing features).

### 4.3.1 *Feature diagrams*

Feature diagrams are useful in the sense that they allow software engineers to immediately understand the variability in a given system through a visual model. This diagram is usually represented as a tree where nodes are different features. In order to draw a feature diagram, one has to pay attention to the following rules:

- The root feature is the most general feature and it is at the top of the feature model hierarchy.
- The edges that are connecting features demonstrate that child features can only be part of an instance of the model if their parent is also part of it.
- A feature may be represented as optional, when it displays a small empty circle on top of its name.
- Features with no special graphical notation are mandatory, yet, to contrast with optional features, those might also exhibit a small full circle.
- Alternative features are represented as children of the same parent feature with an empty arc embracing those features, meaning that of those, one and one feature only must be chosen.

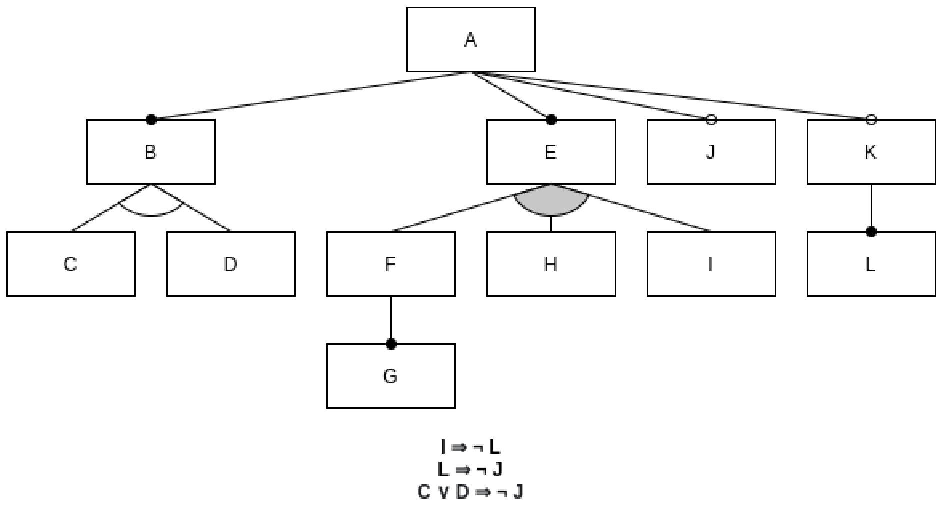


Figure 15: Example of a feature model.

- `or` features sets are represented by a full arc that embraces two or more features. This means that of those, at least one must be selected.
- Beyond the constraints implicit in the tree, one may also specify others that cannot be visually depicted. These are logic propositions that are placed alongside the feature diagram tree and they also establish relations between the existing features. They are called cross-tree constraints.
- When one wants to instantiate a configuration of the feature diagram, the selected features for that configuration are drawn within a rectangle. In case of all the features are already within rectangles, these must be filled with a color that differentiates features within the configuration from the others.

### 4.3.2 *Feature diagram example*

Figure 15 feature diagram was built to better illustrate the different symbols and constraints that one can find in a more complex model.

Therefore, this feature diagram contains twelve features, named from `A` to `L`, where `A` is the root feature and it has four direct children. Whereas `B` and `E` are mandatory, `J` and `K` are optional, and `L` is selected every time `K` is too. `B` has two children and the empty arc under its rectangle denotes that one and only one feature must be chosen and it will either be `C` or `D`.

Feature `E` has three children, and the full arc under its rectangle depicts an `or`, where at least one of those three features must be selected, reminding that selecting `F` will force `G` to be selected too.

Regarding the cross-tree constraints, the running example has three. One, that tells us that `I` cannot be selected at the same time has `L`, another that tells us than `L` cannot be selected at the same time has `J` and another that tells us that if `C` or `D` is selected, `J` cannot be too.

Let us now instantiate a configuration out of this model, considering all the constraints written before. Starting by selecting `A`, `B` and `E` are automatically selected too. Then, the feature `C` is chosen out of the two children of `B` and finally, `F` and `H` are selected too, which automatically forces `G` to be selected. The resulting configuration is shown in Figure 16.

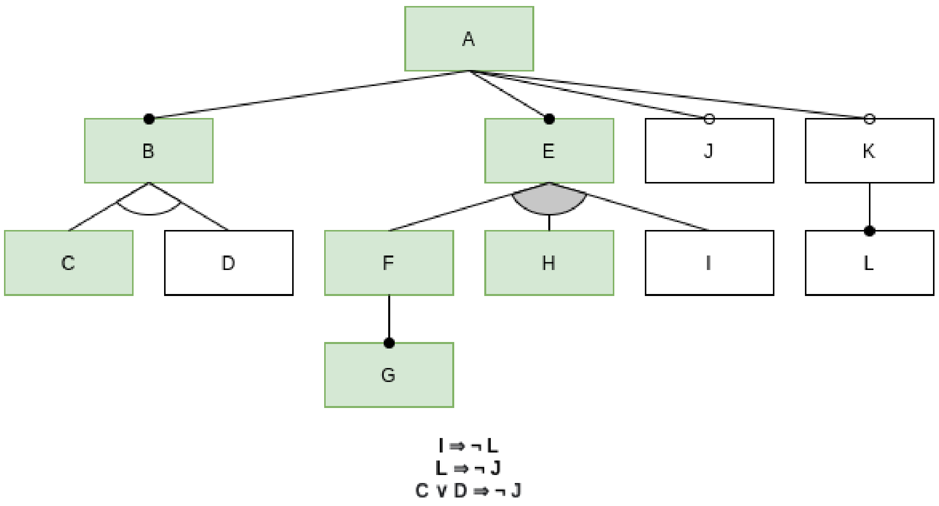


Figure 16: Example of a feature model configuration.

## 4.4 TVL: A TEXT-BASED VARIABILITY LANGUAGE

To assist with SPL development method, experts use feature models to represent the variability. Those feature models are usually defined according to the FODA notation, where the main component is the graphical representation of the features, the feature diagrams. Yet, for large systems it might not be the best approach due to the following problems:

- software engineers often see graphical syntax and models as a burden [3], specially when it comes to drawing them;
- since a feature diagram is displayed as a two dimension image it can become really big, making it hard to understand and to search for any of its elements, meaning that it is not scalable;
- the graphical notation is too poor to differentiate multiple types of elements (for example, attributes);

To solve those problems, Quentin et al. [1] proposed TVL, which stands for *Text-based Variability Language*, a feature modelling language where features are not represented in diagrams but through text. At first, such approach might not be the best in terms of representation, but in practical terms there already exist lots of tools to process text, which makes this language easy to handle and scalable too. However, a visual representation can easily be inferred from TVL feature models.

TVL has a *C*-like syntax, with its language elements being represented within blocks enclosed by braces. Two other characteristics that TVL inherits from *C* are the syntax to denote comments (`//` and `/* */`) and declarations (end up in semicolons). The reason why the authors adopted this type of syntax is related with the familiarity of developers with languages they already knew, such as C or Java. Such decision can help gather more users, since the learning process of TVL is reduced.

The following subsections refer to the main syntactic components of TVL, namely: features, attributes, expressions, constraints and modularisation strategies.

### 4.4.1 *Features*

Just like in FODA feature models, a TVL model starts with the keyword `root` followed by the designation of the root feature and then the rest of the model between braces, as shown in the Listing 4.1. The model evolves henceforth and elements written within those braces should be indented in relation to its parent. Some other syntactic elements, that will be later explained, may appear at the same hierarchical level of the root feature.

Inside of those braces, the specification of the root sub-features is introduced by the keyword `group` followed by a decomposition operator, which can be one of the following: `allOf` (and-decomposition), `oneOf` (xor-decomposition), `someOf` (or-decomposition) or `[i..j]` (cardinality-based decomposition, where `i` is the minimum number of features that is possible to choose, `j` the maximum, and `*` the absolute number of sub features of that parent). The decomposition operator (Line 1) is followed by a block enclosed by braces, where the root sub-features are declared separated by commas. The optional ones should be preceded by the keyword `opt`[1]. In our example, the root only has one sub-feature called `Turtlesim`.

If it happens that one of those sub-features has any child (in our example, `Turtlesim` has two), the whole process is repeated again with the exception that the decomposition operator is declared right after the feature designation with no `root` keyword.

In TVL, scalability can also be a problem, just like in FODA feature diagrams, since when the model has too many layers it becomes incomprehensible. In TVL, the model can easily become a giant chain of nested features. To avoid those situations, it suffices to declare the designation of a sub feature and declare its sub features anywhere else on the model at the same hierarchical level of the root feature (Lines 6 to 11, Listing 4.2). Closely related to this

1 Cardinality and optionality fields can overlap because a group of TVL feature children can be declared under the decomposition operator `allOf`, even when at least some of those children features are optional. In that case optionality prevails.

is the freedom that TVL provides to its users to allow the addition of more attributes or constraints just by recalling the feature designation anywhere else on the model document at the same hierarchical level of the root feature too.

Though rare, TVL also allows case scenarios where a feature can have more than one parent. Thus, its designation must be preceded by the keyword `shared` every time it occurs in the model again (it suffices to declare this keyword only at the first mention of that feature designation).

```
root Turtlerand {
    group allOf {
        Turtlesim group allOf {
            opt Teleop,
            opt Random
        }
    }
}
```

Listing 4.1: `turtlerand` TVL feature model.

```
root Turtlerand {
    group allOf {
        Turtlesim
    }
}

Turtlesim {
    group allOf {
        opt Teleop,
        opt Random
    }
}
```

Listing 4.2: A variant of Listing 4.1 `turtlerand` TVL feature model.

### 4.4.2 *Attributes*

TVL also supports the specification of the attributes of a certain feature and recognizes four types of data which can be assigned to them: boolean (with keyword `bool`), integer (`int`), real (`real`), and enumeration (`enum`). The lack of the string data type and others is seen as a downside of TVL, yet, the authors argue that it is intentional in order to keep the language simple and concise [3].

Attributes are declared inside the block of the feature they are related with, their data type keyword must precede the attribute designation just like it happens in C language and the declaration of each must end with a semicolon.

Some attributes may be declared within a range of values. An integer, for example, may be declared between 0 and 100. To do so, one only has to add, right after the attribute designation, the keyword `in` followed by the range (`[0..100]`) and a semicolon. The syntax may be slightly different if one chooses to add expressions to the declaration.

On the other hand, enumeration attributes have necessarily to declare the list of values they can take. These values are separated by a comma and the list is surrounded with braces.

Other keywords are available, such as:

- `is` - keyword followed by an absolute value, a function or an expression, according to its data type;
- `ifIn` - this keyword means "in case the feature is selected" the expression next to the colon is executed;
- `ifOut` - this one has the opposite meaning of the previous keyword, meaning "in case the feature is not selected"; it usually complements `ifIn`, but it can also be used alone;
- `constant` - this keyword is used in the declaration of an attribute to prevent it from being modified; its designation is preceded by the `constant` keyword, followed by the `is` keyword and then by the constant value.

### 4.4.3 *Other keywords and operators*

As said before, TVL is much more manageable than common FODA feature models. Thus, it allows the use of classical operators, cross-tree constraints, aggregation functions and other useful keywords, such as:

- `-`, `+`, `*`, `/`, for numeric values;
- `&&`, `||`, `->`, `<->`, `!`, for boolean values;
- `>`, `<`, `=>`, `<=`, `==` for both;
- `sum` (sums), `mul` (multiplications), `min` and `max` (minimum and maximum calculations, respectively), `abs` (absolute values), `avg` (averages), `count` (counts), `and`, `or` and `xor`;
- `children` keyword, responsible for gathering all the children of a feature;
- `selectedChildren` is another keyword example, responsible for gathering all the selected children of a feature;

- `fct` keyword, that allows to compose the previous keywords with a certain attribute, which gathers the values of the child attributes.

### 4.4.4 *Constraints*

Constraints (the equivalent to feature diagrams cross-tree constraints) of a feature are written within the body of the feature definition, just like its sub features designations. Some keywords seen before are also allowed to be applied in constraints, as well as the `parent` keyword, which can be used to refer/define parent attributes values of a certain feature. The end of a constraint is denoted by a semicolon.

### 4.4.5 *Modularisation*

Most languages have some method to facilitate code reuse. The process is called modularisation, since it allows to partition the code according to its purposes, forming modules. The most obvious application of this method in most programming languages is the declaration of functions, which are used later on the code. In TVL, enumerations, for instance, can be declared anywhere in the model document, and they can later be used on multiple feature definitions as an attribute type. The definition of structured types is also a practice and those can also be used as attribute types just like enumerations.

Another important detail of TVL is the `include` statement, which will import, at the point it is declared, the content of the referenced file.

### 4.4.6 *TVL feature model example*

Recalling the feature model example presented in Figure 15, it is possible to convert it to a TVL feature model (Listing 4.3). Some important aspects to highlight are:

- the cross-tree constraints in Lines 7 to 9;
- the `B` feature empty arc which translates to the `oneOf` operator in Line 12;
- the `E` feature full arc which translates to the `someOf` operator in Line 19;
- the decomposition operator in Line 31 - it could also be `allOf`, `[1..1]`, `[1..*]` or `[*..1]`.

Some additional attributes and constraints were also added to exemplify how to apply some of the previous concepts. Of which:

- the boolean attribute `y` in Line 16, which is the result of the `or` operation applied to the `x` attribute values of `B` selected children;
- the boolean attributes `x` in Line 36 and 39, which are `false` and `true` respectively.

```
0  root A {
1      group allOf {
2          B,
3          E,
4          opt J,
5          opt K
6      }
7      I -> !L;
8      L -> !J;
9      C || D -> !J;
10 }
11 B {
12     group oneOf {
13         C,
14         D
15     }
16     bool y is or(selectedChildren.x);
17 }
18 E {
19     group someOf {
20         F,
21         H,
22         I
23     }
24 }
25 F {
26     group allOf {
27         G
28     }
29 }
30 K {
31     group [*..*] {
32         L
33     }
34 }
35 C {
36     bool x is false
37 }
38 D {
39     bool x is true;
40 }
```

Listing 4.3: Example of a TVL feature model.

After submitting Listing 4.3 to the reference TVL parser (described afterwards), the feature model was considered satisfiable with twenty possible configurations. The configuration `A`, `B`, `C`, `E`, `F`, `G` and `H`, which matches Figure 16 configuration, is among those.

### 4.4.7 *The TVL reference implementation*

Having a feature model in text format may be an advantage according to the prior stated reasons, but TVL offers more than that. Its authors created a Java library with a reference implementation of TVL [2] that is able to perform selection/deselection operations upon the TVL feature model. It does that by submitting the model to a logical solver, where a state is assigned to each feature, as well as all of the constraints between them. A feature state eventually changes according to the operation the user does.

The TVL library has a module that parses and validates the TVL file, and builds a Java object with its content. It is the resulting object that implements the prior functionalities, but it also implements methods for counting features, computing possible configurations, evaluate satisfiability, etc.

The TVL library has two distinct usage modes: a simple one where its user runs a TVL file and submits it to multiple tests inside a terminal - first a validity test, and then operations such as counting the number of features, perform satisfiability tests, returning the list of possible configurations (in TVL documentation referred as products), etc. - and another more complex where the user has to develop Java code to get the most out of it - allowing operations such as declaring a feature model, initialize it with a state, selecting/deselecting some of its features, and more. The simpler use does not suffice for the goals of this thesis, so we will now describe with more detail the second one.

Bearing that in mind, the first class to analyse is `TVLParser`. This class parses the content of a given TVL file and pours it into a structure which records every data in that file. This class contains some useful methods, yet it is not the one that matters the most for the present work. Among those constructors and methods are:

- `TVLParser(File inputFile)`: the constructor that creates a new object to the given path of a TVL file;
- `public FeatureSymbol getRoot()`: returns a `FeatureSymbol` object which contains all of the information of the TVL model root feature;
- `public int[][] getSolutions()`[3]: returns a multidimensional `int` array with all of the possible configurations - its elements are TVL features identification numbers

2 https://projects.info.unamur.be/tvl/index.html#parser
3 This method may fail sometimes due to timeout (a high number of possible configurations makes this task time-consuming).

- where the first index represents a configuration as an `int` array, and the second represents the `int` identification value corresponding to a feature of that configuration;

- `public boolean isValid()`: returns a `boolean true` if the feature model inside TVL file is valid, otherwise returns `false`;
- `public boolean isSyntacticallyCorrect()`: returns a `boolean true` if the feature model inside TVL file is syntactically correct, otherwise returns `false`;
- `public boolean isCorrectlyTyped()`: returns a `boolean true` if the feature model inside TVL file has no type errors, otherwise returns `false`;

A `FeatureSymbol` is a class which represents a TVL feature and it contains all of the information that is related with that feature, namely:

- `public String getID()`: returns a `string` containing the feature name;
- `public int getMinCardinality()`: returns an `int` which represents the minimum cardinality (see Section 4.4.1);
- `public int getMaxCardinality()`: returns an `int` which represents the maximum cardinality (see Section 4.4.1);
- `public boolean isOptionnal()`[4]: returns a `boolean true` if the feature is optional, otherwise returns `false`;
- `public Map<String, FeatureSymbol> getChildrenFeatures()`: returns a `Map` object containing all the children features of a `FeatureSymbol` object, where each key is the child feature name and the value the corresponding `FeatureSymbol`.

Although `TVLParser` contains all of the information that is within TVL file, it does not have methods to interact with the TVL feature model. That task is left for `FeatureModel` class which has all of the selection/deselection methods and other operations that help to understand and to deal with a feature model. By instantiating this class and performing such operations, what is really happening is the continuous change of the TVL feature model structure according to the restrictions that a user may put on it. Any change to a certain feature is propagated to the rest of the model, which means that, for example, if some constraint declares that a given feature `A` cannot coexist with a given feature `B`, then, by selecting `A`, `B` gets automatically unselected, which in turn may cause the selection/deselection of other features, and so on. Some of the elements of this class are:

4 `FeatureSymbol isOptional` method has an error, once it was tested with optional features `Teleop` and `Random` of Listing 4.1 feature model and returned `false` for both.

- `FeatureModel(FeatureSymbol root)`: instantiates a new object of the `FeatureModel` class given a `FeatureSymbol` object (the same type of the value returned by `getRoot` in `TVLParser`); there is another constructor which instantiates a new object of the `FeatureModel` class directly with a given TVL file path, which avoids instantiating a `TVLParser` object (but validation methods such as `isValid` will not then be available);
- `public List<FDElement> getModel()`: returns a list containing `FDElement` objects, that is, the current state of the TVL feature model;
- `public boolean isSelectable(String featureName, boolean longName)`: returns a `boolean true` if the given feature is selectable or if it is already selected, otherwise returns `false`;
- `public boolean isUnSelectable(String featureName, boolean longName)`: returns a `boolean true` if the given feature is unselectable or if it is already deselected, otherwise returns `false`;
- `public boolean isIncluded(String featureName, boolean longName)`: returns a `boolean true` if the given feature is included, otherwise returns `false`;
- `public boolean isExcluded(String featureName, boolean longName)`: returns a `boolean true` if the given feature is excluded, otherwise returns `false`;
- `public boolean isUnassigned(String featureName, boolean longName)`: returns a `boolean true` if the given feature is unassigned, otherwise returns `false`;
- `public List<FDElement> include(String featureName, boolean longName)`: sets a given unassigned feature as included in the `FeatureModel` object, propagates consequent changes to the rest of the model and returns a list containing `FDElement` objects which their state has changed, excepting the actual included feature;
- `public List<FDElement> exclude(String featureName, boolean longName)`: sets a given unassigned feature as excluded in the `FeatureModel` object, propagates consequent changes to the rest of the model and returns a list containing `FDElement` objects which their state has changed, excepting the actual excluded feature;
- `public List<FDElement> unassign(String featureName, boolean longName)`: sets a given manually[5] included/excluded feature as unassigned in the

5 Automatically included/excluded features cannot be unassigned.

`FeatureModel` object, propagates consequent changes to the rest of the model and returns a list containing `FDElement` objects which their state has changed[6], excepting the actual unassigned feature.

Moreover, the `FDElement` class (abbreviation for feature diagram element) represents a `FeatureModel` feature and it provides information about its name and state (if it is selected, unselected or unassigned). Notice that both `FeatureSymbol` and `FDElement` represent the same concept, yet the later class is applicable to the manageable part of the TVL library (`FeatureModel`), which allows the user to interact with the feature model. Therefore, a `FeatureSymbol` does not have a state or any information related to the administration of a `FeatureModel` object and it is most commonly used alongside `TVLParser`.

6 `FDElement` objects which had their state recomputed but still the resulting state is the same as the previous, are also part of the returned `List<FDElement>` in `include`, `exclude` and `unassign` methods. This is most likely a TVL library bug.

# 5 HANDLING VARIABILITY IN ROS AND HAROS

As we have seen before, feature models can be used to describe the variability in a SPL. In this chapter we will describe a proposal to incorporate them in the ROS development process and in HAROS. A primary concern when thinking about possible solutions was to find something that was not applicable to HAROS only, but an approach that developers could easily reuse and apply to manage variability in ROS. An additional requirement was that the solution would need to be able to handle legacy applications, that were not developed properly as SPLs.

With that said and reminding the state of the art documented in the present dissertation, the chosen strategy was to somehow put TVL at ROS and HAROS disposal to solve the majority of the problems stated before. The following sections describe exactly the steps proposed to implement this solution.

To clarify and to stress the independence of the presented technique of the HAROS framework, the present chapter will be partitioned into two main parts:

- a first section where the technique is applied to ROS only, which documents:
  - the usefulness of having a TVL feature model;
  - the details to have in consideration when building a launch file.
- a second section where the technique is improved to be applicable for HAROS too, containing a new possible structure for its project configuration file;
- a third and final section describing what would be needed to build a ROS application and to assess it in HAROS with TVL.

## 5.1 HANDLING VARIABILITY IN ROS

First, a few very popular ROS packages (`turtlebot`[1], `lizi`[2], and `kobuki`[3]) were analysed as case studies in order to explore and experience how variability is managed in such applications. In this analysis the following problems were detected:

- the user has to know exactly which and how features need to be launched by combining and parameterizing different launch files;
- unacquainted users need to make a tremendous effort to understand launch files [8];
- the launch files syntax is not flexible enough to describe feature interactions;
- the complexity of the launch files leads to functional, syntactic and calculation errors.

The lack of a standardized pattern to implement launch files might be a cause of some of the previous problems. Concerning feature interaction, the usage of `$(eval)` expressions could alleviate the problem of conditional syntax inflexibility. Although these expressions are not being widely applied yet, they allow more flexibility in the implementation of a launch file, in comparison with `if` attributes of launch file tags, since they allow to launch ROS resources according to logic conditions. Nevertheless, HAROS is not currently able to handle `$(eval)` expressions.

The analysis of the packages and its launch files wound up in a conclusion: if launch files were built according to a feature model, then it would be possible to build and differentiate configurations with the help of that model. Moreover, if this feature model is backed by a logic solver, which is the case of TVL, a whole set of satisfiability operations would be immediately available. Despite not being fully functional, the TVL library provides an API with functions that allow us to do that task. Considering this, the following sections detail the several phases of proposal to link ROS and TVL, putting into practice the research of the state of the art.

### 5.1.1 *Describing the variability with a TVL file*

The whole process of building a ROS application should start by this file. As is widely known by developers, software should always be built according to a solid plan resulting from a preliminary and persistent study of the software domain. Only after this formal specification process, the software would start being built. That is what is intended to happen with this work: in essence, the TVL file is nothing more, nothing less than a formal specification which documents as much relations as possible between the existent features of a SPL.

1 http://wiki.ros.org/turtlebot
2 http://wiki.ros.org/lizi
3 http://wiki.ros.org/kobuki

Another particular aspect of formal specifications, of which a TVL feature model is one, is that a single domain can have multiple ways of being modeled. These may differ according to the adopted strategy, the depth of the study, the person that models it, but in the end each formal specification of the same application should be more a less identical. A TVL file was already built in Listing 4.1 for the `turtlerand` application, which serves as the main practical example for the current dissertation.

### 5.1.2 *Implementing a generic Launch File*

Previously in Chapter 2, a launch file was produced for each possible configuration of `turtlerand` application. Yet, the analysis of the prior mentioned packages, showed that all the three launch files could be grouped into a generic one.

```
<launch>
    <arg name="Turtlesim"/>
    <arg name="Turtlerand"/>
    <arg name="Teleop"/>
    <arg name="Random"/>

    <group if="$(arg Turtlesim)">
        <node pkg="turtlesim" type="turtlesim_node" name="Turtlesim"/>
    </group>

    <group if="$(arg Random)">
        <group if="$(arg Teleop)">
            <node pkg="turtlerand" type="multiplexer" name="Multiplexer">
                <remap from="cmd_vel" to="/turtle1/cmd_vel"/>
            </node>

            <node pkg="turtlesim" type="turtle_teleop_key" name="Teleop">
                <remap from="/turtle1/cmd_vel" to="cmd_vel_high"/>
            </node>

            <param name="vel" type="double" value="0.5" />

            <node pkg="turtlerand" type="random_controller" name="Random">
                <remap from="cmd_vel" to="cmd_vel_low"/>
            </node>
        </group>
    </group>

    <group if="$(arg Teleop)">
        <group unless="$(arg Random)">
            <node pkg="turtlesim" type="turtle_teleop_key" name="Teleop"/>
        </group>
```

```
32     </group>
33
34     <group if="$(arg Random)">
35         <group unless="$(arg Teleop)">
36             <param name="vel" type="double" value="0.5" />
37
38             <node pkg="turtlerand" type="random_controller" name="Random">
39                 <remap from="cmd_vel" to="/turtle1/cmd_vel"/>
40             </node>
41         </group>
42     </group>
43 </launch>
```

Listing 5.1: `turtlerand` generic launch file.

The first step to build such a generic launch file is to declare, inside the `launch` tag, an `arg` tag for each existing feature in the TVL file. Each `arg` tag must be declared with the attribute `name`, where its value is the exact string that is used to identify the corresponding feature in the TVL file (Listing 5.1, Lines 1 to 4). Note that only `arg` tags which are direct children of the `launch` tag will match TVL features (for instance, if an `arg` tag is declared as a child of Line 10 `group` tag, despite being valid, it will not match any feature in TVL).

What follows is the declaration of `node`, `param`, `remap` and similar tags which represent ROS resources to be launched. Each of these tags will be wrapped inside of the corresponding `group` tag which is linked to an `arg`, which in turn is linked to a TVL feature. Furthermore, it is recommended to declare tags representing ROS resources with the same `name` as the TVL feature that they are depending on. For instance, Line 16 `node` tag is the head element that is actually representing TVL `Teleop` feature, thus its `name` value is `Teleop` too. Nonetheless, Line 20 `param` tag corresponds to a ROS parameter which is not a feature but a complement to it, in this case a complement to feature `Random`. The `name` of this element is really simple because there are not any other parameters to be mistaken with, but it is recommended to somehow relate this `name` to the features which it is associated to.

The idea is to adopt these `arg` tags as boolean variables which, according to their value, will or will not launch the ROS resources which are depending on them in the launch file. For example, the `node` tag in Line 7 launches `Turtlesim` node if the `Turtlesim` `arg` is `true` (Line 6). If it is `false`, that instruction (Line 7) is not even parsed. The same happens with other ROS resources launched by `node` and `param` tags between Lines 12 and 24 which are only parsed if `Random` and `Teleop` `arg` tags are `true`.

*Eliminating generic launch file redundancy*

Any redundant information within the launch file must be removed since that improves the launch file quality. Listing 5.1 is a blatant example of a launch file that can easily be simplified in multiple ways, resulting in the launch file of Listing 5.2.

The first adaption to be done is the extinction of duplicated nodes, meaning those that are redundant. For instance, `node` tags `Teleop` and `Random` are declared twice thanks to the existence of `Multiplexer` `node` tag which must be launched every time booth those nodes are launched too. Yet, whether `Teleop` and `Random` are launched together or not, the only change they suffer lies in `remap` tags. The solution for the `Teleop` `node` tag is to:

- remove the first occurrence of `Teleop` `node` tag (Listing 5.1, Lines 16 to 18);
- and add to the other `Teleop` `node` tag (Line 30) the prior removed `remap` tag (Line 17) with an additional `if` attribute depending on the `Random` `arg` tag value (Listing 5.2, Line 17).

The solution for the `Random` `node` tag is to:

- remove the first occurrence of `vel` `param` tag (which is a complement of `Random` feature only; Listing 5.1, Line 20);
- remove the first occurrence of `Random` `tag` (Lines 22 to 24);
- add to the other `Random` `node` tag (Line 38) the prior removed `remap` tag (Line 23) with an additional `if` attribute depending on the `Teleop` `arg` tag value (Listing 5.2, Line 23);
- and add to the existing `remap` tag (Line 39) an `unless`[4] attribute depending on the `Teleop` `arg` tag value (Listing 5.2, Line 25).

More simplifications can be done to `group` tags which are wrapping launch file elements: some of these tags (for example, `group` tag wrapping `Turtlesim` `node` tag in Line 6) can be replaced by an `if` attribute placed in the tags they are wrapping, with the same value that it had in the respective `group` tag (Listing 5.2, Line 6).

```
0 <launch>
1     <arg name="Turtlesim"/>
2     <arg name="Turtlerand"/>
3     <arg name="Teleop"/>
4     <arg name="Random"/>
5
```

4 In opposite to `if`, the `unless` attribute causes the corresponding XML element to be parsed if the argument is `false`.

```
6      <node pkg="turtlesim" type="turtlesim_node" name="Turtlesim" if="$(arg
           Turtlesim)"/>
7
8      <group if="$(arg Random)">
9          <group if="$(arg Teleop)">
10             <node pkg="turtlerand" type="multiplexer" name="Multiplexer">
11                 <remap from="cmd_vel" to="/turtle1/cmd_vel"/>
12             </node>
13         </group>
14     </group>
15
16     <node pkg="turtlesim" type="turtle_teleop_key" name="Teleop" if="$(arg Teleop
           )">
17         <remap from="/turtle1/cmd_vel" to="cmd_vel_high" if="$(arg Random)"/>

18     </node>
19
20     <param name="vel" type="double" value="0.5" if="$(arg Random)"/>
21
22     <node pkg="turtlerand" type="random_controller" name="Random" if="$(arg
           Random)">
23         <remap from="cmd_vel" to="cmd_vel_low" if="$(arg Teleop)"/>
24
25         <remap from="cmd_vel" to="/turtle1/cmd_vel" unless="$(arg Teleop)"/>

26     </node>
27 </launch>
```

Listing 5.2: `turtlerand` generic launch file simplified (in relation to Listing 5.1).

### *Reducing the complexity of the generic launch file*

Sometimes, `if` statements lead to a nightmare of nested expressions, something that happens quite frequently in the "traditional" software development process too. In the particular case of Listing 5.2, the generic launch file became really simple, but let us suppose that `turtlerand` application has another node which `Multiplexer` node is depending on. In that case, those `remaps` that were exclusively depending on `Teleop node` (Listing 5.2, Lines 23 and 25) would have to depend on this new `node` too, which is not possible[5]. Therefore, one would need to step back to the Listing 5.1 generic launch file approach, where `Multiplexer node` would not be under two, but three conditional `group` tags. Fortunately, there is another pattern to encode generic launch files, which can help reduce such conditional complexity.

5 Only one `arg` tag value can be assigned to an `if` attribute

This pattern was inspired by launch files of much larger packages, as is the case of `turtlebot` and `lizi`.

The strategy consists in assuming each ROS resource as a feature, as long as these are represented by the following launch file tags: `node`, `param`, `machine`, `rosparam`, `test`, `include`. In the case of `turtlerand` application, its TVL file (Listing 4.1) needs to be redefined, as well as the generic launch file (Listing 5.2).

Starting by the TVL file, two new features need to be added: `Vel`, which will be a mandatory sub feature of `Random`, and `Multiplexer`, an optional sub feature of `Turtlesim`. `Multiplexer` will only be available if both `Random` and `Teleop` features are too - this condition will be encoded by two cross tree constraints. Listing 5.3 shows the resulting TVL file.

```
root Turtlerand {
    group allOf {
        Turtlesim
    }

    Multiplexer -> (Teleop and Random);
    !Multiplexer -> !(Teleop and Random);
}

Turtlesim {
    group allOf {
        opt Teleop,
        opt Random,
        opt Multiplexer
    }
}

Random {
    group allOf {
        Vel
    }
}
```

Listing 5.3: `turtlerand` alternative TVL feature model.

In what concerns to the generic launch file, new `Vel` and `Multiplexer` `arg` tags must be added, as well as an `if` attribute to the respective `node` and `param` tags , where their value is conditioned by the corresponding `arg` tag value. The `group` tags that were acting like conditional wrappers around `Multiplexer` `node` are removed. At last, `if` and `unless` `remap` tags values must be reassigned to depend on the `Multiplexer` `arg` tag value, and not on the `Teleop` `arg` tag, like they were. Listing 5.4 shows the resulting generic launch file.

```
<launch>
```

```
    <arg name="Turtlesim"/>
    <arg name="Turtlerand"/>
    <arg name="Teleop"/>
    <arg name="Random"/>
    <arg name="Vel"/>
    <arg name="Multiplexer"/>

    <node pkg="turtlesim" type="turtlesim_node" name="Turtlesim" if="$(arg Turtlesim)"/>

    <node pkg="turtlerand" type="multiplexer" name="Multiplexer" if="$(arg Multiplexer)">
                <remap from="cmd_vel" to="/turtle1/cmd_vel"/>
    </node>

    <node pkg="turtlesim" type="turtle_teleop_key" name="Teleop" if="$(arg Teleop)">
        <remap from="/turtle1/cmd_vel" to="cmd_vel_high" if="$(arg Random)"/>

    </node>

    <param name="Vel" type="double" value="0.5" if="$(arg Vel)"/>

    <node pkg="turtlerand" type="random_controller" name="Random" if="$(arg Random)">
        <remap from="cmd_vel" to="cmd_vel_low" if="$(arg Multiplexer)"/>

        <remap from="cmd_vel" to="/turtle1/cmd_vel" unless="$(arg Multiplexer)"/>

    </node>
</launch>
```

Listing 5.4: `turtlerand` alternative generic launch file.

*Declaring `arg` tags representing abstract features*

Some `arg` tags which correspond to TVL abstract features may not directly represent any ROS resource(s) at all. That situation occurs quite frequently in applications larger than `turtlerand`. Yet, this one also has an abstract feature which does not match any concrete ROS resource - the `Turtlerand` feature. This feature represents the root element of the TVL feature model, standing for the all `turtlerand` application itself. Still, regardless of representing or not any actual ROS resource(s), for orthogonality reasons it should be declared as an `arg` in the generic launch file.

*Distinguishing `arg` tags that are not features*

Suppose one wants to declare an `arg` tag which is not a feature in the TVL file. If we declare it right below the `launch` tag it would be wrongly considered a TVL feature. To prevent that from happening, the user should declare a `group` tag wrapping every non-feature tag in it, as exemplified in Listing 5.5, so that the `arg NotAFeature` is not confounded with a feature.

```
<launch>
    <arg name="Turtlesim"/>
    <arg name="Turtlerand"/>
    <arg name="Teleop"/>
    <arg name="Random"/>

    <group>
        <arg name="NotAFeature"/>

        <node pkg="turtlesim" type="turtlesim_node" name="turtlesim_node" if="$(arg Turtlesim)"/>

...

                <remap from="cmd_vel" to="/turtle1/cmd_vel" unless="$(arg Teleop)"/>
        </node>
    </group>
</launch>
```

Listing 5.5: `turtlerand` generic launch file with `<group>` tag technique.

*Summary*

All in all, Listing 5.4 strategy removes even more complexity from the generic launch file comparing to Listings 5.1 and 5.2, but the feature model may become overloaded with minor features that are rather easily seen as complements of a feature. It is the user choice to decide which approach he prefers to adopt:

- in one extreme, all feature interactions can be declared as conditions within the launch file, rendering the TVL useless;
- in another extreme, all feature interactions can be declared within the TVL file, meaning everything will be considered a feature, even minor ROS resources;
- or a combination of both, where most of the conditions are declared within the TVL file and some in the launch file.

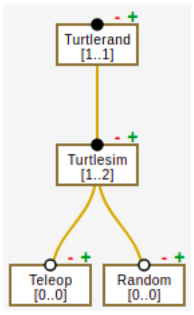


Figure 17: Mock of an interactive TVL feature diagram.

### 5.1.3 *Towards a variability aware ROS launcher*

So far, a generic launch file pattern was established in order to make launch files connectable to TVL feature models. Given this, it would be very useful to have a tool that checks the validity of the TVL file, the validity of the generic launch file and that connects both in order to ease the process of analysing and deploying a specific robot configuration.

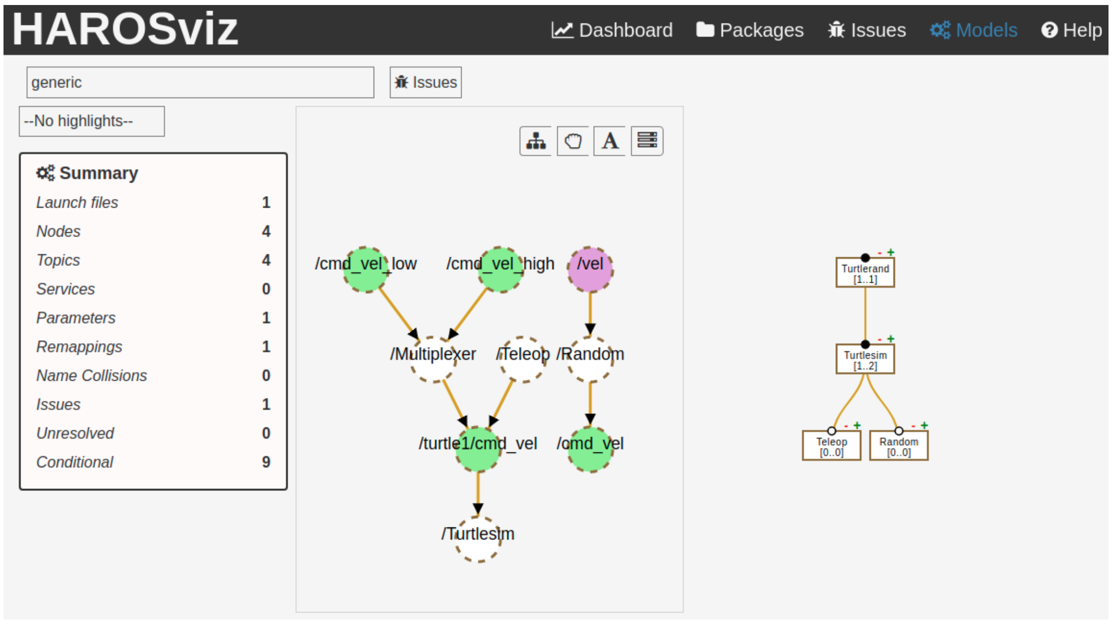
Such tool should start by displaying a feature diagram corresponding to the provided TVL file. The user could then interact with the diagram by selecting or deselecting features, actions that are actually changing the state of the TVL feature model. Some scalability concerns were raised about feature diagrams, nevertheless some simple interface tricks can make it capable of supporting a large number of features, namely the option to collapse children nodes to their parents position (which reduces the feature diagram size). Additionally, the tool should be able to generate every possible configuration according to the selected model features.

In essence, supposing that a ROS user wants to launch multiple ROS resources of a launch file, with this approach he does not need to understand the conditional `if` attributes attached to that elements in the file, but rather look and interact with the TVL feature diagram (like the mock one in Figure 17, where the plus and minus symbols allow for the selection and deselection of features). Therefore, although variability still exists it is now much easier to manage.

## 5.2 HANDLING VARIABILITY IN HAROS

In what concerns to variability arising from the direct use of launch files, a solution was already presented. Yet, these ROS issues are also reflected in HAROS, which lacks a method to differentiate ROS resources according to the features they represent. In another words:

- a project file containing launch files that launch a large quantity of ROS resources (nodes, services, parameters, etc.), will originate an unintelligible architectural model in the HAROS visualiser `Models` tab;



**Summary**

| | |
|---|---|
| Launch files | 1 |
| Nodes | 4 |
| Topics | 4 |
| Services | 0 |
| Parameters | 1 |
| Remappings | 1 |
| Name Collisions | 0 |
| Issues | 1 |
| Unresolved | 0 |
| Conditional | 9 |

Figure 18: Proposed interface for the new HAROS visualiser with an interactive TVL feature diagram.

- the same happens with the `Issues` tab, where the issues reported by the analysis plugins are all shown at once, not considering different configurations;
- the conditional relations between ROS resources are not perceivable.

### 5.2.1 *HAROS-TVL server*

Concerning the problems in the HAROS reporting stage (implemented by the HAROS visualizer), we propose adopting an interface similar to the one on Figure 17, to ease the selection of different configurations. This requires replacing the original Python server that is used in the HAROS visualiser (to read and parse models and issues from JSON files) with a new so-called HAROS-TVL server, which is capable of processing and interacting with a TVL feature model. All of the design and functional logic of the HAROS former server would also need to be implemented in the latter.

By interacting with a TVL feature model, the HAROS visualiser would be able to show in the computation graph, under `Models` tab (see Figure 18), only the ROS resources which are related to the selected features (likewise for the issues under the `Issues` tab). Thus, the computation graph would have to be automatically updated every time the HAROS user selects/deselects a feature in the feature diagram.

### 5.2.2 *Project configuration file*

The HAROS project configuration file format also needed some refinements too. Our proposal is to use variability aware generic launch files, as proposed in the previous section, and pair them with the TVL file that characterizes the respective feature model. Even though the previous project file format did not consider the latter, the YAML syntax and the way that the project file format was initially structured, makes it quite easy to attach the TVL file path to the launch file path.

Actually, we could already use generic launch files with standard HAROS. For example, we could define a project configuration file for the generic launch file of our running example (Listing 5.2), as shown in Listing 5.6. The main problem is the need to replace configuration based hints by node based hints (Listing 5.6, starting at Line 6), so that they could be applied to any configuration encoded in the generic launch file.

```
 0 project: SAFER
 1 packages: ["turtlerand"]
 2 configurations:
 3     generic:
 4         launch:
 5             - turtlerand/launch/generic.launch
 6 nodes:
 7     turtlesim/turtlesim_node:
 8         subscribe:
 9             /turtle1/cmd_vel: geometry_msgs/Twist
10     turtlesim/turtle_teleop_key:
11         advertise:
12             /turtle1/cmd_vel: geometry_msgs/Twist
13 rules:
14     parameters_readers:
15         name: Parameters readers
16         description: "One or more parameters are being read."
17         tags:
18             - custom
19         query: "for n in <configs/nodes>
20                 where len(n.reads) > 0
21                 return n"
```

Listing 5.6: Generic `turtlerand` project configuration file.

So far no change is needed to the project file format, and HAROS would be able to read this file without any difficulty. Nevertheless, so support the the new HAROS-TVL server, the project configuration needs to specify the TVL file location somewhere. Our proposal is to change the type of `launch` key values: instead of strings with locations to the launch files, these would be objects containing two key value pairs. The first element of each object would be lf key (abbreviation for launch file) where the value would be the launch file path and the second element, if the user chooses to introduce it, would be tvl key where the value would be the TVL file path. Moreover, each project should now be restricted to one configuration only. In Listing 5.6 the generic and launch keys are no longer needed for that reason and the object containing the file paths should be unique. Listing 5.7 shows the proposal for the new configurations section, applied to our running example.

```
project: SAFER
packages: ["turtlerand"]
configurations:
    {lf: turtlerand/launch/generic.launch, tvl: turtlerand/launch/generic.tvl}
...
```

Listing 5.7: HAROS-TVL turtlerand project configuration file.

## 5.3 SUMMARY

In short, with our proposal for a variability aware HAROS, the user would follow the following steps to analyze a project:

1. define a TVL feature model according to the preliminary study of the domain;
2. define a generic launch file taking that TVL feature model in consideration;
    a) declare for each TVL feature an arg tag;
        - launch file arguments must have the exact same name as TVL features;
        - launch files arguments and TVL features must match.
    b) declare inside the launch file the ROS resources to be launched;
        - $(eval) expressions are not allowed.
    c) attach if/unless attributes to the launch file elements according to the TVL features they are depending on;
3. define a project configuration file;
4. launch the new HAROS visualizer supported by the HAROS-TVL server;
5. navigate to the models tab and select/deselect features, to restrict the architectural model and issues list to the desired configurations.

# 6

# HAROS-TVL: A VARIABILITY AWARE HAROS EXTENSION

To implement the proposed extensions to HAROS, and considering that HAROS was developed in Python (analyser) and JavaScript (visualiser), it would make sense to develop any additional functionality in one of these programming languages. However, the TVL library is developed in Java, which raises some conflicts. Within the available options, of which one was to develop a new independent Java or Python server, it was decided that the best one was to develop a NodeJS server that was able to process Java code. That would allow this server to be used independently (for ROS developers not using HAROS) and it would also allow an easy integration with the HAROS visualiser.

The present chapter documents the development process of HAROS-TVL. The next sections detail the execution of Java code under a JavaScript environment, the conversion of a feature diagram into a JSON object (as required by *D3* library used in the HAROS model visualization), the development of the HAROS-TVL server, and finally its integration in the HAROS visualiser, resulting in the HAROS-TVL extension.

## 6.1 EXECUTING JAVA CODE WITHIN A JAVASCRIPT ENVIRONMENT

Since the TVL library is implemented in Java and both TVL and HAROS-TVL servers would be developed in JavaScript, two options were considered: the first was to develop a Java server which was able to communicate with the TVL server and the other was to find a way of running Java code inside JavaScript. After some research, a package named *java* [1] was found under *Node Package Manager* (npm) [2] which allows its users to connect to and use Java APIs. Thereafter, it was not only possible to benefit from Java collections but also to import other Java libraries such has `TVLParser`. The only additional requirement was to add the `Sync` suffix at the end of each Java method.

1 npm *java* package official website - https://www.npmjs.com/package/java
2 npm official website - https://www.npmjs.com/

## 6.2 CONVERTING THE TVL FEATURE DIAGRAM INTO A D3 JSON STRUCTURE

The TVL library does not provide any graphical interface, so `FeatureModel` operations are done by directly calling the preferred method and not by interacting with any button or visual element. Whereas some ROS users are software developers, others are not, and interacting with an intuitive interface is much more feasible than understanding and typing Java code. Therefore, some research resulted in finding the *D3* JavaScript library [3] which eases the process of showing data in an interactive and dynamic way by using HTML, CSS and SVG. This is also the library that the HAROS visualiser uses to draw the computation graph.

In order to make an interactive TVL feature model with this library it was necessary to draw its feature diagram, which in essence is a graph tree drawn accordingly to the `FeatureModel` object state. Simply put, this process is done by providing *D3* a specific JSON structure which is built by traversing the `TVLParser` object features. This JSON also has the structure of a graph tree, so the whole JSON is an object which represents the root feature where one of its attributes will be an array containing, multiple objects - its children features, which will may be have their children too, and so on. The object that represents a feature in that JSON has the following attributes:

- `name`: identifies the feature;
- `parent`: identifies the parent feature of the current feature;
- `selected`: stores the state of the feature, meaning if it is selected (`true`), unselected (`false`) or unassigned (`null`, its default state);
- `minCard`: stores the minimum cardinality value (see Section 4.4.1);
- `maxCard`: stores the maximum cardinality value (see Section 4.4.1);
- `opt`: contains information whether the feature is either optional (`true`) or mandatory (`false`);
- `children`: an array which contains other objects with the same structure of the one here described.

## 6.3 IMPLEMENTING THE HAROS-TVL SERVER

The following sections will summarily document the implementation of the HAROS-TVL server.

3 *D3* JavaScript library official website - `https://d3js.org/`

### 6.3.1 *Initialization*

Firstly, a folder is specified with static content which the server will provide to the web browser to build the web page. Then, the given TVL file and launch file location paths are stored in different variables. To be used, these files must be submitted to a validation process in which:

- the existence of those files is assessed;
- the syntax and typing of the TVL file must be correct (this validation is done by instantiating `TVLParser`, which has proper methods to do it; after that successful validation a `FeatureModel` object is also instantiated);
- the number of TVL model features is computed, as well as the number of all possible configurations;
- TVL model features must match the corresponding launch file `arg` tags.

Once that is done, the server can turn the TVL file into a JSON with the help of `jsonize` function. This new JSON is then sent to the browser and the TVL file that was once completely textual is now a viewable and manageable diagram. By clicking in certain features buttons or refreshing the web page, different server handlers are triggered.

### 6.3.2 *Selecting or deselecting a feature*

Selecting or deselecting a feature, also seen as including or excluding it, respectively, are probably the main and most frequent operations performed by this server. Due to both having the same behaviour, their logic is under the same handler function - the one that handles routes `/include` and `/exclude`. Every time this handler is triggered, the server must:

1. check whether that feature is unassigned or not; in case it is not, nothing will happen and the following steps are dismissed;
2. call `FeatureModel` `include` or `exclude` methods, depending whether the route is `/include` or `/exclude`, respectively;
3. check if the feature is either includable (selectable) or excludable (unselectable), respectively;
4. apply the new changes to the data that is displayed afterwards by the computation graph;

5. retrieve the state of each `FeatureModel` object feature with the help of `getModel` method, and store it inside `object` variable (of which each feature is an attribute where the key is its name and the value its state in the `FeatureModel` object);
6. answer to the browser request with that `object`.

### 6.3.3 *Refreshing the server*

The user may want to reset/refresh the feature model sometimes, and the HAROS-TVL server allows him to do so by handling that request which is depicted with the / route. This handler is also triggered every time a user opens the HAROS-TVL server web page. The only task that this handler function does is to instantiate `FeatureModel` again, an action that resets features states.

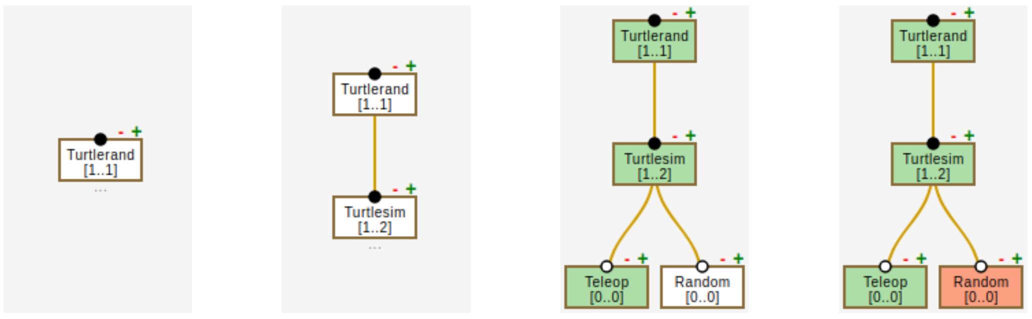
### 6.3.4 *The server web page*

As soon as HAROS-TVL web page is launched, it requests to the HAROS-TVL server the JSON which contains the graph structure. The server includes that JSON in the response and when the browser receives it, a new object is created with that information.

Upon the call of `update` function that takes this newly created object as an argument, the browser creates a node for each feature which is pictured as a rectangle with one plus and other minus signs buttons. Should that feature has children, that rectangle will be clickable to either unveil or hide its children features. Three grey dots below a feature rectangle mean that it has children, otherwise (no children) there will be no dots.

As previously mentioned, a feature has three states: selected, unselected and unassigned. The feature diagram also distinguishes those three states with red colour for selected features, green for the unselected features and white for the unassigned ones (the default colour of all features).

Additionally, each rectangle has information about the cardinality of that feature, as explained in Section 4.4.1, and a filled or empty circle (according to Section 4.3.1). Cardinality and optionality are two types of information that can overlap or even contradict each other sometimes, as is the case of a feature that has `[2..2]` cardinality and both children are optional. With respect to cardinality, both children must be chosen, yet with respect to optionality, they can both become excluded. To untie such situations, the user must assume that optionality prevails over cardinality (see Section 4.4.1 footnote), meaning that, in the previous example, each feature can either be included or excluded and that they are not automatically included alongside their parent as a `[x..x]` cardinality type always leads to.



(a) Feature diagram initial state and appearance.

(b) Expanding `Turtlerand` to unfold its children.

(c) Expanding `Turtlesim` and selecting `Teleop`.

(d) Deselecting unassigned `Random` feature.

Figure 19: Evolution of a feature model until a configuration is selected.

In order for the previous handlers to be activated, some triggers must be pressed. These are the prior mentioned buttons which exist in the web page that displays the feature model, which in turn are tied to browser side functions that throw the corresponding requests to the actual TVL server. For instance:

- clicking in a feature rectangle will cause its children features to either appear or disappear; `toggle` function changes the graph object;
- clicking in a feature plus sign will request the HAROS-TVL server to include that feature; `changeFm` function sends the request to the HAROS-TVL server and the `changeSelections`, `changeSelectionsRec` and `goToRoot` functions manage to update graph object with the new `FeatureModel` state that came in the server response;
- clicking in a feature minus sign will request the TVL server to exclude that feature; the logic is the same as the previous item.

Figure 19 illustrates the evolution of a feature model until it reaches a configuration, as well as all the buttons and visual details of a feature.

### 6.3.5 *Generating configurations*

The previous chapter as shown how variability can be managed in ROS and HAROS with the help of TVL and generic launch files. Since TVL allows us to compute all possible configurations/products of a certain TVL feature model, HAROS-TVL can provide an additional functionality to its users.

By interacting with the feature model, the user may come to a point where he wants to know which configurations exist according to the selected/unselected/unassigned features.

In fact, reaching a specific configuration and knowing which features must be launched and which must not, may be one of the reasons some users might want to use HAROS-TVL. Unfortunately, using the TVL library it is not possible to infer all possible configurations according to the `FeatureModel` object state, only all the configurations of the feature model.

Supposing that the TVL library allows to perform such operation, the HAROS-TVL server could be easily modified to generate, from the generic launch file, a new launch file for each possible configuration according to the `FeatureModel` state. To do this, each new launch file would need to have the same `arg` tags (of those corresponding to features) of the generic launch file, but each one would be declared with `true` or `false` values. Every new launch file would then include the generic launch file and pass those `arg` tags values to the generic one. That way, executing `roslaunch` with one of the new launch files would launch a specific configuration.

To illustrate this functionality, we implemented it for the situation where the user wants all possible configurations of the feature model. Listing 6.1 portrays one of the generated launch files, corresponding to a possible configuration for `turtlerand` application (where `Turtlerand`, `Turtlesim` and `Teleop` features are included, and `Random` feature is excluded).

```
<launch>
    <arg name="Turtlerand" value="true"/>
    <arg name="Turtlesim" value="true"/>
    <arg name="Random" value="false"/>
    <arg name="Teleop" value="true"/>
    <include file="$(find turtlerand)/launch/generic.launch">
        <arg name="Turtlerand" value="$(arg Turtlerand)"/>
        <arg name="Turtlesim" value="$(arg Turtlesim)"/>
        <arg name="Random" value="$(arg Random)"/>
        <arg name="Teleop" value="$(arg Teleop)"/>
    </include>
</launch>
```

Listing 6.1: `turtlerand` launch file corresponding to the configuration selected in Figure 19.

## 6.4 IMPLEMENTING THE HAROS-TVL EXTENSION

To incorporate the HAROS-TVL server in the implementation of the new HAROS-TVL extension the following steps are required:

- as suggested in Section 5.2.2, the path to the location of the TVL file should be placed alongside the path to the location of the corresponding generic launch file, so that the HAROS-TVL server can retrieve those paths from the project file, as well as the project name;

- the HAROS-TVL server needs to have access to the data generated by HAROS analyser, in another words, it needs to access the same data that the former HAROS visualiser server used to access (which is commonly stored in the directory `<home directory>/.haros/viz/data`);
- new variables need to be created to store the content of the `configurations.json` and `generic.json` files, which have all of the information of the computation graph and runtime issues, respectively;
- every time a user selects/deselects a feature, `configurations.json` file must be updated, and the computation graph must be redrawn (this item will be covered in depth in the next section);
- every time a user selects/deselects a feature, the runtime issues data must be updated, and the list of issues under `Issues` tab must be rewritten;
- the HAROS web page (particularly, the `Models` tab) source code needs to be adjusted to include the TVL feature model diagram.

### 6.4.1 *Computation graph file*

When the `haros analyse` command is executed, a `configurations.json` file is generated, containing the application computation graph data, alongside other files corresponding to the given project. For example, by running HAROS with Listing 5.7 project configuration file, the generated `configurations.json` will be stored in the `/.haros/viz/data/SAFER` folder. File `configurations.json` has all of the data in which HAROS visualiser relies on to draw the computation graph. For example, every data directly related to an existing node will be stored inside the `nodes` property, if a topic is connected to a node, that information will be held inside property `links`, a query will be an object of `queries` property, and so on. In order to change the computation graph at the same time the user changes the feature diagram, the HAROS-TVL server must update that file and message the browser to update the computation graph.

It is important to mention that selecting or deselecting features introduces different changes in `configurations.json`. Nevertheless, it does not matter if the user selects/deselects a feature since the TVL library can propagate than action and select/deselect other features on its own. What this means is that, either by selecting or deselecting a feature, the HAROS-TVL server not only changes information in `configurations.json` with respect to that feature, but to all of those who are included or excluded as a consequence of that action.

```
...
    "nodes":[
266       {
...
287         "name":"/Multiplexer",
288         "type":"turtlerand/multiplexer",
289         "writes":[],
290         "clients":[],
291         "conditions":[
292           {
293             "condition":"$(arg Random)",
294             "statement":"if",
295             "location":{
296               "function":null,
297               "package":"turtlerand",
298               "column":3,
299               "file":"launch/generic.launch",
300               "line":10,
301               "class":null
302             }
303           },
304           {
305             "condition":"$(arg Teleop)",
306             "statement":"unless",
307             "location":{
308               "function":null,
309               "package":"turtlerand",
310               "column":5,
311               "file":"launch/generic.launch",
312               "line":11,
313               "class":null
314             }
315           }
316         ]
317       },
...
```

Listing 6.2: Snippet of `configurations.json` file.

So, from the point of view of `configurations.json`, selecting/deselecting features where an element is affected by an `if/unless` statement, respectively, implies removing every object from the `condition` property that references those features. This applies to any element under properties `nodes`, `topics`, `parameters` or `services`. For instance, Listing 6.2 is a snippet of `configurations.json` where we can see information about an object of `nodes` property, which is related to `Multiplexer` ROS resource (according to Line 287). If feature `Random` is selected, then `Multiplexer` will no longer depend on it to be executed, and therefore it is irrelevant to consider it a condition for `Multiplexer` - as a result of that, the first object of `conditions` is removed (Lines 292 to 303). If feature `Teleop` is deselected, then the other existing condition is also fulfilled (remind that `Multiplexer` is executed if `arg Teleop` is false, that is, if `Teleop` feature is not enabled - see Line 306). The resultant file is depicted in Listing 6.3 and this also means that, in the computation graph, the node that represents `Multiplexer` has its dashed line redrawn as a thick line.

```
264 ...
265     "nodes":[
266       {
...
287         "name":"/Multiplexer",
288         "type":"turtlerand/multiplexer",
289         "writes":[],
290         "clients":[],
291         "conditions":[]
292       },
...
```

Listing 6.3: `configurations.json` file (Listing 6.2) after selecting `Random` and deselecting `Teleop` features.

In what comes to selected/deselected features concerning elements affected by `unless/if` statements, respectively, that process is much more burdensome, as it implies removing any information related to that feature. In particular, this case requires:

1. removing from properties `nodes`, `topics`, `parameters` and `services` each object where its `condition` property references at least one selected/deselected feature, in the array `conditions`; identifications of the removed elements must be recorded in order to remove their data from other elements too;

2. based on the elements previously removed, clean the remaining data that may have been left behind in other elements which were somehow connected to them; this task is needed because, when the browser is redrawing the computation graph, it cannot draw, for example, a link from a topic to a node, where the topic exists but the node was removed.

For example, in Listing 6.2, if `arg Random` is deselected, then it does not matter if `arg Teleop` is either `true` or `false`, because a condition is already failing for `Multiplexer` to be executed. Now, in spite of removing the condition that fails, the whole object of `nodes` that is related to `Multiplexer` is removed as well as other information in the file that is related to this feature (not visible in Listing 6.2). The same happens if `arg Teleop` is selected.

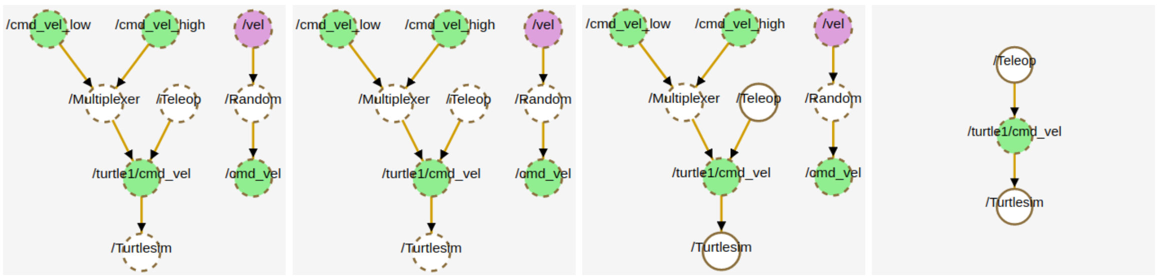
### 6.4.2 *Runtime issues files*

In the former HAROS version, each configuration has a dedicated file which stores all its issues. Now that HAROS-TVL is not intended to accept multiple configurations but a single launch file where all of them can be inferred, there will only exist a single runtime issues file. For the running example, the runtime issues file stored within `/.haros/viz/data/SAFER/compliance/runtime` folder (for the `SAFER` project) contains all of the runtime issues data that appears under `Issues` tab.

Runtime issues are often related to specific computation graph elements and therefore they are also related to features. Nonetheless, HAROS does not establish a connection between both, so, for each runtime issue JSON object, there is not any information about the features it is related too, which makes it impossible to apply any type of filtering. This could be easily solved by providing for each runtime issue object a `conditions` array containing all of the features that the issue is depending on, similarly to what happens with `configurations.json` file object. The process of removing those issues would be equal to what happens in `configurations.json`.

To make this functionality work, the content of the runtime issues file is being temporarily manually generated, until HAROS-TVL starts applying conditions to runtime issues.

### 6.4.3 *Variational computation graph*



(a) Computation graph initial state and appearance.

(b) Computation graph state remains unchanged.

(c) `Teleop` and `Turtlesim` nodes become thick.

(d) ROS resources depending on `Random` feature disappear.

Figure 20: Different appearances of the computation graph according to the respective different states of the Figure 19 feature diagram.

A computation graph and a visual feature model are depicted when the user launches the HAROS-TVL server, opens a browser and navigates to the `Models` tab. To distinguish elements which presence is not yet certain according to the selected / deselected features, these will be shown as dashed. For example, when launching HAROS-TVL with a generic launch file as conceived in Section 5.1.2, it is expected the majority of the computation graph elements to be dashed (since features on which they are depending on are unassigned).Then, by selecting or deselecting TVL model features, these dashed nodes must either become thick or disappear, respectively. That way, the user can become aware of which ROS resources are depending on which features, and have a better understanding of what is happening in the ROS application. Figure 20 shows how the computation graph unfolds as the HAROS-TVL user selects/deselects features in the TVL feature model diagram.

# 7 HAROS-TVL EVALUATION

Although HAROS-TVL approach seems to be a promising technique to handle variability in small applications such as `turtlerand`, it is necessary to evaluate its effectiveness in larger ROS applications. In particular, HAROS-TVL requires a specific structure for each of the setup files (generic launch file, project file and TVL file) and existing applications were not built considering them. As far as one can tell from the state of the art, the TVL language is not used in the ROS community and so the existing applications do not even have a TVL file to describe them. To test the proposed technique with existing applications it is thus necessary to write such TVL files and write generic launch files that fit the HAROS-TVL required format (apart from the project configuration file required by HAROS).

In this chapter we will evaluate the HAROS-TVL extension with small and larger ROS application examples, namely some which are widely known by the ROS community. For each one, a TVL file, a generic launch file, and a project configuration file will be created/adapted to be later submitted and executed with HAROS-TVL. The most relevant conclusions about using and interacting with HAROS-TVL visualiser using such applications will then be presented.

## 7.1 APPLYING HAROS-TVL TO `TURTLERAND`

As mentioned before, `turtlerand` is not a sufficiently large application to properly evaluate HAROS-TVL. Yet, it is a valid ROS application and substantial readability improvements were observed when changing from HAROS to HAROS-TVL. Bearing this in mind, it was decided even those small improvements would be valuable to showcase HAROS-TVL qualities.

### 7.1.1 *Defining a TVL file*

So, considering `turtlerand` application as it was described in Chapter 2, one abstract and three concrete features stand out from the available domain. `Turtlerand` is the abstract feature which represents the application itself and Turtlesim, Teleop and Random are the other three concrete features. Listing 4.1 shows the final TVL file for this application.

### 7.1.2 *Defining a generic launch file*

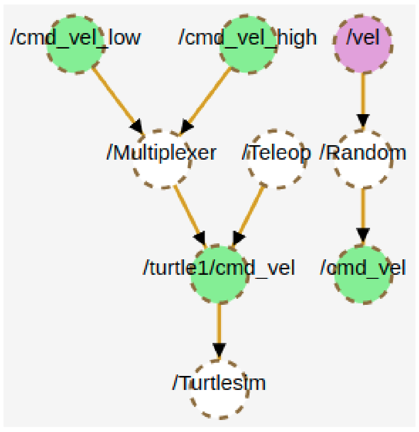
The generic launch file is easily inferred from the previous TVL feature model. One only has to identify which features are related to a certain ROS resource and put it under the corresponding conditions. Thus, the arg tags will be declared with name attributes Turtlerand, Turtlesim, Teleop and Random. The definition of conditions is not as straightforward as declaring the arg tags, but still easy. The turtlerand generic launch file can be seen in Listing 5.2.

### 7.1.3 *Defining a project configuration file*

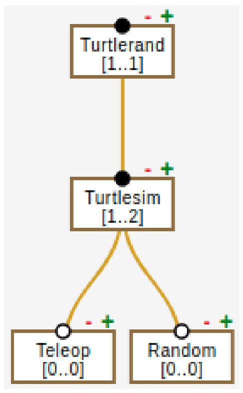
Defining the project configuration file is also simple. The generic launch file path is simply declared alongside the TVL file path, as explained before. Listing 7.1 is the resulting project configuration file.

```
project: SAFER
packages: ["turtlerand"]
configurations:
    {lf: turtlerand/launch/generic.launch, tvl: turtlerand/launch/generic.tvl}
nodes:
    turtlesim/turtlesim_node:
        subscribe:
            /turtle1/cmd_vel: geometry_msgs/Twist
    turtlesim/turtle_teleop_key:
        advertise:
            /turtle1/cmd_vel: geometry_msgs/Twist
rules:
    parameters_readers:
        name: Parameters readers
        description: "One or more parameters are being read."
        tags:
            - custom
        query: "for n in <configs/nodes>
                where len(n.reads) > 0
                return n"
```

Listing 7.1: turtlerand project configuration file.



(a) Computation graph.



(b) Feature diagram.

Figure 21: Initial state of `Models` tab for the `turtlerand` application.

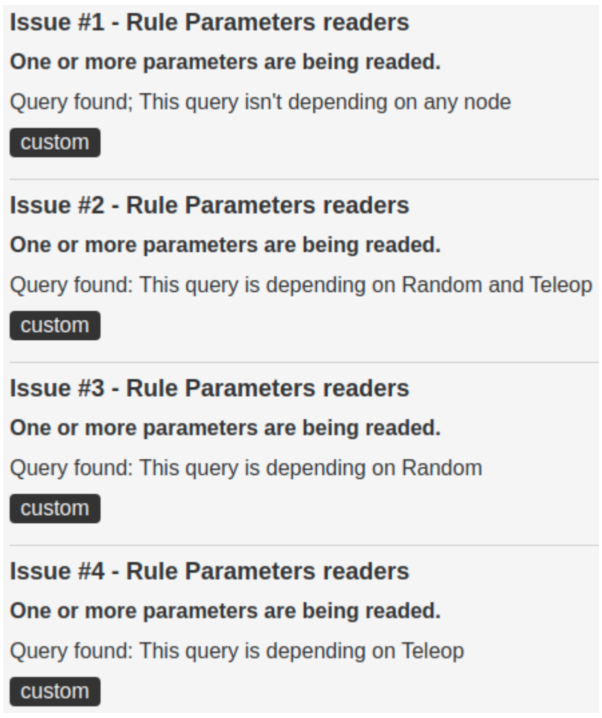
### 7.1.4 *Evaluation*

Figure 21 depicts the HAROS-TVL `Models` tab, where every computation graph element is conditioned and no feature is selected in the feature diagram (all rectangles are white). In Figure 22, `Issues` tab shows all of the `turtlerand` runtime issues. Then, after expanding the feature diagram, it is possible to select or deselect its features - the computation graph gets automatically adapted, as well as the runtime issues list.

Let us, for instance, select the `Turtlerand` feature; `Turtlerand` feature changed its colour to green (Figure 23) but so `Turtlesim` did. Looking at `Turtlerand` node cardinality (`[1..1]`) and `Turtlesim` optionality indicator, it is possible to understand why did that happen: when `Turtlerand` is selected, one child feature of a total of one child feature (which boils down to `Turtlesim`) must be selected and that child feature (`Turtlesim`) is not optional, so `Turtlesim` is automatically included. Notice that in the computation graph, `/Turtlesim` node circle gets thick, because that ROS resource was depending on `Turtlesim` feature. The runtime issues remain unchanged since there is not any issue depending on those two features.

Let us now see what happens when selecting the remaining features; `Turtlesim` cardinality indicates that at least one out of its two children features must be selected, but looking at each child, both have an empty circle, meaning that both are optional (remember that optionality prevails against cardinality). Four situations are then allowed:

- deselect `Random` and `Teleop`;
- select `Random` and deselect `Teleop`;
- deselect `Random` and select `Teleop`;
- select `Random` and `Teleop`.



**Issue #1 - Rule Parameters readers**

**One or more parameters are being readed.**

Query found; This query isn't depending on any node

custom

**Issue #2 - Rule Parameters readers**

**One or more parameters are being readed.**

Query found: This query is depending on Random and Teleop

custom

**Issue #3 - Rule Parameters readers**

**One or more parameters are being readed.**

Query found: This query is depending on Random

custom

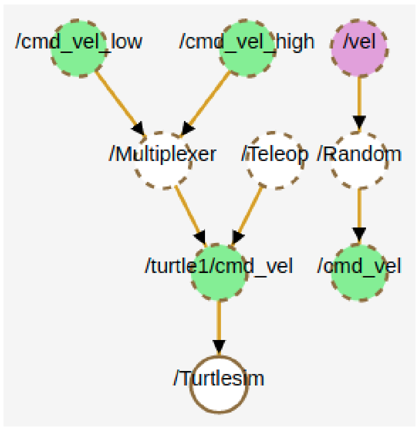
**Issue #4 - Rule Parameters readers**

**One or more parameters are being readed.**

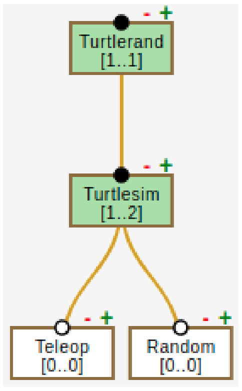
Query found: This query is depending on Teleop

custom

Figure 22: Initial state of `Issues` tab for the `turtlerand` application.

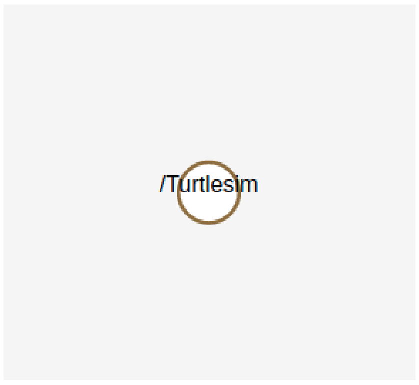


(a) Computation graph.

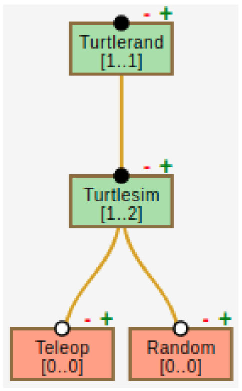


(b) Feature diagram.

Figure 23: Resulting state of `Models` tab after selecting `Turtlerand` feature (in Figure 21).

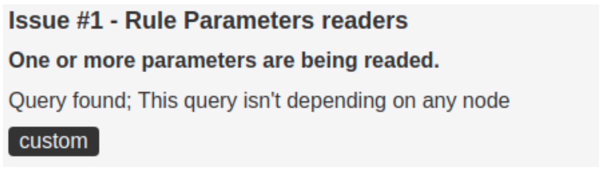


(a) Computation graph.



(b) Feature diagram.

Figure 24: Resulting state of `Models` tab after deselecting `Random` and `Teleop` features (in Figure 23).



**Issue #1 - Rule Parameters readers**

**One or more parameters are being readed.**

Query found; This query isn't depending on any node

custom

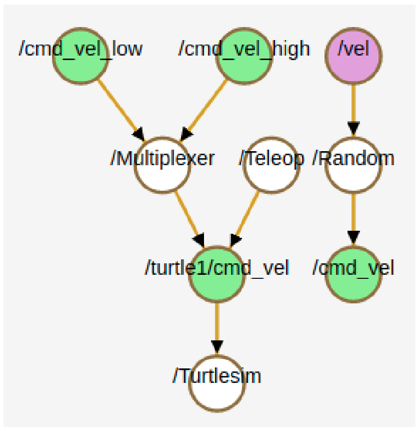
Figure 25: Resulting state of `Issues` tab after deselecting `Random` and `Teleop` features (in Figure 23).

By opting for deselecting `Random` feature, a large number of computation graph elements which were depending on it disappear. The same happens by deselecting `Teleop`, as Figure 24 illustrates. `/Turtlesim` node is the only element and `Issue #1` (Figure 25) the only issue exclusively depending on `Turtlesim` feature.

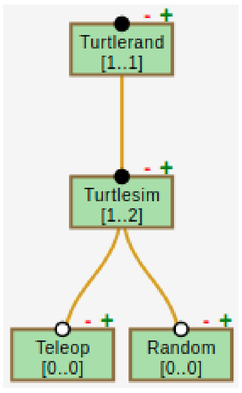
By opting to select both `Random` and `Teleop` features, the opposite happens, as all computation graph elements change from dashed to thick, as seen on Figure 26, and the `Issues` tab remains with the same initial issues.

### 7.1.5 *Conclusion*

All in all, by analyzing the `turtlerand` application with HAROS-TVL, one can clearly have an immediate idea of its architecture by looking at the feature diagram. The launch file is compact and readable. Another crucial aspect, and probably the most important is the fact that the computation graph gets projected to the essential elements, which are those related to the selected and unassigned features. Within these four features, the HAROS-TVL was able to compute the four possible configurations.



(a) Computation graph.



(b) Feature diagram.

Figure 26: Resulting state of `Models` tab after selecting `Random` and `Teleop` features (in Figure 23).

## 7.2 APPLYING HAROS-TVL TO `TURTLEBOT`

In comparison to `turtlerand`, the `turtlebot` application is much larger, comprising a higher number of features. Inevitably, that leads to a higher variability which can be experienced by interacting with the HAROS-TVL computation graph. `turtlebot` was specifically created to serve as an example like `turtlerand` did, and it is in fact much older than the current project. Yet, besides others, that factor was really important to understand how HAROS-TVL would behave with applications that were not built according to the proposed structure.

### 7.2.1 *Defining a TVL file*

The process to define the `turtlebot` feature model is no different from the other applications, requiring a detailed domain study to understand how to group the existing ROS resources into features, where the existing launch files play an important role. One should not ignore the fact that building feature models for unowned applications is much more difficult than for the others, once there will always exist some details which will be unnoticed, given the lack of a formal specification and good documentation. Therefore, this TVL feature model is only a suggestion, what means that there are certainly other possible and valid models.

Having said that, in this preliminary study twenty-two features were inferred from the `turtlebot` application domain. The root feature, `TurtleBot`, represents the whole application which splits between `Minimal` and `Teleop` features, and so on. Some incompatibilities were found involving `Logitech`, `Ps3` and `Xbox360`, of which:

- if `Logitech` is selected, `Ps3` cannot be;

- if `Logitech` is selected, `Xbox360` cannot be either;
- if `Ps3` is selected, `Xbox360` cannot be;

All of that led to the feature model in Listing 7.2.

```
root TurtleBot {
    group allOf {
        opt Minimal,
        opt Teleop
    }
    Logitech -> !Ps3;
    Logitech -> !Xbox360;
    Ps3 -> !Xbox360;
}

Minimal {
    group allOf {
        Base,
        Sensor_3D,
        Stacks,
        opt Battery
    }
}

Base {
    group oneOf {
        Create,
        Roomba,
        Kobuki
    }
}

Kobuki {
    group allOf {
        opt Bumper2pc,
        opt Safety_controller
    }
}

Sensor_3D {
    group oneOf {
        Kinect,
        Asus_xtion_pro,
        Astra,
        R200
    }
}

```

```
Stacks {
    group oneOf {
        Circles,
        Hexagons
    }
}

Teleop {
    group someOf {
        Keyboard,
        Logitech,
        Ps3,
        Xbox360
    }
}
```

Listing 7.2: `turtlebot` TVL feature model.

### 7.2.2 *Defining a generic launch file*

Besides the TVL file, the generic launch file was also defined taking into account the existing launch files. In essence, the idea was to create a copy of the existing launch file and adapt it with new `arg` tags corresponding to TVL model features. Then, `group` tags with `if` attributes are placed wrapping the corresponding launch file tags. Some `include` tags were replaced by the content of the file they were including in order to apply the respective conditions, a decision that made the launch file content to grow a lot and also caused some nodes to be repeated.

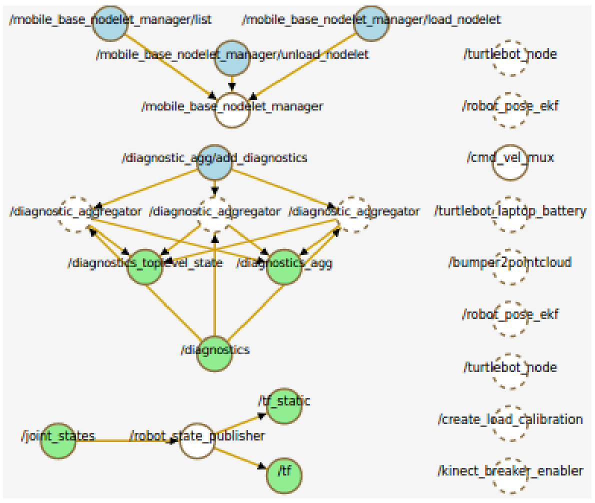
The resulting generic launch file is shown in Listing A.1.

### 7.2.3 *Defining a project configuration file*

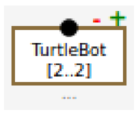
The project configuration file is as simple as Listing 7.3, by specifying the generic launch file and the corresponding TVL file. Queries and some ROS resources that are not automatically extracted in the analysis may be declared too.

```
project: SAFER
packages: ["turtlebot_bringup"]
configurations:
    {lf: turtlebot_bringup/launch/generic.launch, tvl:
        turtlebot/launch/turtlebot.tvl}
```

Listing 7.3: `turtlebot` project configuration file.



(a) Computation graph.



(b) Feature diagram.

Figure 27: Initial state of `Models` tab for `turtlebot` application.

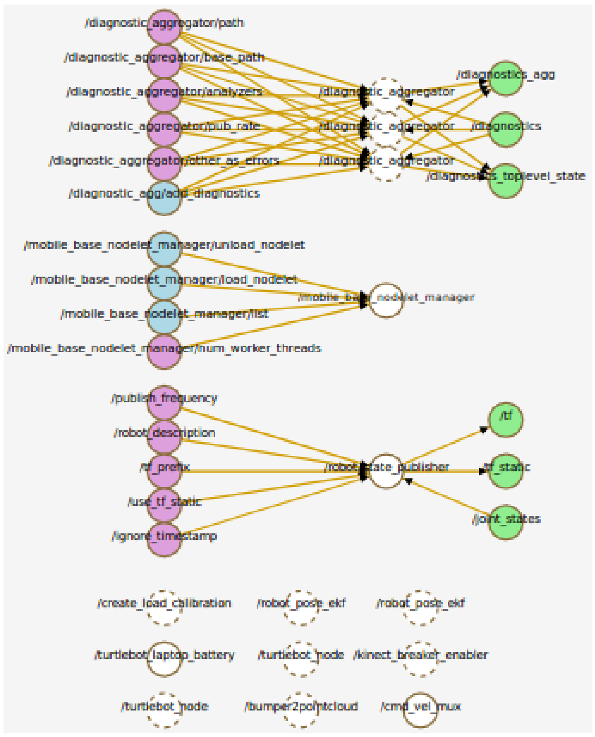
### 7.2.4 *Evaluation*

Considering all the possible configurations that can be inferred from the current feature model, which has twenty-two features, one quickly realizes how big this one is. For that reason, only some configurations will be covered. Figure 27 depicts the initial state of both graphs (parameters are hidden).

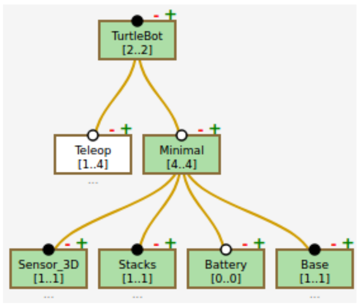
Let us start by revealing `TurtleBot` and `Minimal` features children, and then to select `Battery` feature (Figure 28). Immediately, `Sensor_3D`, `Stacks` and `Base` features get selected too, because, in order to `Battery` become selected, `Minimal` has to be selected, and by selecting `Minimal`, these additional three features have necessarily to be selected too. These examples of automatic selections confirm a key quality of HAROS-TVL. If done manually, that operation would require a tremendous effort from the user. Also, some computation graph elements change from dashed to thick.

The expansion of the remaining descendants of `Base`, `Stacks` and `Sensor_3D` features, of which `Bumper2pc`, `Safety_controller`, `Hexagons` and `Astra` are manually selected, shows the magnitude of the feature model. These operations lead to some automatic selections/deselections (namely, those made on `Base` children, which can only have one child selected), update some computation graph elements and eliminate others, like it is shown in Figure 29.

Now that nothing else can be done in the `Minimal` feature nor in its descendants, this feature can be collapsed (to reduce feature diagram width) and the `Teleop` feature expanded. Before any other operation, remind that the feature model has some constraints involving `Logitech` feature. To test the impact they have on the feature model, let us deliberately select

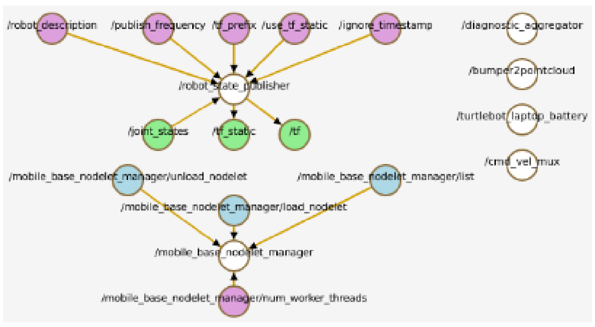

/diagnostic_aggregator/path
/diagnostic_aggregator/base_path
/diagnostic_aggregator/analyzers
/diagnostic_aggregator/pub_rate
/diagnostic_aggregator/other_as_errors
/diagnostic_agg/add_diagnostics
/diagnostic_aggregator
/diagnostic_aggregator
/diagnostic_aggregator
/diagnostics_agg
/diagnostics
/diagnostics_toplevel_state
/mobile_base_nodelet_manager/unload_nodelet
/mobile_base_nodelet_manager/load_nodelet
/mobile_base_nodelet_manager/list
/mobile_base_nodelet_manager/num_worker_threads
/mobile_base_nodelet_manager
/publish_frequency
/robot_description
/tf_prefix
/use_tf_static
/ignore_timestamp
/robot_state_publisher
/tf
/tf_static
/joint_states
/create_load_calibration
/robot_pose_ekf
/robot_pose_ekf
/turtlebot_laptop_battery
/turtlebot_node
/kinect_breaker_enabler
/turtlebot_node
/bumper2pointcloud
/cmd_vel_mux

(a) Computation graph.

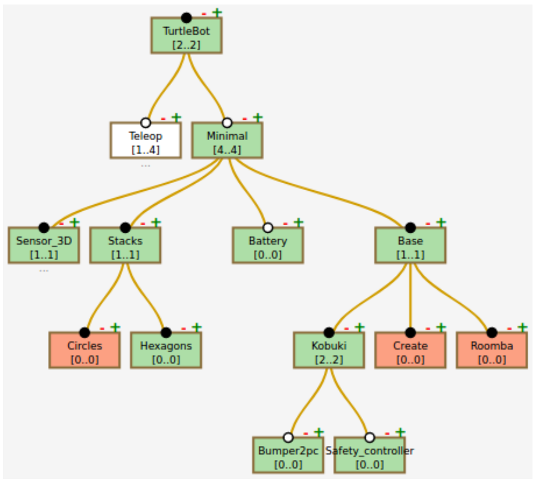


(b) Feature diagram.

Figure 28: Resulting state of `Models` tab after selecting `Battery` feature (in Figure 27).

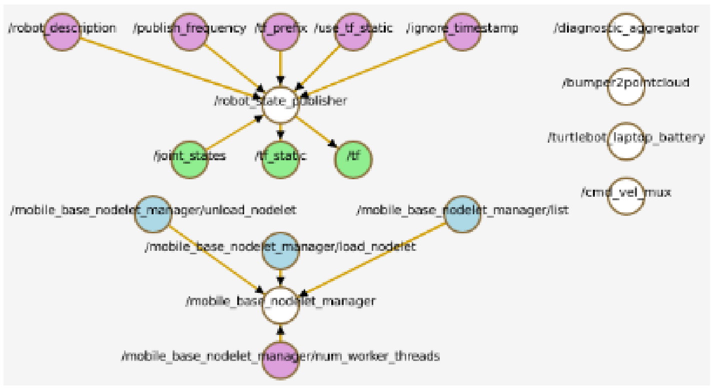

/robot_description
/publish_frequency
/tf_prefix
/use_tf_static
/ignore_timestamp
/diagnostic_aggregator
/robot_state_publisher
/bumper2pointcloud
/joint_states
/tf_static
/tf
/turtlebot_laptop_battery
/cmd_vel_mux
/mobile_base_nodelet_manager/unload_nodelet
/mobile_base_nodelet_manager/list
/mobile_base_nodelet_manager/load_nodelet
/mobile_base_nodelet_manager
/mobile_base_nodelet_manager/num_worker_threads

(a) Computation graph.

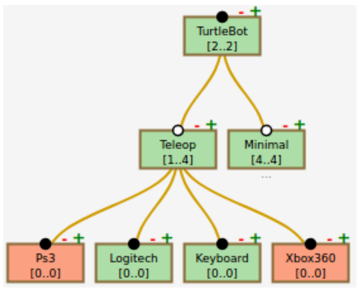


(b) Feature diagram.

Figure 29: Resulting state of `Models` tab after selecting four features (in Figure 28).


/robot_description
/publish_frequency
/tf_prefix
/use_tf_static
/ignore_timestamp
/diagnostic_aggregator
/bumper2pointcloud
/robot_state_publisher
/turtlebot_laptop_battery
/joint_states
/tf_static
/tf
/cmd_vel_mux
/mobile_base_nodelet_manager/unload_nodelet
/mobile_base_nodelet_manager/list
/mobile_base_nodelet_manager/load_nodelet
/mobile_base_nodelet_manager
/mobile_base_nodelet_manager/num_worker_threads

(a) Computation graph.



(b) Feature diagram.

Figure 30: Resulting state of `Models` tab after selecting `Logitech` and `Keyboard` features (in Figure 29).

`Logitech` feature; `Ps3` and `Xbox360` get automatically unselected due to those constraints, as expected.

Finally, the `Keyboard` feature is selected too.Figure 30 contains the resulting feature model and the computation graph which are now representing a concrete configuration. If one rather chooses to deselect `Keyboard` feature, a different concrete configuration is shown.

### 7.2.5 *Conclusion*

As expected, this larger example better illustrated what HAROS-TVL is or is not capable of. Some conclusions from this example are:

- The generic launch file can get large and complex - this problems are due to the fact that it is an adaptation of existing launch files and not built from scratch according to the HAROS-TVL required format;
- The TVL library was not able to calculate the total number of possible configurations due to the feature model complexity - that operation has timed out;
- The fact that the diagram is collapsible/expandable, prevents it from getting too large to be understood - that allows HAROS-TVL users to focus on what is important and what is not at any moment;
- The computation graph got more and more succinct as features got selected/unselected, allowing HAROS-TVL users to better understand the architecture of the application.

## 7.3 APPLYING HAROS-TVL TO LIZI

The lizi application is a particular case where its authors were already adopting a SPL development methodology. In what concerns to its original launch file, arg tags were declared and other ROS resources related tags were wrapped in conditions involving those arg tags values. In some way, the lizi application is already sharing some commonalities with turtlerand, grouping ROS resources and treating them as features. Nonetheless, it was not obviously built according to the HAROS-TVL setup rules.

### 7.3.1 *Defining a TVL file*

The process of defining a TVL file was simplified in this case thanks to the previous reasons, but its domain still needs an in-depth study. Therefore, sixteen features were found, three of them with children. Three constraints were also disclosed, namely:

- Move_base and Lidar features must be selected every time Gmapping feature is selected too;
- Move_base and Lidar features must also be selected every time Hector_slam feature is selected too;
- Move_base, Lidar and Map features must be selected every time Amcl feature is selected too;

Listing 7.4 shows the resulting feature model.

```
root Lizi {
    group allOf {
        opt Hw,
        opt Navigation,
        opt Espeak,
        opt Robot_state,
        opt Gazebo
    }
    Gmapping -> (Move_base && Lidar);
    Hector_slam -> (Move_base && Lidar);
    Amcl -> (Move_base && Lidar && Map);
}

Hw {
    group someOf {
        Depth_cam,
        Lidar,
        Cam,
        Diagnos
    }
}

Navigation {
    group someOf {
        Gmapping,
        Hector_slam,
        Robot_localization,
        Amcl,
        Move_base,
        Map
    }
}
```

Listing 7.4: `lizi` TVL feature model.

### 7.3.2 *Defining a generic launch file*

Again, the most reasonable approach is to build a new `lizi` generic launch file by adapting the existing one. Unlike what happened with `turtlebot` generic launch file, the modifications required in `lizi` are much smaller precisely because it is already feature-aware.

Thus, one only had to do some minor changes, such as:

- modify `arg` names to match the TVL model features;
- separate `arg` tags which correspond to features from those which do not - the `group` tag technique (Listing 5.5) was employed to do this;
- add six more features as `arg` tags - `Lizi`, `Hw`, `Navigation`, `Espeak`, `Robot_state` and `Gazebo`.

These modifications resulted in Listing A.2.

### 7.3.3 *Defining a project configuration file*

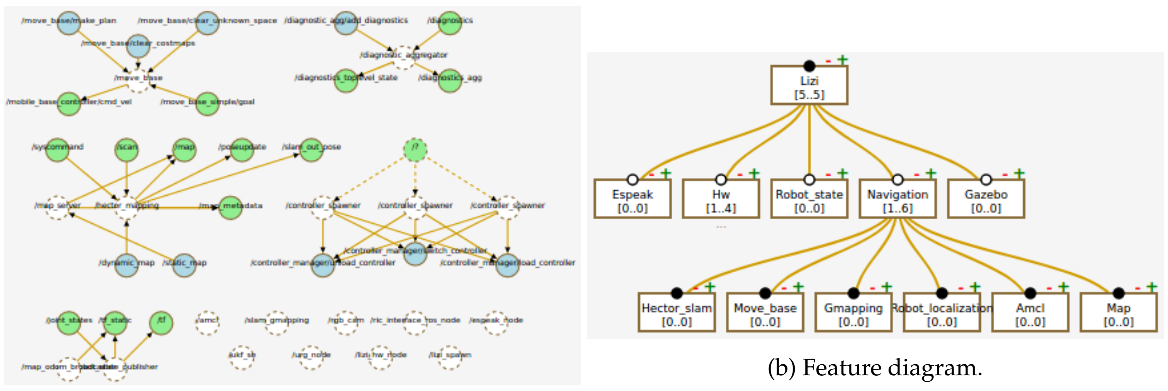
The project configuration file structure is something that does not really change from application to application, unless one specifies hints for a non-extracted element, a rule, a plugin blacklist, among other elements. Listing 7.5 shows the project configuration file of `lizi`, which only differs in the syntax used for defining the file paths.

```
project: SAFER
packages: ["lizi"]
configurations:
    lf:
        lizi/launch/generic.launch
    tvl:
        lizi/launch/generic.tvl
```

Listing 7.5: `lizi` project configuration file.

### 7.3.4 *Evaluation*

The initial state of the feature model and computation graph is shown by Figure 31, where `Lizi` and `Navigation` features were expanded. Again, given the feature diagram size, it will be expanded and collapsed as features are being selected/unselected.

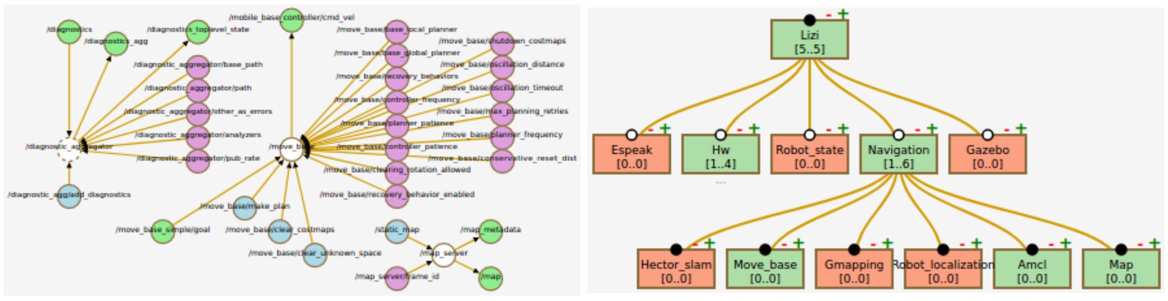


(a) Computation graph.

(b) Feature diagram.

Figure 31: Initial state of `Models` tab for `lizi` application.

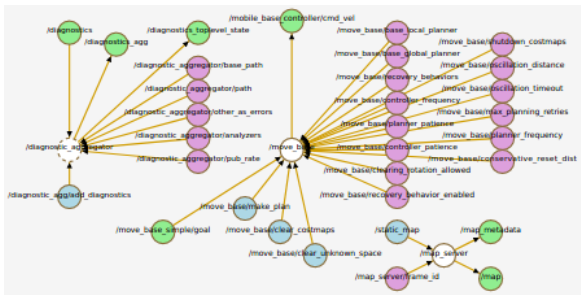
Let us now select the `Amcl` feature. Immediately, other selections are triggered - `Move_base`, `Lidar` and `Map` - in line with the previously mentioned constraints. Then, deselecting the remaining direct children of `Lizi` and `Navigation` features results in Figure 32.



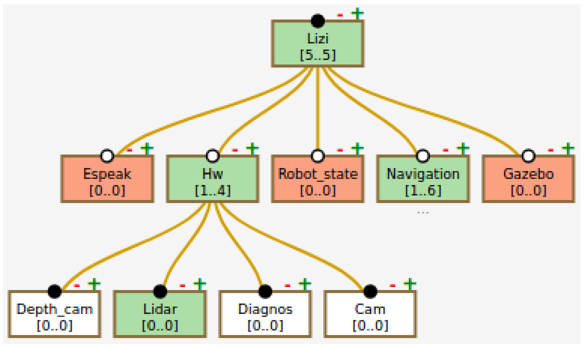
(a) Computation graph.

(b) Feature diagram.

Figure 32: Resulting state of `Models` tab after selecting `Amcl` feature and deselecting others (in Figure 31).



(a) Computation graph.

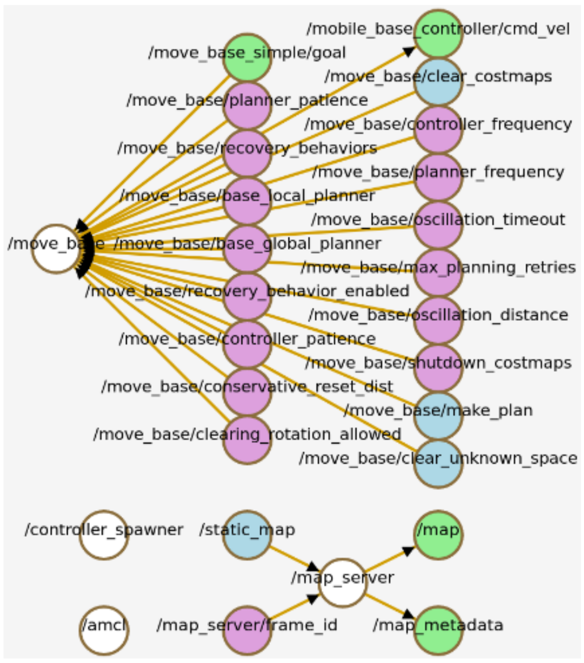


(b) Feature diagram.

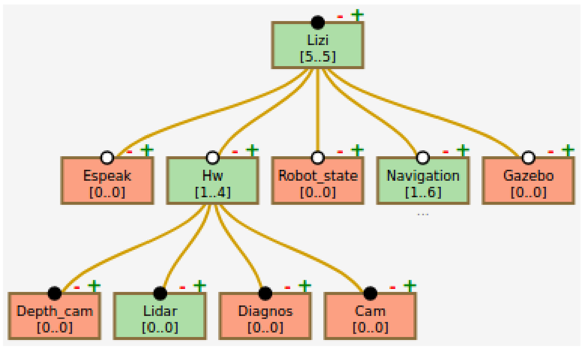
Figure 33: Resulting state of `Models` tab after collapsing `Navigation` and expanding `Hw` features (in Figure 32).

Next, by collapsing `Navigation` and expanding the `Hw` features, the feature diagram gets more intelligible (Figure 33).

The configuration is almost completed, but there are three features more that remain unassigned - `Depth_cam`, `Diagnos` and `Cam`. By deselecting these three, the computation graph gets even more simpler to understand, without any node depending on unassigned features.


/mobile_base_controller/cmd_vel
/move_base_simple/goal
/move_base/clear_costmaps
/move_base/planner_patience
/move_base/controller_frequency
/move_base/recovery_behaviors
/move_base/planner_frequency
/move_base/base_local_planner
/move_base/oscillation_timeout
/move_base
/move_base/base_global_planner
/move_base/max_planning_retries
/move_base/recovery_behavior_enabled
/move_base/oscillation_distance
/move_base/controller_patience
/move_base/shutdown_costmaps
/move_base/conservative_reset_dist
/move_base/make_plan
/move_base/clearing_rotation_allowed
/move_base/clear_unknown_space
/controller_spawner
/static_map
/map
/map_server
/amcl
/map_server/frame_id
/map_metadata

(a) Computation graph.



(b) Feature diagram.

Figure 34: Resulting state of `Models` tabs after deselecting `Depth_cam`, `Diagnos` and `Cam` features (in Figure 33).

### 7.3.5 *Conclusion*

This last case study was able to unveil some aspects that were not experienced in the previous examples. On the other hand, it also allowed us to corroborate some of HAROS-TVL qualities.

This time, the TVL library was able to compute all the possible configurations arising from the interaction of those sixteen features: a total of 2304 possible configurations. That is an astronomic number, and its unlikely that any user would be aware of all of them. In fact, without the TVL library (or a similar tool) it would be nearly impossible to compute them, as it is not enough to disclose all of the possible combinations between features - all of the relations between them alongside the existing constraints must be taken into consideration too.

Here are some other important conclusions:

- An application built according to a SPL methodology is not only easier to understand but it also simplifies the process of adaptation to HAROS-TVL;
- `lizi` and `turtlebot` are both applications that needed to be adapted for HAROS-TVL, therefore they share similar aspects in the setup files definition process; in essence, both situations act as examples to be taken into consideration when adapting such kind of applications;
- Once more, toggling the feature diagram appearance prevents it from getting too big to be understood, as well as the computation graph (aspects which were already experienced before).

It is important to mention that the resulting configurations arising from the interaction with `turtlerand`, `turtlebot` and `lizi` feature models were chosen randomly, because the main goal was to navigate through all the available functionalities of the feature diagrams and exhibit them to the reader, and not to reach a specific configuration. That is not the way a user would interact with HAROS-TVL, as he would firstly idealize the features to select/deselect and then apply that to the feature diagram. Simply put, the fact that explanations were not tied to a specific configuration, allowed to showcase a lot more functionalities.

# 8 CONCLUSION

The overall goal of this thesis was to extend HAROS to better handle variability. With that goal in mind, a detailed study of the state of the art was conducted.

Firstly, a ROS example application - `turtlerand` - was implemented according to the provided tutorials and the available examples. Although this step might seem superfluous, it helped a lot to understand how does that framework operate and what tools it has to support the development. In particular it helped master the definition of launch files, a key artifact for this thesis. Secondly, the HAROS framework was reviewed using the example `turtlerand` application. This was an opportunity to disclose some problems with HAROS when dealing with applications with variability. Finally, a thorough study was made to collect information about existing methodologies to handle variability. In this study it became clear that the TVL language (and its Java library) was the best approach to model variability in HAROS.

We defined a methodology to link TVL feature models with generic launch files, to be used in the HAROS-TVL extension to be implemented. Then, the development process of HAROS-TVL server started. The server development was divided into multiple tasks, namely, executing the TVL library under a NodeJS server, converting a TVL feature model into a D3 JSON, creating handlers to feature model operations, performing changes inside HAROS `configurations.json` file, and so on. Finally, the HAROS-TVL extension was developed based on this server.

Chapter 7 evaluated HAROS-TVL with a few case studies, and shows it is capable of properly handle the variability arising from the existence of multiple software features. We believe the extra work required to define a TVL feature model (a formal specification of the application domain) and to define a generic launch file pays up due to the following advantages:

- It depicts an interactive feature diagram which is user friendly (namely, it automatically includes/excludes features) and scalable (the collapsing/expanding of features allows it to not become too complex);
- It reduces the computation graph to match the selected features only, which greatly simplifies the understanding of the application architecture for concrete configurations;

- Likewise for the runtime issues;
- It is able to compute all possible configurations of the applications, something that is unfeasible to do manually for large feature models.

There are still some problems, namely, the generic launch file can get excessively big and the TVL library is unable to compute all possible configurations with large feature models.

## 8.1 FUTURE WORK

Although HAROS-TVL seems promising, some particular details require a few improvements. These are not critical to HAROS-TVL functionality, acting more like suggestions to make this tool even better. They are mostly related to the TVL library or HAROS bugs and improvements, and are detailed in the following paragraphs.

At the moment, the HAROS model extraction process is done by analysing ROS resources source code, but it cannot simply rely on this information, for instance, to tell whether a node is subscribing/advertising a certain topic. As a matter of fact, in the evaluated case studies, the great majority of ROS nodes suffered some kind of topic remapping, however the HAROS computation graph is unable to show them and it should not miss such an important piece of information. A suggestion to solve that problem would be to reference the node's remapped topics and also to record the conditional constraints they are under.

In order to filter the `Issues` tab issues according to the selected/unselected/unassigned features of the feature model, these must have some kind of information that relates them to those features. Therefore, it would be useful to include a `conditions` array in every JSON object that is representing an issue, just like those that appear in the `configurations.json` objects for the computation graph elements. For the moment, we manually inserted those conditions to test the functionality.

The computation of all possible configurations, in the TVL library referred as products, is a valuable add-on to HAROS-TVL, simply because it helps to understand how large a SPL is. For this functionally to work properly the TVL library needs to be improved, in order to efficiently count the number of products in large feature models, even if only approximately. Additionally, HAROS-TVL should allow to compute all possible configurations corresponding to a given feature model state.

# BIBLIOGRAPHY

[1] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. Introducing tvl, a text-based feature modelling language. In *VaMoS' 2010*, 2010.

[2] Renato Carvalho, Alcino Cunha, Nuno Macedo, and André Santos. Verification of system-wide safety properties of ROS applications. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2020, Las Vegas, NV, USA, October 24, 2020 - January 24, 2021*, pages 7249–7254. IEEE, 2020. doi: 10.1109/IROS45743.2020.9341085. URL https://doi.org/10.1109/IROS45743.2020.9341085.

[3] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling: Syntax and semantics of tvl. *Science of Computer Programming*, 76(12): 1130–1143, 2011. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2010.10.005. URL https://www.sciencedirect.com/science/article/pii/S0167642310001899. Special Issue on Software Evolution, Adaptability and Variability.

[4] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.

[5] Sergio García, Daniel Strüber, Davide Brugali, Alessandro Di Fava, Philipp Schillinger, Patrizio Pelliccione, and Thorsten Berger. Variability modeling of service robots: Experiences and challenges. In Danny Weyns and Gilles Perrouin, editors, *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS 2019, Leuven, Belgium, February 06-08, 2019*, pages 8:1–8:6. ACM, 2019. doi: 10.1145/3302333.3302350. URL https://doi.org/10.1145/3302333.3302350.

[6] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990. URL http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231.

[7] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 311–320, 2008. doi: 10.1145/1368088.1368131.

[8] Niloofar Mansoor, Jonathan A. Saddler, Bruno Silva, Hamid Bagheri, Myra B. Cohen, and Shane Farritor. Modeling and testing a family of surgical robots: An experience

report. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 785–790, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3275534. URL https://doi.org/10.1145/3236024.3275534.

[9] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005. ISBN 978-3-540-24372-4. doi: 10.1007/3-540-28901-1. URL https://doi.org/10.1007/3-540-28901-1.

[10] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[11] André Santos. *Safety Verification for ROS Applications*. PhD thesis, University of Minho, Portugal, 2021. URL https://git-afsantos.github.io/publication/phd-thesis.

[12] André Santos, Alcino Cunha, Nuno Macedo, and Cláudio Lourenço. A framework for quality assessment of ROS repositories. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4491–4496, 2016. doi: 10.1109/IROS.2016.7759661.

[13] André Santos, Alcino Cunha, and Nuno Macedo. Static-time extraction and analysis of the ROS computation graph. In *IEEE International Conference on Robotic Computing (IRC)*, pages 62–69, 2019. doi: 10.1109/IRC.2019.00018.

[14] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. Variability-aware static analysis at scale: An empirical study. *ACM Transactions on Software Engineering and Methodology*, 27:1–33, 11 2018. doi: 10.1145/3280986.

# A LISTINGS

```
<launch>
    <arg name="TurtleBot"/>
    <arg name="Minimal"/>
    <arg name="Base" doc="mobile base type [create, roomba]"/>
    <arg name="Battery"/>
    <arg name="Sensor_3D" doc="3d sensor types [kinect, asux_xtion_pro]"/>
    <arg name="Stacks" doc="stack type displayed in visualisation/simulation [
        circles, hexagons]"/>
    <arg name="Create"/>
    <arg name="Roomba"/>
    <arg name="Kobuki"/>
    <arg name="Kinect"/>
    <arg name="Asus_xtion_pro"/>
    <arg name="Circles"/>
    <arg name="Hexagons"/>
    <arg name="Keyboard"/>
    <arg name="Teleop"/>
    <arg name="Logitech"/>
    <arg name="Ps3"/>
    <arg name="Xbox360"/>
    <arg name="Safety_controller"/>
    <arg name="Bumper2pc"/>
    <arg name="Astra"/>
    <arg name="R200"/>

    <group if="true">

        <arg name="simulation" default="$(env TURTLEBOT_SIMULATION)" doc="set
            flags to indicate this turtle is run in simulation mode."/>
        <arg name="serialport" default="$(env TURTLEBOT_SERIAL_PORT)" doc="used
            by create to configure the port it is connected on [/dev/ttyUSB0, /dev
            /ttyS0]"/>
        <arg name="battery" doc="kernel provided locatioN for battery info, use /
            proc/acpi/battery/BAT0 in 2.6 or earlier kernels." />

        <param name="/use_sim_time" value="$(arg simulation)"/>

        <group if="$(arg Base)">
            <group if="$(arg Stacks)">
                <group if="$(arg Sensor_3D)">

                    <group if="$(arg Create)">

                        <group if="$(arg Circles)">
                            <group if="$(arg Kinect)">
                                <arg name="urdf_file" default="$(find xacro)/
                                    xacro.py '$(find turtlebot_description)/robots
                                    /create_circles_kinect.urdf.xacro'" />
                                <param name="robot_description" command="$(arg
                                    urdf_file)" />
                            </group>
                            <group if="$(arg Asus_xtion_pro)">
                                <arg name="urdf_file" default="$(find xacro)/
                                    xacro.py '$(find turtlebot_description)/robots
                                    /create_circles_asus_xtion_pro.urdf.xacro'" />
                                <param name="robot_description" command="$(arg
                                    urdf_file)" />
                            </group>
                        </group>

                        <group if="$(arg Hexagons)">
                            <group if="$(arg Kinect)">
                                <arg name="urdf_file" default="$(find xacro)/
                                    xacro.py '$(find turtlebot_description)/robots
                                    /create_hexagons_kinect.urdf.xacro'" />
                                <param name="robot_description" command="$(arg
                                    urdf_file)" />
                            </group>
                            <group if="$(arg Asus_xtion_pro)">
                                <arg name="urdf_file" default="$(find xacro)/
                                    xacro.py '$(find turtlebot_description)/robots
                                    /create_hexagons_asus_xtion_pro.urdf.xacro'"
                                    />
                                <param name="robot_description" command="$(arg
                                    urdf_file)" />
                            </group>
                        </group>

                        <node pkg="diagnostic_aggregator" type="aggregator_node"
                            name="diagnostic_aggregator">
                            <rosparam command="load" file="$(find
                                turtlebot_bringup)/param/create/diagnostics.yaml"
                                />
                        </node>

                        <include file="$(find turtlebot_bringup)/launch/includes/
                            create/mobile_base.launch.xml">
                            <arg name="serialport" value="$(arg serialport)"/>
                            <arg name="manager" value="
                                mobile_base_nodelet_manager"/>
                        </include>
                    </group>

                    <group if="$(arg Roomba)">

                        <group if="$(arg Circles)">
                            <group if="$(arg Kinect)">
                                <arg name="urdf_file" default="$(find xacro)/
                                    xacro.py '$(find turtlebot_description)/robots
                                    /roomba_circles_kinect.urdf.xacro'" />
                                <param name="robot_description" command="$(arg
                                    urdf_file)" />
                            </group>
                            <group if="$(arg Asus_xtion_pro)">
                                <arg name="urdf_file" default="$(find xacro)/
                                    xacro.py '$(find turtlebot_description)/robots
                                    /roomba_circles_asus_xtion_pro.urdf.xacro'" />
                                <param name="robot_description" command="$(arg
                                    urdf_file)" />
                            </group>
                        </group>

                        <group if="$(arg Hexagons)">
                            <group if="$(arg Kinect)">
                                <arg name="urdf_file" default="$(find xacro)/
                                    xacro.py '$(find turtlebot_description)/robots
                                    /roomba_hexagons_kinect.urdf.xacro'" />
                                <param name="robot_description" command="$(arg
                                    urdf_file)" />
                            </group>
                            <group if="$(arg Asus_xtion_pro)">
                                <arg name="urdf_file" default="$(find xacro)/
                                    xacro.py '$(find turtlebot_description)/robots
                                    /roomba_hexagons_asus_xtion_pro.urdf.xacro'"
                                    />
                                <param name="robot_description" command="$(arg
                                    urdf_file)" />
                            </group>
                        </group>

                        <node pkg="diagnostic_aggregator" type="aggregator_node"
                            name="diagnostic_aggregator">
                            <rosparam command="load" file="$(find
                                turtlebot_bringup)/param/roomba/diagnostics.yaml"
                                />
                        </node>

                        <include file="$(find turtlebot_bringup)/launch/includes/
                            roomba/mobile_base.launch.xml">
                            <arg name="serialport" value="$(arg serialport)"/>
                            <arg name="manager" value="
                                mobile_base_nodelet_manager"/>
                        </include>
                    </group>

                    <group if="$(arg Kobuki)">

                        <group if="$(arg Circles)">
                            <group if="$(arg Kinect)">
                                <arg name="urdf_file" default="$(find xacro)/
                                    xacro.py '$(find turtlebot_description)/robots
                                    /kobuki_circles_kinect.urdf.xacro'" />
                                <param name="robot_description" command="$(arg
                                    urdf_file)" />
                            </group>
                            <group if="$(arg Asus_xtion_pro)">
                                <arg name="urdf_file" default="$(find xacro)/
                                    xacro.py '$(find turtlebot_description)/robots
                                    /kobuki_circles_asus_xtion_pro.urdf.xacro'" />
                                <param name="robot_description" command="$(arg
                                    urdf_file)" />
                            </group>
                        </group>

                        <group if="$(arg Hexagons)">
                            <group if="$(arg Kinect)">
                                <arg name="urdf_file" default="$(find xacro)/
                                    xacro.py '$(find turtlebot_description)/robots
                                    /kobuki_hexagons_kinect.urdf.xacro'" />
                                <param name="robot_description" command="$(arg
                                    urdf_file)" />
                            </group>
                            <group if="$(arg Asus_xtion_pro)">
                                <arg name="urdf_file" default="$(find xacro)/
                                    xacro.py '$(find turtlebot_description)/robots
                                    /kobuki_hexagons_asus_xtion_pro.urdf.xacro'"
```

```
                                    />
                                <param name="robot_description" command="$(arg urdf_file)" />
                            </group>
                        </group>

                        <node pkg="diagnostic_aggregator" type="aggregator_node" name="diagnostic_aggregator">
                            <rosparam command="load" file="$(find turtlebot_bringup)/param/kobuki/diagnostics.yaml" />
                        </node>

                        <include file="$(find turtlebot_bringup)/launch/includes/kobuki/mobile_base.launch.xml">
                            <arg name="serialport" value="$(arg serialport)"/>
                            <arg name="manager" value="mobile_base_nodelet_manager"/>
                        </include>

                    </group>

                </group>
            </group>
        </group>

        <node pkg="nodelet" type="nodelet" name="mobile_base_nodelet_manager" args="manager"/>

        <param name="robot/name" value="$(optenv ROBOT turtlebot)"/>
        <param name="robot/type" value="turtlebot"/>

        <node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_state_publisher">
            <param name="publish_frequency" type="double" value="5.0" />
        </node>

        <node pkg="nodelet" type="nodelet" name="cmd_vel_mux" args="load yocs_cmd_vel_mux/CmdVelMuxNodelet mobile_base_nodelet_manager">
            <param name="yaml_cfg_file" value="$(find turtlebot_bringup)/param/mux.yaml"/>
            <remap from="cmd_vel_mux/output" to="mobile_base/commands/velocity"/>
        </node>

        <group if="$(arg Battery)">
            <include unless="$(eval arg('battery') == 'None')" file="$(find turtlebot_bringup)/launch/includes/netbook.launch.xml">
```

```xml
                <arg name="battery" value="$(arg battery)" />
            </include>
        </group>

        <group if="$(arg Teleop)">
            <group if="$(arg Keyboard)">
                <include file="$(find turtlebot_teleop)/launch/keyboard_teleop.launch"/>
            </group>

            <group if="$(arg Logitech)">
                <include file="$(find turtlebot_teleop)/launch/logitech.launch"/>

            </group>
            <group if="$(arg Ps3)">
                <include file="$(find turtlebot_teleop)/launch/ps3_teleop.launch" />
            </group>

            <group if="$(arg Xbox360)">
                <include file="$(find turtlebot_teleop)/launch/xbox360_teleop.launch"/>
            </group>
        </group>

        <group if="$(arg R200)">
            <include file="$(find realsense_camera)/launch/r200_nodelet_default.launch.xml"/>
        </group>

        <group if="$(arg Safety_controller)">
            <include file="$(find turtlebot_bringup)/launch/includes/kobuki/safety_controller.xml"/>
        </group>

        <group if="$(arg Bumper2pc)">
            <include file="$(find kobuki_bumper2pc)/launch/standalone.launch"/>
        </group>

        <group if="$(arg Astra)">
            <include file="$(find astra_launch)/launch/astra.launch"/>
        </group>

    </group>

</launch>
```

Listing A.1: `turtlebot` generic launch file.

```
<launch>
    <arg name="Lizi"/>
    <arg name="Hw"/>
    <arg name="Navigation"/>
    <arg name="Espeak"/>
    <arg name="Robot_state"/>
    <arg name="Gazebo" doc="execute lizi inside gazebo sim"/>
    <arg name="Cam"/>
    <arg name="Depth_cam"/>
    <arg name="Lidar"/>
    <arg name="Diagnos"/>
    <arg name="Gmapping"/>
    <arg name="Hector_slam"/>
    <arg name="Amcl"/>
    <arg name="Map" doc="set to true to use pre-saved map"/>
    <arg name="Move_base"/>
    <arg name="Robot_localization"/>

    <group>
        <arg name="world" default="worlds/empty.world"/>
        <arg name="x" default="0.0"/>
        <arg name="y" default="0.0"/>
        <arg name="z" default="0.0"/>
        <arg name="Y" default="0.0" />
        <arg name="map" default="map.yaml" doc="pre-saved map path"/>

        <group if="$(arg Robot_state)">
            <node name="robot_state_publisher" pkg="robot_state_publisher" type="
                robot_state_publisher" respawn="false" output="screen"/>
        </group>

        <group if="$(arg Espeak)">
            <include file="$(find espeak_ros)/launch/espeak_ros.launch" />
        </group>

        <group if="$(arg Diagnos)">
            <include file="$(find lizi_hw)/launch/diagnostics.launch" />
        </group>

        <group if="$(arg Amcl)">
            <include file="$(find lizi_control)/launch/lizi_controllers.launch">
                <arg name="enable_mbc_odom_tf" value="true"/>
            </include>
        </group>

        <group if="$(arg Robot_localization)">
            <include file="$(find lizi_control)/launch/lizi_controllers.launch">
                <arg name="enable_mbc_odom_tf" value="false"/>
            </include>
        </group>

        <group unless="$(arg Robot_localization)">
            <group unless="$(arg Amcl)">
                <include file="$(find lizi_control)/launch/lizi_controllers.
                    launch">
                    <arg name="enable_mbc_odom_tf" value="true"/>
                </include>
            </group>
        </group>

        <group if="$(arg Map)">
            <node name="map_server" pkg="map_server" type="map_server" args="$(
                arg map)" />
        </group>

        <group unless="$(arg Gmapping)">
            <group unless="$(arg Hector_slam)">
                <group unless="$(arg Amcl)">
                    <group unless="$(arg Robot_localization)">
                        <node pkg="tf" type="static_transform_publisher" name="
                            map_odom_broadcaster" args="0 0 0 0 0 0 /map /odom 20"
                             />
                    </group>
                </group>
            </group>
        </group>

        <group if="$(arg Gazebo)">
            <param name="robot_description" command="$(find xacro)/xacro '$(find
                lizi_description)/urdf/lizi_gazebo.xacro' --inorder depth_cam:=$(
                arg depth_cam) cam:=$(arg cam) urf:=true imu:=true gps:=true lidar
                :=true" />

            <include file="$(find gazebo_ros)/launch/empty_world.launch">
                <arg name="world_name" value="$(arg world)"/>
                <arg name="gui" value="true"/>
            </include>

            <node name="lizi_spawn" pkg="gazebo_ros" type="spawn_model" output="
                screen" args="-urdf -param robot_description -model lizi -x $(arg
                x) -y $(arg y) -z $(arg z) -Y $(arg Y)" />
        </group>

        <group unless="$(arg Gazebo)">
            <param name="robot_description" command="$(find xacro)/xacro '$(find
                lizi_description)/urdf/lizi.xacro' --inorder" />
            <include file="$(find lizi_hw)/launch/lizi_hw.launch" />

            <group if="$(arg Depth_cam)">
                <include file="$(find lizi_hw)/launch/d435_cam.launch" />
            </group>

            <group if="$(arg Cam)">
                <include file="$(find lizi_hw)/launch/microsoft_cam.launch" />
            </group>

            <group if="$(arg Lidar)">
                <include file="$(find lizi_hw)/launch/hokuyu_lidar.launch" />
            </group>
        </group>

        <group if="$(arg Gmapping)">
            <include file="$(find lizi_navigation)/launch/gmapping.launch" />
        </group>

        <group if="$(arg Hector_slam)">
            <include file="$(find lizi_navigation)/launch/hector_slam.launch" />
        </group>

        <group if="$(arg Robot_localization)">
            <include file="$(find lizi_navigation)/launch/robot_localization.
                launch" />
        </group>

        <group if="$(arg Amcl)">
            <include file="$(find lizi_navigation)/launch/amcl.launch">
                <arg name="initial_pose_x" value="$(arg x)"/>
                <arg name="initial_pose_y" value="$(arg y)"/>
                <arg name="initial_pose_a" value="$(arg Y)"/>
            </include>
        </group>

        <group if="$(arg Move_base)">
            <include file="$(find lizi_navigation)/launch/move_base.launch" />
            <rosparam file="$(find lizi_navigation)/config/move_base_params.yaml"
                 command="load" ns="move_base"/>
            <rosparam file="$(find lizi_navigation)/config/costmap_common_params.
                yaml" command="load" ns="move_base/global_costmap" />
            <rosparam file="$(find lizi_navigation)/config/costmap_common_params.
                yaml" command="load" ns="move_base/local_costmap" />
            <rosparam file="$(find lizi_navigation)/config/local_costmap_params.
                yaml" command="load" ns="move_base/local_costmap" />
            <rosparam file="$(find lizi_navigation)/config/global_costmap_params.
                yaml" command="load" ns="move_base/global_costmap"/>
        </group>
    </group>

</launch>
```

Listing A.2: `lizi` generic launch file.