Sara Maria Barreira Melo

Estudo Empírico da Variabilidade em Sistemas ROS

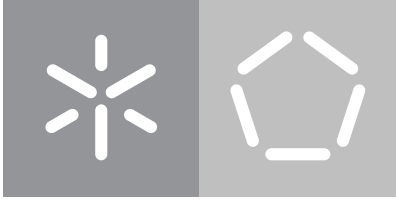
UMinho | 2021

Universidade do Minho
Escola de Engenharia

Sara Maria Barreira Melo

Estudo Empírico da Variabilidade em Sistemas ROS

outubro de 2021



Universidade do Minho
Escola de Engenharia

Sara Maria Barreira Melo

Estudo Empírico da Variabilidade em Sistemas ROS

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Trabalho efetuado sob a orientação de
Professor Doutor Manuel Alcino Pereira da Cunha

Outubro, 2021

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Braga, 22 de Outubro de 2021

(Localização) (Data)

Sara Maria Barreira Melo

(Sara Maria Barreira Melo)

Agradecimentos

A concretização desta tese não teria sido possível sem a ajuda e paciência das pessoas que sempre me incentivaram e ajudaram durante esta caminhada.

Em primeiro lugar quero agradecer ao meu orientador Alcino Cunha pela disponibilidade, conhecimento que transmitiu durante todo o processo e pela orientação que sempre me levou no melhor caminho. Agradeço também ao professor Nuno Macedo pela ajuda e ideias que sempre apresentou, assim como ao André Santos pelo conhecimento que sempre partilhou com generosidade.

A família foi um apoio fundamental durante todo este processo, e por isso é tão importante este agradecimento. Tenho que agradecer à minha mãe, Cândida por toda a educação, paciência e sinceridade, que durante toda a minha vida foram tão importantes e que agora dão os frutos de todo o trabalho que não é só meu, mas também dela. À minha irmã Rita tenho que agradecer por ser a minha melhor amiga, pela boa disposição e alegria com que sempre me presenteou, assim como todo o apoio e compreensão que manifestou desde sempre. Agradeço também aos meus avós, Jorge e Constança pelo carinho e enorme apoio que sempre me deram.

É importante agradecer também aos meus amigos que, durante esta jornada nunca deixaram de me apoiar e que se mostraram sempre compreensíveis. Quero assim agradecer à Inês, Maria Ana e Ana Luísa que, mesmo não fazendo ideia do que estava a ser feito aqui, sempre me apoiaram e incentivaram, tornando este projeto mais fácil de concretizar. Agradeço também ao Rui, por ser o excelente amigo que é e que sempre foi durante todos os anos de amizade. Durante este percurso universitário encontrei a melhor pessoa que este curso me deu, a Joana, quero agradecer-lhe por ter sido a melhor amiga para tudo, gargalhadas principalmente, pois sem ela teria sido muito mais difícil ultrapassar os obstáculos que fomos encontrando no caminho.

Agradeço ainda à Universidade do Minho que tão bem me acolheu durante estes 5 anos de aprendizagem. Orgulho-me de dizer que não poderia ter feito melhor escolha, pois toda a instituição revelou ser generosa e respeituosa para com todos os seus alunos.

Este trabalho foi realizado no contexto do projecto SAFER - *Safety Verification for Robotic Software*, financiado por Fundos FEDER através do Programa Operacional Competitividade e Internacionalização - COMPETE 2020 e por Fundos Nacionais através da FCT - Fundação para a Ciência e a Tecnologia no âmbito do projeto PTDC/CCI-INF/29583/2017 - POCI-01-0145-FEDER-029583.

Estudo Empírico da Variabilidade em Sistemas ROS

A utilização de sistemas robóticos tem vindo a crescer nos últimos anos e o *software* destes sistemas tem-se tornado cada vez mais importante para o seu funcionamento. O *Robot Operating System* (ROS) é um *middleware* que simplifica a implementação destes sistemas, fornecendo várias primitivas que facilitam a escrita de *software* e a coordenação dos diversos componentes que os constituem. Os sistemas ROS são distribuídos, com uma arquitetura organizada a partir de nós que comunicam entre si através da passagem de mensagens. Estes sistemas robóticos são fortemente configuráveis pois necessitam de se ajustar a ambientes de trabalho cada vez mais diversificados e adversos. Em sistemas ROS existem ficheiros que incluem a configuração do sistema e é através destes que se pensa que é gerida a variabilidade.

Com esta tese pretende-se estudar empiricamente o modo como, de facto, é gerida a variabilidade destes sistemas visto que existe muito pouca informação sobre como é feita essa gestão. Em particular, pretende-se estudar a viabilidade da extração automática de *feature models* (modelos gráficos que podem ajudar na quantificação da variabilidade) a partir dos ficheiros de configuração de um sistema ROS.

Durante todo o processo de análise conseguiram-se identificar algumas técnicas de gestão de variabilidade. Foi também possível desenvolver uma ferramenta capaz de extrair *feature models* automaticamente, apenas através da análise do código de sistemas ROS. Foram escolhidos cinco sistemas ROS para avaliar a ferramenta desenvolvida, tendo sido possível obter resultados interessantes sobre a variabilidade dos mesmos.

Palavras-chave: configuração, estudo empírico, *feature models*, robótica, ROS, variabilidade

An Empirical Study of Variability in ROS Systems

The use of robotic systems has been growing in the last years and the software of these systems has become increasingly important for their operation. The *Robot Operating System* (ROS) is a middleware that simplifies the implementation of these systems, by providing several primitives that ease the writing of software and the coordinatation of the various components that constitute them. ROS systems are distributed, with an architecture organized in nodes that communicate with each other through message passing. These robotic systems are highly configurable, as they need to adjust to increasingly diverse and adverse work environments. In ROS systems there are files that include the system configuration and it is through these that the variability is thought to be managed.

This thesis intends to empirically study how the variability of these systems is managed, since information about this management is almost nonexistent nowadays. In particular, we intend to study the viability of automatic extraction of *feature models* (graphic models that can help in the quantification of variability) from the configuration files of a ROS system.

During the entire analysis process, it was possible to identify some variability management techniques. It was also possible to develop a tool capable of extracting feature models automatically, just by analyzing the code of ROS systems. Five ROS systems were chosen to evaluate the developed tool, and some interesting results were obtained concerning their variability.

Keywords: configuration, empirical study, feature modules, robotics, ROS, variability

Índice

Lista de Figuras

Lista de Tabelas

Listagens

Siglas

AHEAD	*Algebraic Hierarchical Equations for Application Design*
CIDE	*Colored Integrated Development Environment*
CP	*Construction Primitive*
DSL	*Domain-Specific Language*
EPF	*Eclipse Process Framework*
FAMILIAR	*Feature Model Script Language for Manipulation and Automatic Reasoning*
FreeBSD	*Free Berkeley Software Distribution*
FST	*Feature Structure Tree*
GenArch-P	*Generative Architecture-P*
GPS	*Global Positioning System*
IDE	*Integrated Development Environment*
IMU	*Inertial Measurement Unit*
LiDAR	*Light Detection And Ranging*
Orocos	*Open Robot Control Software*
RAFs	*Ranked All-Features*
RIFs	*Ranked Implied Features*
ROS	*Robot Operating System*
SLAM	*Simultaneous Localization And Mapping*
SPL	*Software Product Line*

TCP *Transmission Control Protocol*
TVL *Text-based Variability Language*

UDP *User Datagram Protocol*
UML *Unified Modeling Language*
URDF *Unified Robot Description Format*
USB *Universal Serial Bus*

XML *Extensible Markup Language*

YARP *Yet Another Robot Platform*

Capítulo 1

Introdução

A robótica está cada vez mais presente na vida das pessoas e pode adaptar-se às mais variadas atividades, desde o auxílio doméstico à execução de tarefas que se consideram perigosas para o ser humano. Esta é uma tecnologia que tem vindo a alcançar bastante sucesso no que diz respeito à redução de custos e ao cumprimento de tarefas com altos níveis de produtividade. Os robôs operam em várias áreas, e para satisfazer as necessidades de cada uma dessas áreas é necessário desenvolver *software* adequado. Assim, pode-se considerar que o desenvolvimento de *software* se está a tornar cada vez mais importante no mundo da robótica. Apesar disso, torna-se também um obstáculo à criação de sistemas robóticos, pois estes utilizam uma grande variedade de componentes, o que dificulta o desenvolvimento e reutilização de *software*.

Existem muitas plataformas de escrita de *software* robótico, sendo que a mais popular é o *Robot Operating System* (ROS) [26]. Esta plataforma tem como principal objetivo simplificar o processo de criação de robôs com comportamentos complexos, através da disponibilização de ferramentas e bibliotecas que facilitam a escrita do *software*. Os sistemas ROS apresentam uma arquitetura distribuída e baseada em nós que comunicam através da passagem de mensagens classificadas em tópicos, sendo que cada nó recebe apenas as mensagens dos tópicos que subscreveu.

Os ambientes em que os sistemas robóticos são utilizados são, por vezes, desafiantes, levando à necessidade de construir robôs robustos e mais complexos. Para essa construção é necessário desenvolver *software* capaz de suportar variabilidade, levando a que o sistema robótico se torne altamente configurável. De forma a ajudar na modelação da variabilidade no processo de desenvolvimento de *software* podem ser usados *feature models* [8], modelos gráficos onde fica evidente quais as características (*features*) suportadas e as dependências entre elas. Estes modelos poderão ser interessantes para quantificar a variabilidade de sistemas de grande escala, com possibilidade de inúmeras características na sua configuração.

Em ROS existem ficheiros (designados *launch files*) que contêm os parâmetros de configuração dos

robôs, nomeadamente quais os nós existentes e a relação entre os tópicos onde publicam ou subscrevem mensagens. A gestão da variabilidade de sistemas ROS aparenta ser, em geral, feita com uma estratégia *ad hoc*, com base nesses ficheiros de configuração. Esta técnica pode tornar difícil a compreensão do espaço de configuração de um sistema ROS, pois estes ficheiros tipicamente contêm características de componentes individuais dos robôs, sendo que, por vezes, nem sequer é perceptível se a utilização de dois *launch files* diferentes é ou não compatível.

A engenharia de *Software Product Lines* (SPL) é cada vez mais utilizada para o desenvolvimento de *software* de maneira mais eficiente em termos de custo, tempo e qualidade. Esta técnica consiste principalmente, na reutilização de componentes quando os produtos apresentam características comuns. Desta maneira consegue-se poupar tempo em desenvolver novos componentes quando estes já existem e conseguem servir o propósito do produto.

Existem estudos apenas sobre variabilidade no processo de desenvolvimento de *software* em geral [12, 17, 30], mas quando se trata da gestão da variabilidade no ecossistema ROS, a informação existente é bastante reduzida. Assim sendo, o objetivo principal desta tese é realizar um estudo empírico sobre como é gerida a variabilidade nos sistemas ROS. Em particular, pretende-se estudar a viabilidade da extração automática de *feature models* a partir dos ficheiros de configuração de um sistema ROS.

1.1 Estrutura do Documento

No presente documento começa-se por uma introdução ao *Robot Operating System*, no Capítulo 2, que é o sistema onde vai estar centrada esta dissertação. É neste capítulo que vão estar todas as informações relevantes para a compreensão do sistema e do que foi necessário para cumprir o objetivo.

Depois dessa introdução avança-se para o desenvolvimento de *software product lines* no Capítulo 3 onde é explicado este conceito, assim como a exposição do tema da variabilidade na robótica e a gestão da mesma.

Depois de se apresentar todos os conceitos relevantes, chega-se ao Capítulo 4, onde se apresentam casos reais de sistemas ROS com variabilidade. Neste capítulo é realizada uma análise detalhada dos sistemas identificados para se perceber melhor como funciona, na prática, a gestão da variabilidade no ROS.

No Capítulo 5 chega-se à parte mais prática, sendo apresentado passo a passo todo o processo de desenvolvimento de uma ferramenta que tenta construir *feature models* de sistemas ROS de forma automática.

Para melhor perceber a precisão da técnica de extração automática de *feature models*, no Capítulo 6 é apresentado um refinamento manual nos resultados obtidos automaticamente, assim como uma análise dos resultados antes e depois desse refinamento.

No Capítulo 7 é apresentada uma pequena conclusão do trabalho feito, assim como os principais contributos desta tese para a comunidade e o possível trabalho futuro.

Capítulo 2

Introdução ao *Robot Operating System*

O *Robot Operating System* (ROS) é um conjunto de bibliotecas e ferramentas que ajudam na construção de aplicações para robôs. O ROS não é um sistema operativo no sentido clássico pois não faz a gestão e programação de processos. Em vez disso, fornece uma camada de comunicação estruturada acima dos sistemas operativos *host* de um sistema de computação heterogéneo.

Os sistemas ROS apresentam uma arquitetura distribuída, baseada em nós que comunicam através da passagem de mensagens classificadas em tópicos. Cada nó tem a possibilidade de subscrever tópicos para depois poder receber apenas mensagens desses mesmos tópicos. Assim é garantida a comunicação entre nós que publicam e subscrevem os mesmos tópicos.

Uma das maiores vantagens desta plataforma é, sem dúvida, a facilidade em utilizar código já criado e publicado por outrem, mesmo que a finalidade do projeto de destino seja diferente da do projeto de origem. Desta maneira, os programadores podem colaborar entre si, eliminando a dificuldade que iria ser sentida ao programar de raiz comportamentos mais complexos.

2.1 Um Pouco de História

Muitas pessoas dentro da comunidade da robótica sentiram a necessidade da existência de uma estrutura de colaboração aberta e foi com esse objetivo que o ROS foi criado. A dificuldade de programar robôs com comportamentos complexos deu também origem à criação desta plataforma.

Na década de 2000 foram criados vários protótipos de sistemas de *software* flexíveis e dinâmicos destinados ao uso da robótica. Em 2007, um laboratório de pesquisa em robótica chamado Willow Garage disponibilizou recursos importantes que ajudaram a expandir esses conceitos e a criar a primeira distribuição do ROS. O *software* foi desenvolvido abertamente e, de forma gradual, tornou-se uma plataforma bastante utilizada dentro da comunidade da robótica.

Mesmo que a ideia de todos os colaboradores colocarem o seu código nos mesmos servidores

parecesse a mais simples, o modelo *"federated"* acabou por ser um dos pontos fortes do ROS. Assim, qualquer equipa pode criar o seu próprio repositório nos seus próprios servidores e manter o controlo total do mesmo, sem precisar da autorização de terceiros. Os criadores dos repositórios podem optar por disponibilizá-los publicamente e assim receber reconhecimento e crédito pelo seu trabalho e contributos na área, podendo também beneficiar de *feedback* técnico e melhorias.

Por todas as qualidades referidas, esta plataforma está a crescer cada vez mais. Neste momento, o ecossistema ROS é constituído por milhares de utilizadores com projetos que vão desde passatempos a grandes sistemas de automatização industrial.

2.2 Distribuições e Instalação

Para começar a entrar no espírito e ambiente do ROS é necessário fazer a instalação do sistema. Para a instalação, configurações iniciais e compreensão do funcionamento foi visitado o *site* oficial do ROS[1] e consultou-se um livro que ajuda nos primeiros passos e comandos mais utilizados [19]. Depois de instalado já se pode aceder à estrutura de ficheiros disponibilizado pelo ROS através de comandos, tais como:

- **`rospack`**: permite obter informação acerca de pacotes ROS;
- **`roscd`**: permite mudar de diretoria diretamente para um pacote ROS;
- **`rosls`**: permite listar os ficheiros de um pacote ROS sem precisar do caminho completo deste.

Depois de conhecer esta (vasta) estrutura de ficheiros disponibilizada pelo ROS, convém conhecer um comando importante e necessário para o funcionamento do ROS - o `roscore`. Com a execução deste comando é ativado o `rosmaster`, um processo cujo objetivo é permitir que os nós se localizem uns aos outros para comunicar, e também disponibilizar o servidor de parâmetros (que vai ser detalhado na Secção 2.3.5). Assim que esse comando estiver em execução, existem comandos que permitem saber mais informação sobre o sistema em execução, especialmente sobre os seus nós e tópicos, conceitos que vão ser abordados mais à frente.

As distribuições[2] dos ROS são um conjunto de pacotes ROS com versões. Até este momento, o ROS já teve muitas versões, sendo a mais recente e utilizada para parte desta dissertação, o *ROS Noetic Ninjemys*. Ainda que as diferentes versões sejam mais orientadas para o sistema operativo Linux, existem funcionalidades para Mac OS X, Android e Windows. A seguir está a lista de algumas dessas distribuições com o respetivo ano de lançamento:

- ***ROS Noetic Ninjemys*** - 2020
- ***ROS Melodic Morenia*** - 2018

[1] `https://www.ros.org/`
[2] `http://wiki.ros.org/Distributions`

- ***ROS Lunar Loggerhead*** - 2017
- ***ROS Kinetic Kame*** - 2016
- ***ROS Jade Turtle*** - 2015

As versões identificadas não constituem o total de versões do ROS, estas são apenas as 5 mais recentes, pois este sistema conta com um total de 13 distribuições diferentes.

2.3 Principais Componentes

Aqui vão ser descritos os principais componentes presentes num sistema ROS, a infraestrutura de comunicação e características específicas dos robôs. Como componentes principais deste sistema podem-se identificar os nós, os tópicos, as mensagens, os parâmetros e os serviços. Todos estes componentes vão ser detalhadamente expostos a seguir com a ajuda de imagens elucidativas.

2.3.1 Nós

Como já foi referido anteriormente, os sistemas ROS são constituídos por nós[3] que podem ser vistos como processos que, combinados com os tópicos, parâmetros e serviços, formam um grafo de computação. Para que tudo funcione da melhor maneira, estes nós devem operar num domínio específico, ou seja, cada nó deve servir para um fim específico, levando a que um sistema contenha um grande conjunto de nós. A utilização de nós proporciona um conjunto de vantagens, como por exemplo:

- Maior tolerância a falhas;
- Redução da complexidade do código;
- Ocultação de detalhes de implementação.

Para além destas vantagens, os nós trazem também alguns atributos que acabam por facilitar a compreensão do código e do sistema. Dois desses atributos são o nome e o tipo. Os nós em execução têm um nome de recurso[4] que os identifica para o resto do sistema. Cada recurso (nós, tópicos, parâmetros e serviços) é definido num *namespace*, que pode ser partilhado com outros recursos.

Os nomes de recurso são a melhor opção para lidar com a colisão de nomes e existem 4 tipos de nomes: base, relativo, global e privado. Nomes que começam com "/" são globais e fazem sentido onde quer que sejam utilizados, têm significados não ambíguos e não precisam de nenhuma informação adicional de contexto para se perceber a que recurso se está a referir. Quando um nome começa por "~" é um nome privado. Estes não especificam totalmente o *namespace* onde se encontram e por isso utilizam o nome do nó como *namespace*. Os recursos com nomes deste tipo continuam a ser acessíveis através do

[3] http://wiki.ros.org/Nodes
[4] http://wiki.ros.org/Names

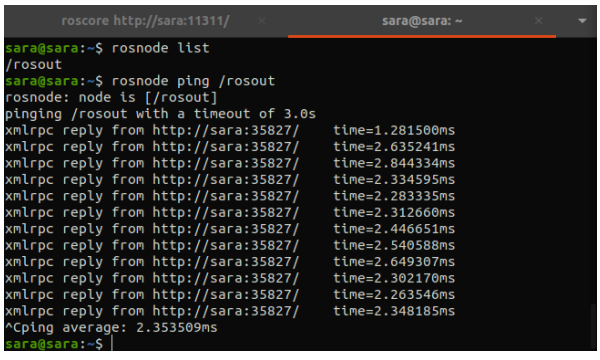
seu nome global tal como os outros recursos. Um nome diz-se relativo quando a sua aparência é do tipo `relativo/nome`. Para resolver este tipo de nome o ROS utiliza o *namespace default* atual e junta-lhe o nome de recurso. No caso do exemplo anterior, se o *namespace default* for `/namespace_default` ficaria `/namespace_default/relativo/nome`. Quanto aos nomes de base, descrevem o nome do próprio recurso e são representados apenas pelo nome, sem utilização de "/" nem "~".

Para além do nome, os nós possuem também um tipo, o que simplifica o processo de referência de um nó executável. Este tipo serve para a implementação do nó, ou seja, contém o nome do pacote do nó e ainda o nome do ficheiro executável do nó. Com isto é importante que dentro do mesmo pacote não existam nomes de executáveis iguais. Assim, é possível executar várias instâncias do mesmo tipo mas com nomes diferentes.

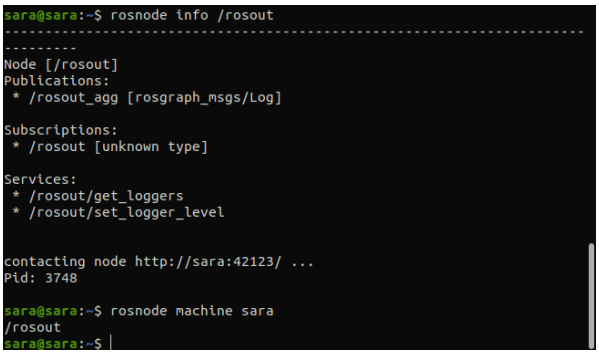
Depois de ter o comando `roscore` a correr pode-se obter informação dos nós que estão a executar naquele momento com o comando `rosnode`. Algumas das opções disponíveis para este comando são:

- **`rosnode ping`**: testa a conectividade para um determinado nó;
- **`rosnode list`**: lista os nós que estão ativos no momento;
- **`rosnode info`**: mostra no ecrã a informação de um determinado nó;
- **`rosnode machine`**: lista os nós que estão a correr numa determinada máquina;
- **`rosnode kill`**: termina um nó em execução.

Agora que foram explicados os comandos possíveis a partir do `rosnode` vai ser apresentado um exemplo quando apenas se tem o `roscore` em execução.



(a) Utilização do `rosnode list` e `rosnode ping`.



(b) Utilização do `rosnode info` e `rosnode machine`.

Figura 2.1: Informação dos nós com recurso ao `rosnode`.

Como se pode observar pela Figura 2.1a, quando apenas o `roscore` está em execução, o único nó que está ativo é o `/rosout`. Este nó acaba por ser o equivalente do ROS ao `stdout/stderr`. A partir da mesma imagem pode-se concluir que existe conetividade para este nó, informação que pode ser retirada do `ping`. As informações que se pode retirar da Figura 2.1b são as suas publicações, subscrições e os serviços que estão disponíveis para esse nó. Ainda na mesma imagem é possível verificar quais os nós

que estão a correr na máquina "sara", neste caso é apenas o `/rosout` pois também é o único nó que se encontra ativo de momento.

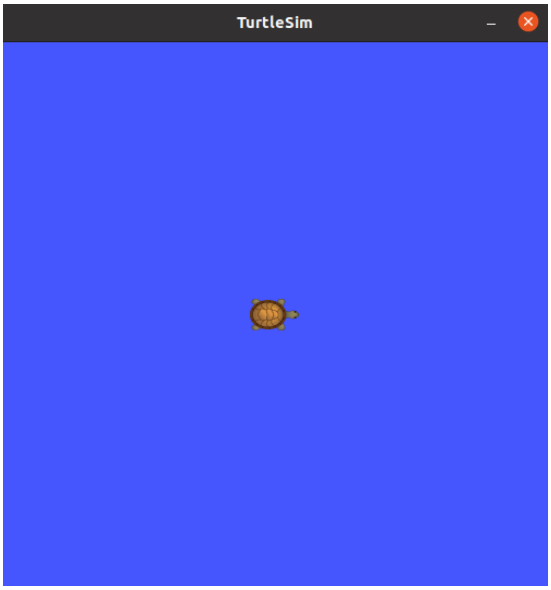
2.3.1.1 *Turtlesim*

O *turtlesim* é uma ferramenta pertencente ao ROS muito utilizada para iniciação e aprendizagem. Esta ferramenta contém um nó que funciona como um simulador de uma tartaruga com a qual se pode praticar, o que ajuda na aprendizagem de comandos e pacotes do ROS.

Para correr um nó basta utilizar o comando `rosrun` com o nome do pacote onde esse nó se encontra e com o nome do nó. Por exemplo, para correr o *turtlesim*[5] tem que se utilizar o comando `rosrun turtlesim turtlesim_node`. Este comando vai colocar esse nó a correr e quando se executar o comando `rosnode list` já vão existir dois nós ativos. Na Figura 2.2a podem-se observar as informações acerca do *turtlesim* que são disponibilizadas pelo ROS. Este é um nó mais complexo do que o `rosout`: publica em três tópicos (que vão ser explicados mais à frente), subscreve apenas um tópico e disponibiliza bastantes serviços (estes serviços também vão ser abordados e mais detalhados em capítulos posteriores). Quando este nó é executado abre também uma janela que se mantém aberta enquanto o *turtlesim* está a correr (Figura 2.2b). É nesta janela que podem ser observadas as mudanças que irão ocorrer na tartaruga.

```
sara@sara:~$ rosnode list
/rosout
/turtlesim
sara@sara:~$ rosnode info /turtlesim
--------------------------------------------------------------------------------
Node [/turtlesim]
Publications:
 * /rosout [rosgraph_msgs/Log]
 * /turtle1/color_sensor [turtlesim/Color]
 * /turtle1/pose [turtlesim/Pose]

Subscriptions:
 * /turtle1/cmd_vel [unknown type]

Services:
 * /clear
 * /kill
 * /reset
 * /spawn
 * /turtle1/set_pen
 * /turtle1/teleport_absolute
 * /turtle1/teleport_relative
 * /turtlesim/get_loggers
 * /turtlesim/set_logger_level


contacting node http://sara:42809/ ...
Pid: 6998
Connections:
 * topic: /rosout
    * to: /rosout
    * direction: outbound (58003 - 127.0.0.1:36874) [24]
    * transport: TCPROS
```

(a) Informações do nó *turtlesim*.



(b) Representação do nó *turtlesim*.

Figura 2.2: Nó *turtlesim*.

2.3.2 Tópicos

Os tópicos[6] já foram referidos anteriormente, mas ainda não foi dada uma definição concreta dos mesmos. Aqui vai ser tratado com mais detalhe o termo "tópico", assim como as suas características e comandos

[5] http://wiki.ros.org/turtlesim

[6] http://wiki.ros.org/Topics

relacionados. Assim sendo, um tópico pode ser definido como um espaço onde os nós podem publicar mensagens e a partir do qual podem receber mensagens se estiverem subscritos nesse mesmo tópico. Um nó pode subscrever um tópico se pretender receber mensagens daquele tópico, por isso pode-se dizer que um tópico é o meio de transporte para comunicação entre nós, visto que transportam as mensagens, e as mensagens são o modo de comunicação entre os nós. Um nó apenas recebe as mensagens dos tópicos que subscreve, mas pode publicar em tópicos que não subscreve.

Para transportar as mensagens, o ROS suporta dois tipos de protocolo, o *TCP* e o *UDP*. O *TCP* é o protocolo padrão em ROS, enquanto que o *UDP* apenas é suportado pelo pacote `roscpp`, que é uma implementação do ROS em C++, sendo que é a biblioteca mais utilizada e foi projetado para ser de alto desempenho. O *UDP* é um transporte que apresenta latências e perdas baixas, sendo por isso, o mais adequado a tarefas do tipo das tele-operações. É necessário que um tópico possua protocolo de transporte para que possa enviar as mensagens para os nós que os subscrevem, e assim garantir a comunicação entre os vários nós.

Tal como os nós, para se saber informações sobre os tópicos é necessário que o `roscore` esteja em execução. Para conseguir essa informação, existe o comando `rostopic` que, tal como para os nós, disponibiliza várias opções:

- **`rostopic bw`**: mostra a largura de banda utilizada por um determinado tópico;
- **`rostopic delay`**: mostra o atraso do tópico a partir da hora no cabeçalho, caso possua;
- **`rostopic echo`**: mostra as mensagens no ecrã;
- **`rostopic find`**: encontra tópicos a partir do tipo;
- **`rostopic hz`**: mostra a taxa de publicação de um determinado tópico;
- **`rostopic info`**: mostra no ecrã a informação de um determinado tópico ativo;
- **`rostopic list`**: lista os tópicos que estão ativos no momento;
- **`rostopic pub`**: publica dados num determinado tópico;
- **`rostopic type`**: mostra o tipo de um determinado tópico.

Com os nós `roscore` e o *turtlesim* em execução vão agora ser exibidos alguns exemplos da utilização dos comandos relativos aos tópicos.

```
sara@sara:~$ rostopic list
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
sara@sara:~$ rostopic info /turtle1/pose
Type: turtlesim/Pose

Publishers:
 * /turtlesim (http://sara:43383/)

Subscribers: None


sara@sara:~$ rostopic type /turtle1/pose
turtlesim/Pose
sara@sara:~$
```

(a) Utilização do `rostopic list`, `rostopic info` e `rostopic type`.

```
sara@sara:~$ rostopic bw /turtle1/pose
subscribed to [/turtle1/pose]
average: 1.04KB/s
        mean: 0.02KB min: 0.02KB max: 0.02KB window: 51
average: 1.06KB/s
        mean: 0.02KB min: 0.02KB max: 0.02KB window: 100
average: 1.16KB/s
        mean: 0.02KB min: 0.02KB max: 0.02KB window: 100
average: 1.12KB/s
        mean: 0.02KB min: 0.02KB max: 0.02KB window: 100
average: 1.11KB/s
        mean: 0.02KB min: 0.02KB max: 0.02KB window: 100
^Caverage: 840.11B/s
        mean: 20.00B min: 20.00B max: 20.00B window: 100
sara@sara:~$ rostopic find turtlesim/Pose
/turtle1/pose
sara@sara:~$
```

(b) Utilização do `rostopic bw` e `rostopic find`.

```
sara@sara:~$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist --
'[4.0, 0.0, 0.0]' '[0.0, 0.0, 2.0]'
publishing and latching message for 3.0 seconds
sara@sara:~$
```

(c) Utilização do `rostopic pub`.

```
sara@sara:~$ rostopic echo /turtle1/pose
x: 6.61053466796875
y: 6.907934188842773
theta: 1.785599946975708
linear_velocity: 0.0
angular_velocity: 0.0
---
x: 6.61053466796875
y: 6.907934188842773
theta: 1.785599946975708
linear_velocity: 0.0
angular_velocity: 0.0
```

(d) Utilização do `rostopic echo`.

Figura 2.3: Utilização de sub-comandos do `rostopic`.

Como é possível observar pela Figura 2.3a existem 5 tópicos ativos no momento, um deles é o `/rosout` que é o tópico padrão para publicar mensagens de *logging*. Além deste existe também o `/rosout_agg` que acaba por ser um tópico agregado para a subscrição das mensagens de `log`. Este tópico fica ativo quando se executa o `roscore` e tem a vantagem de poder receber as mensagens diretamente do nó `rosout`. Os seguintes tópicos ativos já são referentes ao nó `/turtlesim`. O `/turtle1/cmd_vel` corresponde à velocidade da tartaruga, o `/turtle1/color_sensor` refere-se à cor do sítio onde a tartaruga se encontra, ou seja onde a tartaruga está a "pisar" e o `/turtle1/pose` está relacionado com a posição da tartaruga. Ainda na mesma figura tem a informação de um dos tópicos ativos, o `/turtle1/pose`, nomeadamente o tipo do tópico, os nós que publicam nesse tópico e os nós que subscrevem esse tópico. Depois disso pode-se ver qual o comando que permite identificar o tipo de um determinado tópico e como se utiliza. Este tipo é definido pelo tipo das mensagens que suporta, ou seja, o tipo de mensagens que o tópico transporta. De referir que o tipo de um tópico começa sempre com letra maiúscula.

Na segunda figura (2.3b) está representada a forma de utilização e o resultado do comando que permite ver a largura de banda utilizada por um tópico, neste caso o exemplo foi feito com o tópico que já foi utilizado anteriormente (`/turtle1/pose`). Ainda na mesma figura, pode-se também encontrar os tópicos que são de um determinado tipo.

Na Figura 2.3c está um exemplo de uma publicação num tópico e o resultado dessa publicação.

Como se pode observar, fazer uma publicação no tópico que controla a velocidade da tartaruga resulta num movimento da mesma. O comando `rostopic pub` requer alguns argumentos: o nome do tópico, o tipo do tópico e os argumentos que vão fazer alterar o estado da tartaruga. Estes argumentos vão ser melhor compreendidos mais à frente pois estão relacionados com as mensagens que são passadas aos tópicos, mas neste caso é dada uma velocidade linear e angular, em que cada uma é representada por um conjunto de 3 *floats*. Dentro das duas velocidades o formato dos dados é [x, y, z]. Visto que este exemplo apenas funciona em 2D, quando o campo 'z' da velocidade linear é diferente de zero o simulador não vai apresentar resultados. Quando se olha para o caso da velocidade angular, a tartaruga apenas reproduz a rotação quando esta é passada no campo 'z'. A opção '-1' serve para que apenas seja feita uma instância da publicação.

Em relação à ultima figura (2.3d) tem-se a utilização do comando que permite verificar quais as mensagens que estão a ser passadas a um determinado tópico. No caso da figura apresentada pode-se observar a posição atual da tartaruga – ponto ≈(6.6, 6.9) – com rotação de ≈1.79 (note-se que este valor é representado em radianos). Deste comando pode-se também retirar informações acerca das velocidades linear e angular, sendo que neste caso estão as duas com o valor zero, o que significa que a tartaruga está parada, ou seja, esta informação foi retirada já depois de a tartaruga ter processado o comando de velocidade enviado para o tópico `/turtle1/cmd_vel`.

2.3.3 Mensagens

Para que seja possível a comunicação entre os diferentes nós é necessário que exista passagem de mensagens. As mensagens[7] estão diretamente relacionadas com os tópicos pois é através destes que são passadas. Existem alguns comandos que se pode usar para saber mais informação sobre as mensagens tais como:

- **`rosmsg show`**: mostra a descrição de uma mensagem a partir do tópico;
- **`rosmsg info`**: faz o mesmo que o anterior, é apenas outra maneira de invocar;
- **`rosmsg list`**: lista todas as mensagens;
- **`rosmsg package`**: lista as mensagens num determinado pacote;
- **`rosmsg packages`**: lista os pacotes que contêm mensagens.

Já foram referidos anteriormente quais os tópicos que estão disponíveis quando o *turtlesim* está em execução, por isso vão ser utilizados esses tópicos para exemplificar alguns destes comandos.

Na Figura 2.4a pode-se observar quais os pacotes ROS que contém mensagens e facilmente localizar um pacote já conhecido, o *turtlesim*. Outro dos pacotes que também acaba por surgir quando se fala em

[7] `http://wiki.ros.org/Messages`

turtlesim é o `geometry_msgs` que contém o tipo de mensagem `geometry_msgs/Twist` que serve para representar as velocidades linear e angular da tartaruga.

No exemplo da Figura 2.4b pode-se ver quais as mensagens que estão dentro de um pacote e a descrição das mesmas. Neste caso utilizou-se o pacote *turtlesim*, sendo que dentro deste existem dois tópicos com mensagens, a cor da janela e a posição da tartaruga. Para conseguir informação acerca da mensagem, tal como os argumentos que a constituem, existe o comando `show`. No caso do `turtlesim/Pose` pode ver-se que contém a posição (x, y) da tartaruga, o ângulo e as velocidades linear e angular da tartaruga.

```
sara@sara:~$ rosmsg packages
actionlib
actionlib_msgs
actionlib_tutorials
bond
control_msgs
controller_manager_msgs
diagnostic_msgs
dynamic_reconfigure
gazebo_msgs
geometry_msgs
map_msgs
nav_msgs
pcl_msgs
roscpp
rosgraph_msgs
rospy_tutorials
sensor_msgs
shape_msgs
smach_msgs
std_msgs
stereo_msgs
tf
tf2_msgs
theora_image_transport
trajectory_msgs
turtle_actionlib
turtlesim
visualization_msgs
sara@sara:~$
```

(a) Utilização do `rosmsg packages`.

```
sara@sara:~$ rosmsg package turtlesim
turtlesim/Color
turtlesim/Pose
sara@sara:~$ rosmsg show turtlesim/Pose
float32 x
float32 y
float32 theta
float32 linear_velocity
float32 angular_velocity

sara@sara:~$
```

(b) Utilização do `rosmsg package` e do `rosmsg show`.

Figura 2.4: Utilização de sub-comandos do `rosmsg`.

2.3.4 Serviços

Os serviços[8] são outro meio que os nós podem usar para comunicar entre si. Os serviços permitem que os nós enviem pedidos e recebam respostas. Para utilizar os serviços do ROS existem, tal como para os outros componentes, alguns comandos dos quais qualquer utilizador se pode servir. Esses comandos são:

- **`rosservice args`**: mostra os argumentos de um determinado serviço;
- **`rosservice call`**: chama um determinado serviço com os respetivos argumentos;

[8] http://wiki.ros.org/Services

- **`rosservice find`**: encontra serviços com um determinado tipo;
- **`rosservice info`**: mostra informação de um determinado serviço;
- **`rosservice list`**: lista todos os serviços ativos no momento;
- **`rosservice type`**: mostra o tipo de um determinado serviço.

Os comandos anteriores vão ser agora exemplificados de maneira a que seja percetível a sua utilização.

```
sara@sara:~$ rosservice list
/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/spawn
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
sara@sara:~$ rosservice args /spawn
x y theta name
sara@sara:~$
```

(a) Utilização do `rosservice list` e do `rosservice args`.

```
sara@sara:~$ rosservice info /spawn
Node: /turtlesim
URI: rosrpc://sara:40099
Type: turtlesim/Spawn
Args: x y theta name
sara@sara:~$ rosservice type /turtlesim/get_loggers
roscpp/GetLoggers
sara@sara:~$ rosservice find roscpp/GetLoggers
/rosout/get_loggers
/turtlesim/get_loggers
sara@sara:~$
```

(b) Utilização do `rosservice info`, `rosservice type` e do `rosservice find`.

Figura 2.5: Utilização de sub-comandos do `rosservice`.

Na Figura 2.5a pode-se observar o resultado do comando `rosservice list`, a partir deste pode-se verificar que existem alguns serviços disponíveis no momento:

- **`/clear`**: apaga todos os traços de movimento da janela do *turtlesim*, se a tartaruga não tiver feito movimentos este comando não mostra resultados;
- **`/kill`**: mata uma determinada tartaruga, fazendo-a desaparecer da janela;
- **`/reset`**: faz *reset* à janela do *turtlesim*, ou seja, volta a colocar a tartaruga na posição inicial, com o ângulo inicial, fazendo também desaparecer os traços de movimento;
- **`/spawn`**: coloca uma nova tartaruga numa determinada posição, com um determinado ângulo na janela do *turtlesim*;
- **`/turtle1/set_pen`**: muda o traço que a tartaruga deixa quando se move, podendo ser mudada a cor (r, g, b) e a largura do traço. Com este comando pode-se também desativar o traço, ficando invisível;
- **`/turtle1/teleport_absolute`**: teletransporta a tartaruga para uma determinada posição e com um determinado ângulo;

- **`/turtle1/teleport_relative`**: teletransporta a tartaruga com uma determinada distância e ângulo relativamente à posição e ângulo em que se encontra.

Com a Figura 2.5b pode-se examinar toda a informação relativa ao serviço `/spawn`, assim como outras informações, tais como: o tipo do serviço `/turtlesim/get_loggers`, que se pode ver que é do tipo `roscpp/GetLoggers` e ainda quais os serviços ativos desse mesmo tipo, que tal como era de esperar contém o `get_loggers`. Desta forma é possível encontrar os vários serviços existentes e ativos de um determinado tipo.

Para utilizar alguns dos serviços mostrados na Figura 2.5a é necessário conhecer mais a fundo o *turtlesim*, mais concretamente a janela onde a tartaruga se desloca. Esta janela pode ser vista como um referencial cartesiano com mínimo e máximo como sendo 0 e $\approx$11 respetivamente, o ponto (0, 0) situa-se no canto inferior esquerdo e o (11, 11) no canto superior direito. Sempre que o *turtlesim* é iniciado, a tartaruga situa-se no ponto aproximado (5.54, 5.54). Como já foi referido anteriormente, o grau que caracteriza a direção da tartaruga tem como unidade de medida o radiano, sendo que inicialmente a tartaruga apresenta 0 radianos, ou seja, direcionada para a direita. Na Figura 2.6 está representada a janela com os seus limites e a posição inicial da tartaruga.

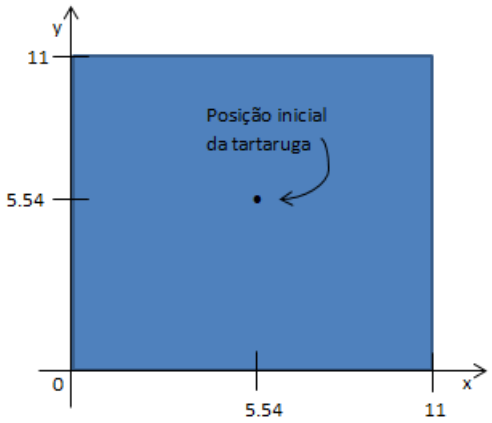
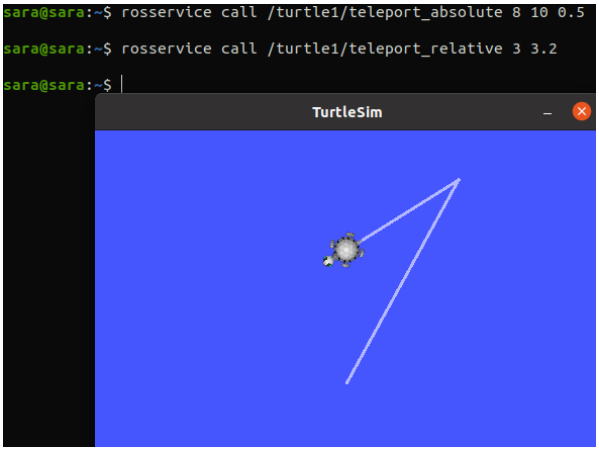


Figura 2.6: Referencial da janela do *turtlesim*.

Nas Figuras 2.7a e 2.7b pode-se ver como utilizar o `rosservice call` com alguns dos serviços mostrados anteriormente.

(a) Invocação do serviço `/spawn`.



(b) Invocação dos serviços `/turtle1/teleport_absolute` e `/turtle1/teleport_relative`.

Figura 2.7: Invocação de serviços com o `rosservice call`.

Na Figura 2.7a está ilustrada a utilização do `rosservice call /spawn`. Como se pode ver apareceu uma nova tartaruga na posição (4, 10) da janela, com um ângulo de 1.57 radianos e com o nome *turtle2*. Este serviço devolve o nome da tartaruga criada.

Já na Figura 2.7b está ilustrada a utilização dos serviços de teletransporte. No primeiro comando os argumentos são a nova posição (x, y) e a nova direção. Segundo o referencial já apresentado pode-se concluir que o traço formado na diagonal da esquerda para a direita é correspondente ao primeiro comando. Em relação ao segundo comando sabe-se que os seus argumentos são a distância e a direção que a tartaruga deve percorrer e seguir. Neste caso percorreu uma distância de 3 unidades em relação ao ponto onde se encontrava na direção de um ângulo de 3.2 radianos desde a direção em que já se encontrava.

2.3.5 Parâmetros

Os parâmetros[9] são componentes do ROS localizados no *Parameter Server*. Este servidor pode ser visto como um dicionário partilhado que os nós podem utilizar para armazenar, manipular e recuperar parâmetros em tempo de execução. Tal como para os componentes anteriores, existem alguns comandos disponíveis para manipular parâmetros:

- **`rosparam set`**: define um novo valor para um determinado parâmetro;
- **`rosparam get`**: encontra e mostra o valor de um determinado parâmetro;
- **`rosparam load`**: carrega parâmetros a partir de um determinado ficheiro;
- **`rosparam dump`**: descarrega parâmetros para um determinado ficheiro;

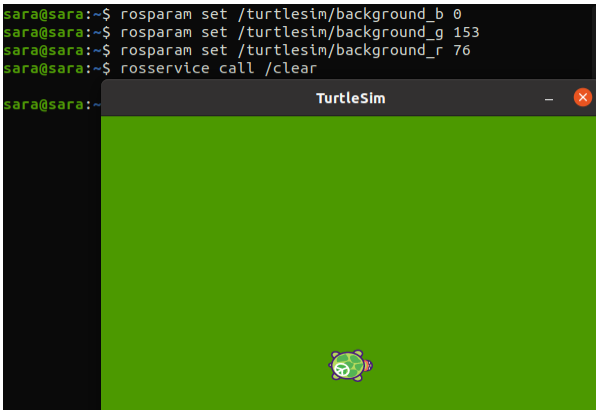
[9] http://wiki.ros.org/Parameter%20Server

- **`rosparam delete`**: apaga um determinado parâmetro;
- **`rosparam list`**: lista os nomes dos parâmetros disponíveis.

Os comandos mais utilizados para tratar de parâmetros no ROS são o `set` e o `get`, sendo que são estes que permitem ver e alterar os valores dos parâmetros. De seguida estão apresentados alguns exemplos do resultado destes comandos.

```
sara@sara:~$ rosparam list
/rosdistro
/roslaunch/uris/host_sara__41817
/rosversion
/run_id
/turtlesim/background_b
/turtlesim/background_g
/turtlesim/background_r
sara@sara:~$ rosparam get /rosdistro
'noetic

  '


sara@sara:~$ rosparam get /roslaunch/uris/host_sara__41817
http://sara:41817/
sara@sara:~$ rosparam get /rosversion
'1.15.8

  '


sara@sara:~$ rosparam get /run_id
1ff7b24e-30a7-11eb-a535-752a30962ff5
sara@sara:~$ rosparam get /turtlesim/background_b
255
sara@sara:~$ rosparam get /turtlesim/background_g
86
sara@sara:~$ rosparam get /turtlesim/background_r
69
sara@sara:~$
```

(a) Utilização do `rosparam list` e do `rosparam get`.



(b) Utilização do `rosparam set` e o seu resultado.

Figura 2.8: Utilização de sub-comandos do `rosparam`.

Na Figura 2.8a está demonstrado o `rosparam list` para se saber quais os parâmetros que estão disponíveis no momento, neste caso existem 7 parâmetros sendo que 3 deles correspondem à cor do *background* do *turtlesim*, pois quando o comando foi executado o *turtlesim* estava em execução. Os primeiros 4 parâmetros ficam disponíveis assim que se inicia o ROS, o primeiro corresponde à distribuição ROS que está a ser utilizada, o segundo corresponde ao servidor, o terceiro à versão do ROS e o quarto corresponde ao *id* de execução. Ainda na mesma figura estão todos os valores de cada um dos parâmetros, para isso foram realizados 7 comandos `rosparam get` com o nome do parâmetro, mas o mesmo poderia ter sido feito com o comando `rosparam get /`, assim eram listados os valores de todos os parâmetros disponíveis de uma só vez.

Na Figura 2.8b está representada a utilização do `rosparam set` de maneira a modificar a cor do *background*. Para isso atribui-se um valor a cada um dos parâmetros, depois disso a cor não é automaticamente alterada, para que a alteração seja visível na janela do *turtlesim* tem que se chamar um serviço já apresentado anteriormente (`/clear`). Apesar de não ser visível na janela, os valores são alterados logo que se executa o `set`.

2.4 *Launch Files*

Os *launch files* são muito comuns em sistemas ROS. Estes ficheiros utilizam a linguagem *Extensible Markup Language* (XML) para configurar o sistema, nomeadamente tornar a inicialização de nós mais fácil e conveniente. A partir deles é também possível definir outros requisitos de inicialização, tal como a configuração dos parâmetros. Nesta secção vai ser demonstrado um exemplo simples de um *launch file*, vão ser detalhados todos os campos e discutidos os resultados do *launch* desse mesmo ficheiro. O código na Listagem 2.1 mostra o conteúdo do *launch file* criado para o exemplo.

```
1  <launch>
2     <node pkg=''turtlesim'' name=''turtle1'' type=''turtlesim_node''>
3
4     <param name=''/background_r'' value=''64''/>
5     <param name=''/background_g'' value=''64''/>
6     <param name=''/background_b'' value=''64''/>
7
8     </node>
9
10    <node pkg=''turtlesim'' name=''teleop_key'' type=''turtle_teleop_key''/>
11 </launch>
```

Listagem 2.1: Exemplo de um *launch file*.

Um ficheiro *launch* tem, necessariamente que começar com a *tag* "`<launch>`" para ser identificado como um *launch file*. Depois desta *tag* foi criado um nó: dentro da respetiva *tag* colocou-se o pacote a que pertence, o nome que se pretende dar ao nó e ainda o tipo do mesmo. Neste caso pretende-se que o *turtlesim* execute com a cor da janela diferente do habitual. Para que isto aconteça o valor dos parâmetros que correspondem à cor do *background* têm que ser alterados, isto ainda dentro da *tag* "`<node>`".

Neste *launch file* é ainda inicializado mais um nó bastante interessante, o *teleop*[10], que permite a movimentação da tartaruga a partir do teclado. Esse nó pertence ao pacote do *turtlesim* e o seu tipo é `turtle_teleop_key`.

Depois de configurar todos os nós que se pretende executar fecha-se a *tag* do ficheiro *launch*. Para fazer o *launch* de um ficheiro deste tipo o comando é o seguinte:

```
$ roslaunch <pacote_do_ficheiro> <nome_do_ficheiro.launch>
```

Na Figura 2.9 é possível observar a utilização desse comando para o exemplo apresentado, bem como o seu resultado.

[10] http://wiki.ros.org/teleop_twist_keyboard

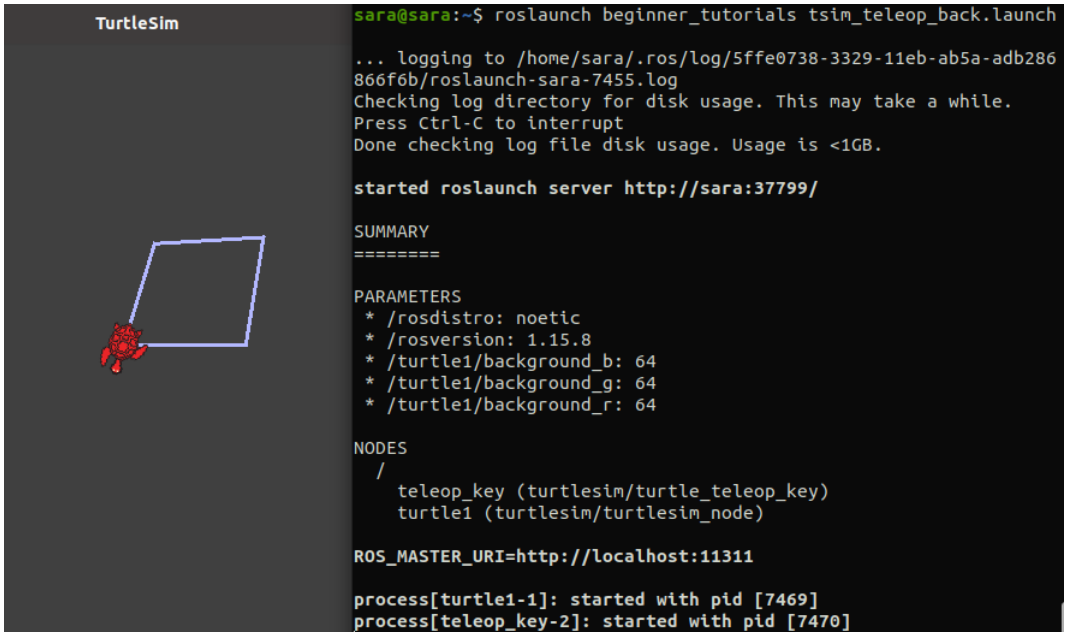


Figura 2.9: Utilização do `roslaunch` e resultado.

Como se pode verificar, o *turtlesim* arrancou com uma janela de cor diferente do normal, a cor definida no *launch file*. Além disso é possível movimentar a tartaruga com o teclado, em que as teclas para cima e para baixo correspondem ao movimento para a frente e para trás, respetivamente e as teclas dos lados correspondem à direção. Isto é possível pois o *turtlesim* e o *teleop* comunicam a partir de um tópico, o *teleop* publica as teclas premidas nesse tópico e o *turtlesim*, subscreve esse tópico e recebe as mensagens, que neste caso são as teclas premidas.

Capítulo 3

Variabilidade em *Software Product Lines*

Uma *Software Product Line* (SPL) [5] – linha de produtos de *software* – é um conjunto de sistemas que partilham características para satisfazer necessidades específicas de um determinado segmento de mercado, e são normalmente desenvolvidos a partir de uma mesma base. Neste sentido é importante perceber o que é um produto e como se definem as características de cada um. Em SPLs, um produto acaba por ser o *software* final desenvolvido e as características (*features*) podem ser entendidas como diferentes funcionalidades desses produtos, e que podem ser reaproveitadas para outros que necessitem dessas funcionalidades. Assim que uma linha de produtos é criada, cada produto acaba por ser uma variante de outro, pois como já foi referido, estes são desenvolvidos a partir da mesma base e apenas variam em algumas características.

Um dos desafios mais difíceis na engenharia de sistemas robóticos passa por conseguir escolher um conjunto de componentes coerente, de maneira a cumprir com os requisitos exigidos e ao mesmo tempo ter em consideração as dependências que existem nesses componentes. Em várias áreas de aplicação, o desenvolvimento de SPLs provou ser o procedimento mais eficaz para responder a esse tipo de desafios.

3.1 Variabilidade em Robótica

No caso de sistemas robóticos a variabilidade é associada às diferentes configurações dos robôs, pois cada um deles pode ser desenvolvido para atividades distintas, ou seja, com diferentes componentes de *software*. O facto de estes sistemas exigirem tanta variabilidade torna necessária a utilização de SPLs, pois permite a partilha de recursos para atingir um determinado objetivo partindo da mesma base, havendo a possibilidade de divergir, mantendo as dependências controladas.

3.1.1 Origem da Variabilidade

Uma fonte de variabilidade é qualquer ação, motivação ou objeto que torne um sistema variável, ou seja, que o force a mudar para que se adapte. Existem várias origens da variabilidade em robótica, umas mais evidentes do que outras, o certo é que todas contribuem para que o desenvolvimento dos sistemas robóticos se torne mais complicado e desafiante. De acordo com um estudo acerca dos desafios da modelação da variabilidade em robótica [18], existem 4 fatores principais que originam variabilidade nestes sistemas:

- Requisitos do cliente. Os clientes apresentam necessidades e preferências diferentes entre si, o que resulta em componentes de *hardware* e *software* diferentes. Esta diferença de componentes é o que leva à diversidade de sistemas, ou seja, à variabilidade. Dependendo da escolha dos clientes, o mesmo robô pode conter diferentes componentes, resultando em diferentes funções, sendo este um dos mais influentes fatores causadores de variabilidade.

- Meio ambiente. Os sistemas robóticos têm que ser adequados ao ambiente onde vão operar. Para isso precisam de estar preparados para obstáculos e configuração dos objetos com os quais vão manter contacto ou que podem encontrar. Tudo isto é a variabilidade do meio ambiente que o sistema pode encontrar e que tem que ultrapassar para que o seu trabalho seja bem sucedido.

- Componentes do robô. Estes acabam por estar ligados aos requisitos do cliente, pois dependendo da função do robô, a sua configuração muda e, consequentemente, mudam os componentes usados na sua construção. Mesmo que a base do sistema seja a mesma, se o seu propósito for minimamente diferente, os seus equipamentos já não vão ser exatamente os mesmos. Assim, pode-se também considerar que esta é mais uma relevante fonte de variabilidade em sistemas robóticos.

- *Middleware.* O que torna um *middleware* um fator causador de variabilidade neste tipo de sistemas é o número de APIs diferentes que os programadores têm de utilizar para implementar toda a variabilidade de *software* que exigem. Atualmente já existem alguns *frameworks* que têm como objetivo ajudar no desenvolvimento de sistemas robóticos, como é o caso do ROS, já detalhado anteriormente, do Orocos[1] (*Open Robot Control Software*) ou do YARP[2] (*Yet Another Robot Platform*).

Estas são as fontes principais da variabilidade em sistemas robóticos, sendo que existem outras, tais como a forma como o robô interage com o meio, que acabam por estar diretamente ligadas com as que foram descritas acima.

[1] https://orocos.org/
[2] https://www.yarp.it/git-master/

3.2 Modelos de Variabilidade

Como se vai detalhar na secção seguinte, para implementar a variabilidade são utilizadas inúmeras estratégias, que vão diferindo dependendo do tipo de sistemas que se pretende implementar, sendo que algumas estratégias são mais populares que outras.

Para a modelação de variabilidade também existem várias estratégias. Algumas serão mencionadas a seguir, mas apenas uma será apresentada com mais detalhe - os *feature models*. Essa mesma estratégia será mais tarde utilizada para modelar a variabilidade de sistemas robóticos reais. Tal como já foi referido anteriormente, o desenvolvimento de SPLs é bastante eficaz em sistemas com dependências, e para esses também é necessária modelação. De acordo com um estudo sobre a modelação de variabilidade [9], o método mais utilizado para modelar a variabilidade são os *feature models*, seguidos de *spreadsheets*, pares chave-valor, *domain-specific languages* (DSLs) e ainda diagramas UML (*Unified Modeling Language*).

Os *feature models* são uma forma de representar resumidamente todos os produtos de uma SPL, onde ficam claras as dependências e possibilidades inerentes a cada *feature*. Outra das vantagens destes modelos é a quantificação da variabilidade, pois normalmente quanto maior for o modelo, mais variabilidade o sistema apresenta. Para produzir *feature models* existem algumas ferramentas que ajudam na sua construção [15], como é o caso do featureIDE[3] [23] que pode ser visto como um *plugin* do Eclipse[4] utilizado para desenvolvimento de *software* orientado a recursos, que é o caso das SPLs. A partir deste *Integrated Development Environment* (IDE) é possível realizar a análise, modelação e implementação de domínio, análise de requisitos e produção de requisitos.

Um *feature model* é representado como um conjunto de *features*, dispostas hierarquicamente, normalmente como uma árvore e composto por relações entre *features* pai e *features* filho. As relações mais utilizadas estão representadas na Tabela 3.1, assim como a sua interpretação no modelo.

Relações principais:	
Obrigatório	a *feature* filho tem que existir no sistema
Opcional	a existência da *feature* filho no sistema é opcional
Alternativo	apenas se pode escolher uma e uma só *feature* filho
Ou	pelo menos uma das *features* filho tem que ser escolhida

Tabela 3.1: Relações principais entre *features* pai e *features* filho.

Para além destas relações entre *features* pai e *features* filho existem mais duas restrições *cross tree* que são comummente utilizadas:

- A necessita de B: se A for selecionada, implica que B também o seja;
- A exclui B: se A for selecionada B não pode ser, e vice-versa.

[3] https://featureide.github.io/
[4] https://www.eclipse.org/

O *feature model* na Figura 3.1 ilustra os vários requisitos mostrados acima. Este é um exemplo de uma SPL com o *turtlesim*, em que existe a possibilidade de adicionar mais uma tartaruga (`Add_turtle`), movimentar a tartaruga (`Move`), alterar a cor da janela (`BackgroundColor`) e ainda ativar um *safety controller* (`Safety_controller`). Como é possível observar, todas estas *features* são opcionais pois não é necessária a existência de nenhuma delas para que o sistema funcione. Dentro da *feature* `Move` existem ainda três possibilidades de escolha para o movimento da tartaruga, visto que é um grupo com relação "ou". Assim, este movimento pode ser feito seguindo outra tartaruga (`Chase_turtle`), e aqui é necessário que seja adicionada outra tartaruga, como se pode ver pela restrição *cross tree* abaixo do modelo, pode-se mover também a partir da aplicação `teleop_key` que foi vista na Secção 2.4 (`Teleop`) ou ainda a partir de um controlador aleatório que define os movimentos da tartaruga (`Random_controller`). Para o *safety controller* (que limita a tartaruga a operar dentro de um determinado raio), existe ainda a possibilidade de fazer a tartaruga voltar ao centro da janela (`Center`) e de mudar a cor do traço quando esta funcionalidade está ativada (`Pen_color`).

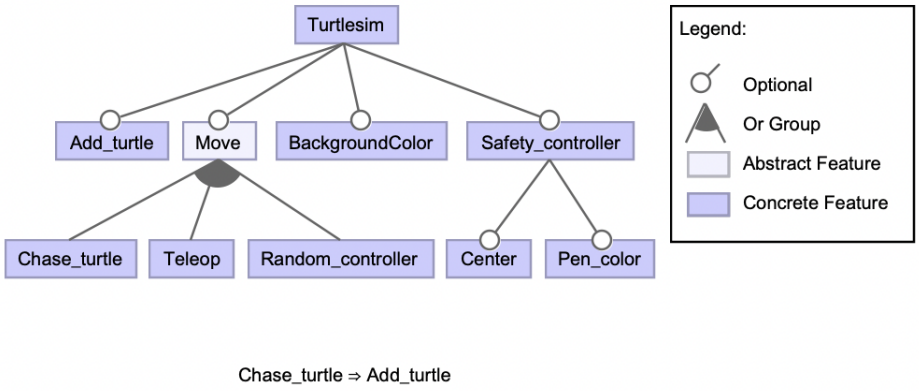


Figura 3.1: Exemplo de um *feature model*.

Tal como referido anteriormente, o featureIDE apresenta bastantes funcionalidades, sendo uma delas a apresentação de muitos resultados e informações relevantes para o desenvolvimento de SPLs, tais como: número de produtos possíveis e quais os produtos válidos e inválidos. No caso do exemplo exposto na Figura 3.1 foi possível observar que existem 120 configurações válidas e visualizar todas essas possibilidade de configuração.

Uma linguagem, bastante interessante que pode ser utilizada também dentro do featureIDE é FAMILIAR[5] (*FeAture Model scrIpt Language for manIpulation and Automatic Reasoning*) [1], que pode servir para importar, exportar, compor, editar, realizar engenharia reversa e testar vários *feature models*.

Para uma melhor visualização e perceção das *features* de um sistema, o mais intuitivo para o ser humano é a construção de modelos visuais tais como os *feature models*. No entanto, para as máquinas, a forma mais fácil de trabalhar é a partir de linguagens que representem esses mesmos sistemas. Para esse efeito existe a linguagem TVL (*Text-based Variability Language*)[6]. Esta linguagem é baseada em texto,

[5] https://familiar-project.github.io/
[6] https://projects.info.unamur.be/tvl/

o que a torna fácil de processar e é indicada para a construção de *feature models*, tal como o que está representado acima. Neste momento existe uma implementação de Java[7] para TVL que permite uma contagem e visualização de configurações válidas de produtos de um determinado modelo. A estrutura de um ficheiro TVL tem algumas regras [10, 11] que é necessário conhecer, tais como:

- O ficheiro inicia sempre com a *feature* raiz, sendo que é colocada uma *tag* `root` antes do nome da *feature*;
- Os nomes das *features* podem conter letras, dígitos e o símbolo *underscore* '_';
- Todas as *features* iniciam com letra maiúscula, não sendo possível iniciar com símbolo nem dígito;
- As *features* filho de cada *feature* têm que estar agrupadas de acordo com algumas regras:
  - `group allOf` - semelhante a uma relação `and`. Podem ser escolhidas zero ou mais *features*;
  - `group oneOf` - equivalente a uma relação `xor`. Apenas uma *feature* pode ser escolhida;
  - `group someOf` - igual a uma relação `or`. Tem que ser escolhida pelo menos uma *feature*.
- Dentro de um `group allOf`, as *features* filho, quando são opcionais necessitam de uma *tag* denominada `opt`, para serem distinguidas das obrigatórias que não necessitam de nenhum tipo de marcação;
- As restrições do modelo têm que estar inseridas dentro de uma *feature*;
- Todas as *features* filho têm que apresentar uma vírgula no final, exceto a última;
- Todas as restrições têm que apresentar um ponto e vírgula no final.

Na Listagem 3.1 é listado o conteúdo de um ficheiro TVL correspondente ao *feature model* mostrado acima.

```
1  root Turtlesim {
2   group allOf{
3     opt Add_turtle,
4     opt Move,
5     opt BackgroundColor,
6     opt Safety_controller
7   }
8   Chase_turtle -> Add_turtle;
9  }
10
11 Move {
12   group someOf {
```

[7] https://www.oracle.com/java/

```
13    Chase_turtle,
14    Teleop,
15    Random_controller
16   }
17  }
18
19  Safety_controller {
20   group allOf {
21    opt Center,
22    opt Pen_color
23   }
24  }
```

Listagem 3.1: Ficheiro TVL correspondente ao Turtlesim.

Para utilizar este ficheiro no *parser* do TVL e obter informações acerca do modelo é necessário instalar o TVL e dentro da diretoria onde se encontra aplicar o seguinte comando no terminal:

```
java -jar TVLParser.jar [opção] ficheiro.tvl
```

Caso a parte `[opção]` não seja utilizada, o TVL faz um teste de validade do modelo, ou seja, obtém-se uma resposta sobre se o modelo é válido ou não. Caso se queira utilizar uma opção, as mais importantes para este caso são:

- `-s`: faz análises sintática do modelo;
- `-c`: imprime o número de *features*;
- `-sat`: verifica se um modelo é satisfazível;
- `-prods`: conta o número de produtos válidos do modelo e imprime-os (pode conter duplicados);
- `-uprods`: conta o número de produtos únicos válidos do modelo e imprime-os (menos eficiente mas não contém duplicados).

No caso do nome do ficheiro TLV tem que se apresentar o caminho para este ficheiro e não apenas o nome. No caso de se querer ver todos os produtos válidos únicos do modelo representado acima, terá que se utilizar o seguinte comando:

```
java -jar TVLParser.jar -uprods path/Turtlesim.tvl
```

O resultado deste comando foi um total de 120 produtos possíveis, tal como se tinha concluído anteriormente, junto com todas as configurações desses produtos. Esse resultado encontra-se na Listagem 3.2. Como se pode ver, são encontrados 120 produtos e todos eles são identificados, sendo que aqui apenas estão alguns desses produtos.

```
- Turtlesim
- Turtlesim, Add_turtle
- Turtlesim, Add_turtle, BackgroundColor
- Turtlesim, Add_turtle, Move, Chase_turtle
- Turtlesim, Add_turtle, Move, Chase_turtle, BackgroundColor
- Turtlesim, Add_turtle, Move, Chase_turtle, Safety_controller
- Turtlesim, Add_turtle, Move, Chase_turtle, Safety_controller, BackgroundColor
- Turtlesim, Add_turtle, Move, Chase_turtle, Safety_controller, Center
- Turtlesim, Add_turtle, Move, Chase_turtle, Safety_controller, Center, BackgroundColor
- Turtlesim, Add_turtle, Move, Chase_turtle, Safety_controller, Center, Pen_color
- Turtlesim, Add_turtle, Move, Chase_turtle, Safety_controller, Center, Pen_color,
    ↪ BackgroundColor
- Turtlesim, Add_turtle, Move, Chase_turtle, Safety_controller, Pen_color
- Turtlesim, Add_turtle, Move, Chase_turtle, Safety_controller, Pen_color, BackgroundColor
- Turtlesim, Add_turtle, Move, Random_controller
- Turtlesim, Add_turtle, Move, Random_controller, BackgroundColor
- Turtlesim, Add_turtle, Move, Random_controller, Chase_turtle
- Turtlesim, Add_turtle, Move, Random_controller, Chase_turtle, BackgroundColor
- Turtlesim, Add_turtle, Move, Random_controller, Chase_turtle, Safety_controller
- Turtlesim, Add_turtle, Move, Random_controller, Chase_turtle, Safety_controller,
    ↪ BackgroundColor
- Turtlesim, Add_turtle, Move, Random_controller, Chase_turtle, Safety_controller, Center
- Turtlesim, Add_turtle, Move, Random_controller, Chase_turtle, Safety_controller, Center,
    ↪ BackgroundColor
[...]
Found 120 unique products.
```

Listagem 3.2: Resultado da aplicação do *parser* do TVL para o Turtlesim.

3.3 Programação com Variabilidade

Como já foi referido anteriormente, é bastante desafiante programar sistemas com variabilidade pois são sistemas complexos que podem depender de vários cenários.

Uma das estratégias mais utilizada para implementar a variabilidade em sistemas de *software* da mesma família é a cópia e adaptação de código para novos requisitos. Esta estratégia é chamada *clone and own*. Apesar de ser um método simples, não resolve todos os problemas, pois quando os sistemas não são parecidos, esta estratégia não funciona, pois não há grande parte *software* que se possa aproveitar. Um problema ainda maior dessa abordagem é a fraca manutenibilidade, pois quando se encontra um *bug* num clone pode ser difícil perceber onde corrigi-lo nos restantes clones.

Para combater estes problemas que a implementação da variabilidade por *clone and own* provoca existem outros mecanismos conhecidos para implementar as SPLs, que se baseiam essencialmente em

dois tipos de abordagens: composicionais e anotativas. Estas duas abordagens vão ser melhor explicadas e comparadas de maneira a adquirir uma melhor compreensão sobre este tema. Também para ajudar na programação de sistemas com variabilidade, é útil o *plugin* do Eclipse já mencionado, featureIDE, pois apresenta suporte para várias linguagens conhecidas, tais como: C, C++, Haskell, Java, e outras, tornando mais acessível o processo de desenvolvimento.

3.3.1 Abordagens Composicionais

Estas abordagens baseiam-se na separação do código em unidades de código em que cada uma representa uma *feature* a ser implementada. A criação das variantes do sistema vai depender das *features* que são incluídas no processo de composição. Primeiro é necessário programar as unidades de código individuais, uma delas referente à base do código, e que será igual para todos os produtos dessa linha de produtos, e várias outras unidades que implementam as diversas *features*. Depois disso é realizada a composição, em que a base é agregada às *features* pretendidas de maneira a gerar o código final desejado. Alguns exemplos de ferramentas para abordagens composicionais são:

- **AHEAD** [7] (*Algebraic Hierarchical Equations for Application Design*): permite que o projeto seja representado de forma composicional a partir de expressões algébricas – suportada pelo featureIDE;
- **FSTComposer** [6]: permite a composição de componentes de *software* representados por FSTs (*Feature Structure Trees*);
- **EPFComposer** [20] (*Eclipse Process Framework Composer*): permite a construção de um processo desde o início, a personalização da estrutura de um processo existente e ainda integrar uma família de processos.

3.3.2 Abordagens Anotativas

As abordagens anotativas implementam as *features* com alguma forma de anotação explícita ou implícita no código fonte. Um exemplo de anotação é o sublinhar da parte do código que pertence a cada uma das *features*. O código acaba por estar na mesma base de código, mas os blocos de código referentes às *features* ficam destacados de alguma forma. Assim, para se criar uma variante, o código anotado de uma *feature* poderá ser retirado antes da compilação. Dentro desta abordagem existem alguns exemplos mais utilizados tais como:

- **Instruções `#ifdef`** [29]: permitem a marcação de blocos de código como opcionais ou condicionais. Esses blocos correspondem ao código referente a cada *feature* de um produto. Esta é uma opção bastante explorada e investigada. A verbosidade das anotações pode tornar o código confuso, o que muitas vezes origina o bem conhecido `#ifdef` *hell* [28];

- **CIDE** [22] (*Colored Integrated Development Environment*): permite a separação virtual do código, sendo o código de cada *feature* anotado com a utilização de cores de fundo – ferramenta construída a partir do featureIDE;

- **GenArch-P** [3]: permite a derivação de um produto *model-based* com suporte para a criação de modelos generativos que representam as variabilidades de cada processo, associando-as aos modelos existentes no processo.

3.3.3 Comparação das Duas Abordagens

Estas duas abordagens são bastante utilizadas e ainda existem dúvidas de qual a melhor abordagem. Aqui vai ser feita uma comparação das duas de acordo com alguns critérios [2, 21]. Esta comparação não significa que uma será melhor do que a outra, apenas que uma das abordagens será mais indicada para um tipo de resultados e outra para outro. Na Tabela 3.2 pode ser observada a comparação entre os dois tipos de abordagens, onde ●, ● e ● correspondem a forte suporte, suporte médio e sem suporte, respetivamente.

Os critérios utilizados foram rastreabilidade (capacidade de seguir uma *feature* desde o modelo até à implementação), modularidade (aproveitamento de blocos de código para vários produtos), granularidade (capacidade de divisão do sistema em partes), correção sintática (garantia de que os produtos estão sintaticamente corretos), independência da linguagem (inexistência de linguagens específicas, ou então, facilidade de extensão para outras), adoção de SPL (facilidade da indústria em adotar a abordagem), deteção de erros (capacidade de verificar a consistência de um SPL e dos seus processos), uniformidade (capacidade de fornecer suporte uniforme para várias linguagens) e gestão da variabilidade (mecanismos de especificação da variabilidade e suporte de derivação automatizada de processos existentes).

	Composicional	Anotativa
Rastreabilidade	●	●
Modularidade	●	●
Granularidade	●	●
Correção sintática	●	●
Independência de linguagem	●	●
Adoção de SPL	●	●
Deteção de erros	●	●
Uniformidade	●	●
Gestão da variabilidade	●	●

Tabela 3.2: Comparação entre as abordagens composicional e anotativa.

3.4 Extração de Modelos de Variabilidade

As SPLs têm a vantagem de criar produtos de alta qualidade, rapidamente prontos para a colocação no mercado e a custos reduzidos, pois são baseados numa reutilização estratégica de recursos, levando também a melhorias ao nível da produtividade.

Normalmente, sistemas mais configuráveis são divididos em espaço do problema e espaço da solução [14], sendo estes definidos pelas *features* e suas dependências e pela realização técnica do sistema e das funcionalidades especificadas pelas *features*, respetivamente. Esta separação de espaços permite a um utilizador, não conhecedor de detalhes de implementação, tomar decisões de configuração.

Existem alguns métodos para a extração de modelos de variabilidade a partir de implementações, que vão ser apresentados de seguida. Estes são procedimentos complexos, baseados em engenharia reversa e que levaram a resultados interessantes.

3.4.1 Engenharia Reversa de *Feature Models*

Muitas são as técnicas utilizadas atualmente para a extração de modelos de variabilidade. A criação destes modelos requer muito tempo e é necessário um esforço muito grande por parte de um modelador. Daí surgiu a ideia de um procedimento para engenharia reversa de *feature models* com base numa heurística para identificar o pai de cada *feature* [27]. Se for possível reduzir a construção de um modelo apenas à seleção do pai de cada *feature* já será possível reduzir o tempo de construção e o número de opções a considerar para a *feature* pai.

Os dados necessários para a execução deste procedimento de seleção de possíveis candidatos a pai são os seguintes:

- Nome das *features*: serve para identificação das mesmas;
- Descrição das *features*: vai servir para ajudar a escolher o melhor candidato a pai;
- Fórmula proposicional que especifique as dependências entre todas as *features*: serve para ajudar na escolha de melhor candidato.

O procedimento pega em cada *feature* e tenta descobrir qual a sua *feature* pai, dentre todas as que estão disponíveis. Para isso são construídas duas listas de *features* candidatas a pai para cada *feature*. Essas duas listas apresentam regras diferentes, o que vai resultar, normalmente, em duas listas diferentes, não excluindo o facto de poderem ser iguais. A semântica dos *feature models* afirma que uma *feature* filho implica sempre uma *feature* pai, por isso, as implicações são utilizadas como principal critério de escolha de candidatos a pai. Os nomes, regras e características de cada lista estão presentes de seguida:

- **RIFs** (*Ranked Implied Features*):
  - Esta lista contém apenas as *features* que a *feature* a analisar implica;

  - É uma lista que está ordenada por similaridade à *feature* analisada;
  - O candidato mais provável de ser pai está no topo da lista;
  - É uma lista "incompleta", mas com maior precisão.

- **RAFs** (*Ranked All-Features*):
  - É uma lista que contém todas as *features* do sistema;
  - Lista ordenada por similaridade à *feature* analisada;
  - O candidato mais provável está no topo;
  - Pode ser uma lista "completa", mas tem uma menor precisão.

Todos os dados de entrada falados anteriormente são utilizados para a construção destas listas. Para que seja possível descobrir as *features* implicadas é necessário recorrer à fórmula proposicional. Mas como uma *feature* pode implicar mais do que uma *feature*, a descrição é utilizada para criar a ordem do candidato mais provável para o menos provável. Se a descrição de uma *feature* candidata for mais semelhante à *feature* que se está a analisar, então fica num lugar superior em relação a outras.

Esta técnica foi testada em dois tipos de sistemas: Linux e eCos, sistemas para os quais existia um *feature model*; *Free Berkeley Software Distribution* (FreeBSD), sistema mais pobre em termos de documentação, não apresentava, por exemplo, *feature model*. Os resultados obtidos destes testes foram bastante animadores pois a lista RIF contém a *feature* pai nas 5 primeiras posições em 76% das *features* no Linux e em 79% das *features* no eCos.

Com isto, conseguiu-se reduzir o número de opções de candidatos a *feature* pai, de milhares para cinco ou menos, o que reduz significativamente o tempo gasto na construção do modelo.

3.4.2 Mineração das Restrições de Configuração

Esta técnica [24] é bastante interessante no ponto de vista desta tese, pois vai ser utilizada uma abordagem deste tipo para estudar a variabilidade nos sistemas ROS. Aqui vai ser apresentado um exemplo, em que o objetivo é avaliar a precisão e escalabilidade do método, perceber quão recuperáveis são as restrições de um modelo de variabilidade e classificar as fontes da variabilidade.

Esta é uma abordagem que utiliza os erros na fase de compilação e uma nova heurística de *feature effect* para extrair automaticamente as restrições de configuração de código C. Para isso foram utilizadas duas especificações: cada configuração válida não deve ter erros na fase de compilação, de modo a poder ser analisada e pré-processada; cada configuração válida do sistema deve produzir um programa lexicalmente diferente.

Na Tabela 3.3 são apresentadas as etapas do processo para chegar aos objetivos pretendidos.

Etapas	Objetivos	Procedimento
1	Avaliar precisão e escalabilidade	- Extrair as restrições do modelo de variabilidade; - Extrair as restrições do espaço da solução; - Verificar a existência das restrições extraídas da solução na fórmula proposicional que representa as restrições do modelo.
2	Avaliar capacidade de recuperação	- Verificar a existência das restrições do modelo de variabilidade nas fórmulas das restrições do espaço da solução que foram extraídas; - Medir automaticamente quantas das restrições no modelo de variabilidade correspondem às dependências estaticamente detetáveis no código.
3	Classificação das restrições	- Análise prévia (aquando da avaliação da recuperação) e automatizada, das restrições que se conseguiu recuperar; - Análise manual das restrições que não se conseguiram recuperar, portanto não puderam ser avaliadas na etapa anterior.

Tabela 3.3: Etapas da mineração das restrições de configuração.

Para extrair as restrições do espaço da solução foram utilizadas estratégias de extração escalonáveis com base no uso estrutural das diretivas `#ifdef`, no analisador e erros de tipo, e nas verificações do *linker*.

Os resultados desta técnica são bastante animadores e interessantes. Como resultado tem-se que a extração de restrições de configuração é viável até um certo ponto. Foi possível recuperar restrições que em 93% dos casos garantem um processo de construção correto. Quanto à recuperação de restrições dos modelos de variabilidade, apenas se conseguiram recuperar 19% das suas restrições. Desta maneira, e com os resultados obtidos por esta técnica, concluiu-se que a engenharia reversa de recursos pode ser suportada.

3.4.3 Identificação de *Features* a Partir de Código-fonte

Este método de extração [31] propõe 3 etapas para a identificação de *features* a partir do código-fonte e posterior criação de modelo de variabilidade, sendo que as duas primeiras etapas são automatizadas. As etapas desta abordagem são:

1. Realização de engenharia reversa em cada modelo a partir do código-fonte de cada produto. É extraído um diagrama de classes de cada produto, por engenharia reversa. Esse diagrama é decomposto num conjunto de peças atómicas, onde cada peça é uma *construction primitive* (CP) do modelo. Assim, cada produto é abstraído como um conjunto de CPs.

2. Identificação dos candidatos a *features* com base num algoritmo automático. Para que o algoritmo funcione é necessário que os candidatos sejam formalmente definidos. A partir da etapa anterior

tem-se as CPs definidas formalmente. Como os produtos são conjuntos de CPs e os candidatos a *feature* são partes de produtos, então os candidatos a *feature* também são conjuntos de CPs. A forma encontrada para que conjuntos de CPs não relacionados não sejam considerados candidatos, foi considerar-se apenas as CPs que estão sempre presentes nos mesmos produtos.

3. Remoção dos candidatos não relevantes e adição de *features* que possam ter sido perdidas, desta maneira é produzido um conjunto final de *features* que servirá como base para construir um *feature model*.

Na prática, o que esta abordagem faz é ir a cada produto e abstraí-lo como um conjunto de CPs. Depois disso, identificam-se as *features*, sendo que essa identificação depende das CPs que estão sempre presentes nos mesmos produtos. No final, a única etapa manual desta abordagem permite remover *features* não relevantes e adicionar outras necessárias que não foram identificadas automaticamente.

Assim, com esta abordagem é possível obter rapidamente um conjunto de candidatos a *feature*, sendo essa a sua principal vantagem. Isto só é possível graças à automatização das duas primeiras etapas e ao alto nível de abstração da primeira etapa. É uma abordagem que tem possibilidade de gerar bons resultados, mas que precisa de ser mais trabalhada.

3.5 Estudos Empíricos

O termo "estudo empírico" pode ser definido como a procura de informações ou resultados a partir da experiência. Existem bastantes estudos empíricos que contribuem para a compreensão dos sistemas ROS, assim como o seu funcionamento. Neste capítulo vão ser expostos dois estudos empíricos, que foram considerados relevantes para o trabalho que se pretende desenvolver. Um deles é referente ao sistema ROS e às suas primitivas e o outro aborda a extração de variabilidade, temas que se revelam importantes para esta tese.

Um dos estudos tem como principal objetivo perceber como são usadas as primitivas dos sistemas ROS na prática e detetar padrões de uso para as funcionalidades desse sistema, tendo sido, para isso, analisados vários pacotes ROS [13]. Este estudo focou-se nas primitivas, *features* e *launch files* do ROS. Para atingir os objetivos pretendidos, tal como em qualquer estudo empírico foram realizadas observações e análises, assim como a extração de atributos e valores interessantes ao estudo. Este estudo é particularmente útil para, por exemplo, o desenvolvimento de novas ferramentas de análise estática para ROS, pois permite compreender como são usadas, na prática as várias primitivas. Em relação a este aspecto, no final concluiu-se que há dois aspetos no uso das primitivas que não podem ser negligenciados: argumentos não literais e ocorrências condicionais.

Quanto ao segundo estudo referido, consiste numa abordagem robusta para a extração da variabilidade de sistemas Linux [16]. Seria espectável que, para a extração da variabilidade fossem analisados todos os *scripts* de construção. Em vez disso, foi explorado o próprio sistema de construção para produzir as informações de variabilidade. Tal como anteriormente, para alcançar os objetivos esperados, foi posta em

prática esta abordagem de maneira a recolher o máximo de informações, ajudando assim em investigações e projetos nesta área. Os resultados deste estudo comprovam que esta abordagem funciona em todas as versões e arquiteturas do sistema Linux.

Capítulo 4

Gestão da Variabilidade em Sistemas ROS

Este capítulo começa por analisar alguns sistemas ROS, de maneira a perceber qual a variabilidade dos mesmos e quais os mecanismos utilizados para gerir essa mesma variabilidade. Depois disso vão ser apresentadas algumas conclusões dessa análise.

4.1 Sistemas Analisados

Existem vários sistemas ROS abertos à comunidade, por isso decidiu-se aproveitar alguns deles para identificar os padrões mais típicos de gestão de variabilidade. Aqui vão ser analisados os sistemas de maneira a perceber melhor o que cada um deles faz e o que permite que haja tanta variabilidade. Os sistemas que vão ser analisados são: Kobuki, TurtleBot, TurtleBot3, Husky e Lizi.

Esta análise passa principalmente pela compreensão do sistema que se está a analisar, para isso é realizada uma pesquisa de todas as informações disponíveis, análise do código pertencente ao projeto e ainda a procura de toda a informação relativa ao sistema, seja ela em documentação no seu repositório como no *site* oficial do sistema. Com isto pretende-se que seja possível uma análise rigorosa e o mais correta possível do sistema. Todas as etapas descritas vão ser demonstradas de seguida para cada sistema.

4.1.1 Kobuki

O Kobuki[1] é uma base móvel projetada para educação e pesquisa em robótica. Esta é uma base que para ser funcional precisa que se construa algo em cima, pois sozinha não consegue ser útil. Para que isso seja possível, normalmente apenas é necessária a adição de sensores extra que ajudem no desempenho do robô e um pequeno computador portátil que funciona como núcleo computacional do sistema. Esta base,

[1] http://kobuki.yujinrobot.com/

para além de permitir a adição destes componentes, é também compatível com a utilização conjunta do TurtleBot, outro sistema que também vai ser analisado neste capítulo. Na Figura 4.1 pode-se observar o aspeto deste sistema, como se pode verificar, este é um sistema de dimensões relativamente pequenas e que permitirá a adição de novos elementos ao seu *hardware* para melhorar o seu funcionamento.

Figura 4.1: Kobuki.

No repositório relativo a este sistema[2] podem ser encontradas várias diretorias, sendo que algumas contêm uma em específico (`launch`) onde se podem encontrar os *launch files*. Posto isto, identificou-se cada diretoria como uma funcionalidade, ou conjunto de funcionalidades do sistema, recolhendo-se assim a seguinte informação acerca deste robô:

- `auto_docking` - busca automática pelo carregador. No interior desta diretoria encontra-se uma diretoria nomeada `launch` que contém cinco *launch files*:
  - `activate` - ficheiro que apenas ativa o serviço de *auto docking*, com este ficheiro apenas se envia uma mensagem para que o serviço seja realizado;
  - `auto_dock_with_safe_keyop` - configuração específica de um robô com *auto docking*, controlador de segurança e movimentação através do teclado, no entanto este ficheiro pressupõe o lançamento prévio do `kobuki_node/minimal.launch` que vai ser descrito mais à frente;
  - `compact` - configuração completa de um robô que utiliza uma base móvel e *auto docking*;
  - `minimal` - ficheiro que representa a funcionalidade de *auto docking*. Para que possa ser utilizado pressupõe-se o lançamento prévio do ficheiro `kobuki_node/minimal.launch`;
  - `standalone` - lança a funcionalidade de *auto docking* sem que seja necessário lançar outro *launch file* como é o caso de alguns ficheiros nesta diretoria.
- `bumper2pc` - conversão de mensagens e eventos. Referente a esta funcionalidade existe apenas um *launch file*:
  - `standalone` - neste ficheiro é ativado um nó para a conversão.
- `capabilities` - capacidades do Kobuki, onde se pode encontrar um *launch file*:

[2] https://github.com/yujinrobot/kobuki

  - `app_manager_with_capabilities` - ficheiro com a função de gestão de aplicações com as capacidades do robô.

- `controller_tutorial` - controlador *bump blink* para o robô. Aqui podem ser encontrados dois *launch files*:

  - `bump_blink_app` - lança um controlador *bump blink*, este ficheiro pressupõe o lançamento do `kobuki_node/minimal.launch` para o seu funcionamento;
  - `standalone` - ficheiro igual ao anterior, com a única diferença que este ativa um `nodelet manager`, o que permite o lançamento isolado deste ficheiro, sem necessidade de outros.

- `description` - descrição do modelo URDF (*Unified Robot Description Format*) e Gazebo. Diretoria com um ficheiro de lançamento:

  - `view_model` - ficheiro utilizado para visualização do modelo.

- `keyop` - movimentação do robô através do teclado. A diretoria `launch` pertencente a esta funcionalidade apresenta dois *launch files* importantes:

  - `keyop` - funcionalidade de movimentação do robô, ficheiro a ser utilizado com as restantes funcionalidades de lançamento padrão;
  - `safe_keyop` - mesma funcionalidade que o ficheiro anterior mas com a adição de um controlador que permite a segurança dos eventos.

- `node` - base de funcionamento do sistema. Podem ser encontrados quatro *launch files* que ajudam a activar esta funcionalidade:

  - `minimal` - ficheiro que lança o sistema padrão na sua forma mais simples;
  - `robot_with_tf` - o sistema é lançado com um método de coordenadas;
  - `test_get_odom` - ficheiro que ativa um nó que dá informação sobre a odometria do robô calculada com base em sensores giroscópicos;
  - `test_get_yaw` - este ficheiro ativa um nó que dá informações acerca da direção e velocidade angular.

- `random_walker` - movimento do robô de forma aleatória. Aqui existem três ficheiros de lançamento:

  - `random_walker_app` - lança um controlador de movimentação aleatória, mas pressupõe o lançamento do `kobuki_node/minimal.launch` para que funcione;
  - `safe_random_walker_app` - configuração específica de um robô com um controlador de movimentação aleatória e um controlador que assegura a segurança dos movimentos, no entanto este ficheiro necessita do lançamento prévio do *launch file* `kobuki_node/minimal.launch`;

  - `standalone` - ficheiro que lança a funcionalidade do controlador de movimentação aleatória, sem a necessidade de lançamento de outros *launch files*, pois ativa um `nodelet manager` que evita esse trabalho.

- `safety_controller` - controlador que assegura a segurança das operações. Esta funcionalidade apresenta um *launch file* `standalone` que funciona isoladamente e permite ativar um controlador de segurança.

- `testsuite` - conjunto de ferramentas para teste do sistema.

Para além das diretorias existentes no repositório, foram também analisados todos os *launch files* de modo a perceber melhor o sistema. A estrutura de um ficheiro deste tipo é bastante compreensível e clara. Na Listagem 4.1 está um exemplo de um *launch file* do sistema Kobuki[3] onde se podem localizar algumas características destes ficheiros. No início de cada ficheiro é sempre aberta uma *tag* chamada *launch*, é desta forma que o ficheiro é identificado como *launch file*, sendo fechada no final do ficheiro. Entre estas duas *tags* existem várias opções para completar o ficheiro, sendo que neste exemplo apenas aparecem duas dessas opções:

- `arg` - especifica um argumento necessário ao lançamento do ficheiro;

- `node` - especifica um nó que é ativado assim que o ficheiro é lançado, para isso apenas é necessário o nome do nó, o pacote a que pertence e o seu tipo.

```
1  <launch>
2    <arg name="kobuki_publish_tf" default="true"/>
3
4    <node pkg="nodelet" type="nodelet" name="mobile_base_nodelet_manager" args="manager"/>
5    <node pkg="nodelet" type="nodelet" name="mobile_base" args="load kobuki_node/KobukiNodelet
         ↪ mobile_base_nodelet_manager">
6      <rosparam file="$(find kobuki_node)/param/base.yaml" command="load"/>
7      <param name="publish_tf" value="$(arg kobuki_publish_tf)"/>
8      <remap from="mobile_base/odom" to="odom"/>
9      <remap from="mobile_base/joint_states" to="joint_states"/>
10   </node>
11
12   <node pkg="diagnostic_aggregator" type="aggregator_node" name="diagnostic_aggregator" >
13     <rosparam command="load" file="$(find kobuki_node)/param/diagnostics.yaml" />
14   </node>
15 </launch>
```

Listagem 4.1: Exemplar de um *launch file* do sistema Kobuki.

[3] `kobuki_node/launch/minimal.launch`

Durante a análise deste sistema chegou-se a algumas conclusões, principalmente acerca das incompatibilidades de *launch files*, informação relevante para o estudo a desenvolver nesta tese.

Neste exemplo são ativados dois nós, como se pode observar apresentam nomes diferentes, pois quando dois nós com o mesmo nome são lançados em simultâneo, o primeiro ao ser ativado morre e apenas o segundo fica ativado. Assim, conclui-se que dois *launch files* que ativem nós com o mesmo nome não deverão ser lançados na mesma configuração e serão considerados incompatíveis.

Analisando ao pormenor todos os ficheiros percebeu-se também que os ficheiros `standalone` são incompatíveis com todos os *launch files* que partilham a mesma diretoria. Assim, conclui-se que servem para ser lançados sozinhos em relação aos ficheiros referentes à mesma funcionalidade. Na Tabela 4.1 estão identificados todos os ficheiros incompatíveis neste sistema, exceto as incompatibilidades com os ficheiros `standalone`, pois aumentaria significativamente o tamanho da tabela. Como alguns ficheiros apresentam o mesmo nome, antes do nome do ficheiro foi acrescentado o nome da diretoria onde este se encontra.

Ficheiros incompatíveis	
robot_with_tf	kobuki_node_minimal
robot_with_tf	compact
robot_with_tf	view_model
kobuki_node_minimal	compact
compact	kobuki_auto_docking_minimal
compact	auto_dock_with_safe_keyop
keyop	safe_keyop
keyop	auto_dock_with_safe_keyop
safe_keyop	auto_dock_with_safe_keyop
safe_keyop	safe_random_walker_app
safe_random_walker_app	auto_dock_with_safe_keyop
random_walker_app	safe_random_walker_app
kobuki_auto_docking_minimal	auto_dock_with_safe_keyop

Tabela 4.1: Ficheiros incompatíveis no Kobuki.

Outra das conclusões importantes a que se chegou sobre este sistema é que a abordagem utilizada para a programação da variabilidade neste robô é composicional. Esta conclusão foi retirada a partir dos *launch files* que se analisou, pois existem vários para as diferentes funcionalidades que devem ser activados em conjunto para criar o robô que se pretende. Isto, tal como visto na Secção 3.3, é uma característica dos sistemas que utilizam uma abordagem composicional.

4.1.2 TurtleBot

O TurtleBot[4] é um robô pessoal que, tal como o Kobuki, apresenta *software* de código aberto[5]. Neste momento existem 3 versões, a primeira já se encontra descontinuada, a segunda vai ser agora analisada e mais à frente será analisada a terceira versão. O TurtleBot é um robô modular, podendo ter como base o Kobuki já referido anteriormente. Na Figura 4.2 mostra-se em que consiste o TurtleBot na sua segunda versão.

Figura 4.2: TurtleBot.

Neste sistema, tal como no Kobuki conseguiram-se identificar algumas características do robô a partir da análise do repositório e do código disponível. Os *launch files* foram encontrados também em diretorias denominadas `launch` que se encontravam dentro de diretorias que implementam as várias características possíveis de activar neste robô. Assim, tal como foi feito no sistema anterior vão-se descrever as diferentes diretorias, bem como os *launch files* existentes em cada uma delas. Deste modo, deverá ser possível perceber o sistema e encontrar as suas características que, mais tarde, vão colaborar na caracterização do sistema e numa automatização dessa caracterização. Na lista seguinte são enumeradas as diretorias principais e para cada delas os *launch files* que se encontram na respectiva diretoria `launch`.

- `turtlebot_bringup` - os ficheiros aqui presentes são os necessários para iniciar as funcionalidades básicas do TurtleBot. Para essa iniciação existem quatro *launch files*:
  - `minimal` - ficheiro que lança o TurtleBot na sua forma mais básica, faz a inclusão de outros ficheiros para que o sistema fique completo, apesar de básico;
  - `concert_minimal` - ficheiro que inclui o `minimal` e, para além disso apresenta mais argumentos e a inclusão de outro ficheiro;
  - `concert_client` - ficheiro que, tal como o `minimal` lança o TurtleBot na sua forma mais simples mas com um gestor de aplicações;
  - `3dsensor` - ficheiro que existe principalmente para lançar um sensor 3D.

[4] https://www.turtlebot.com/
[5] https://github.com/turtlebot/turtlebot

- `turtlebot_teleop` - permite tele-operações com diferentes tipos de dispositivos. Os *launch files* que se encontram nesta diretoria representam cada opção de dispositivo para a tele-operação, que neste caso são quatro:
  - `keyboard_teleop` - permite tele-operações a partir do teclado;
  - `logitech` - permite tele-operações a partir de um comando Logitech;
  - `ps3_teleop` - permite tele-operações a partir de um comando Ps3;
  - `xbox360_teleop` - permite tele-operações a partir de um comando Xbox 360.

Neste sistema, para além dos *launch files* observou-se também a existência de argumentos que o caracterizam. Nos ficheiros da diretoria `bringup`, com exceção do `3dsensor.launch` existem argumentos que se considerou importantes e que se apresentam de seguida, bem como as opções que apresenta cada argumento:

- `base` - tipo de base móvel para o robô:
  - `create`
  - `roomba`
  - `kobuki`
- `battery` - informação da bateria;
- `3dsensor` - tipo de sensor 3D para o robô:
  - `kinect`
  - `asus_xtion_pro`
  - `asus_xtion_pro_offset`
  - `astra`
  - `r200`
- `stacks` - forma das prateleiras que constituem o robô:
  - `circles`
  - `hexagons`

Durante a análise do repositório, tal como no sistema anterior, foram encontrados alguns elementos nos *launch files* que se considerou importantes. Esses elementos fazem parte da estrutura de um ficheiro deste tipo, por isso, na lista a seguir podem-se identificar mais *tags* do que no sistema anterior. Essas opções, que mais uma vez, devem ser usadas entre as duas *tags launch*, são as seguintes:

- `param` - especifica um parâmetro que é ativado; normalmente é necessário o nome do parâmetro e o seu valor;
- `include` - faz a inclusão de um ficheiro; para essa inclusão é necessário fornecer o caminho para o ficheiro que se pretende incluir. Quando esse ficheiro apresenta argumentos, estes também têm que ser especificados aquando da inclusão. Esta é uma *tag* bastante útil na estruturação dos *launch files*;
- `group` - cria um grupo de operações condicionais:
  - Fazer a ativação se algo for verdade: `group if`;
  - Fazer a ativação a não ser que algo seja verdade: `group unless`.

As *tags* identificadas no Kobuki foram também encontradas neste sistema, no entanto, como já foram referidas anteriormente, apenas se mencionaram as que são diferentes.

Na Listagem 4.2 apresenta-se parte de um *launch file* deste sistema[6] onde se podem observar as *tags* referidas acima.

```
1  <launch>
2   [...]
3   <arg name="scan_processing" default="true"/>
4   [...]
5   <include file="$(find turtlebot_bringup)/launch/includes/3dsensor/$(arg 3d_sensor).launch.
        ↪ xml">
6    <arg name="camera" value="$(arg camera)"/>
7    <arg name="publish_tf" value="$(arg publish_tf)"/>
8    <arg name="depth_registration" value="$(arg depth_registration)"/>
9    <arg name="num_worker_threads" value="$(arg num_worker_threads)" />
10   [...]
11  </include>
12
13  <group if="$(arg scan_processing)">
14   <node pkg="nodelet" type="nodelet" name="depthimage_to_laserscan" args="load
        ↪ depthimage_to_laserscan/DepthImageToLaserScanNodelet $(arg camera)/$(arg camera)
        ↪ _nodelet_manager">
15
16    <param name="scan_height" value="10"/>
17    <param name="output_frame_id" value="$(arg camera)_depth_frame"/>
18    <param name="range_min" value="0.45"/>
19    <remap from="image" to="$(arg camera)/$(arg depth)/image_raw"/>
20    <remap from="scan" to="$(arg scan_topic)"/>
21
22    <remap from="$(arg camera)/image" to="$(arg camera)/$(arg depth)/image_raw"/>
```

[6] `turtlebot/turtlebot_bringup/launch/3dsensor.launch`

```
23      <remap from="$(arg camera)/scan" to="$(arg scan_topic)"/>
24    </node>
25  </group>
26 </launch>
```

Listagem 4.2: Parte de um exemplar de um *launch file* do sistema TurtleBot.

Aqui pode-se perceber como funcionam as *tags* identificadas. Na *tag* `include`, por exemplo pode-se perceber que é incluído um ficheiro: foi fornecido o caminho para o ficheiro e abaixo estão todos os argumentos que esse ficheiro necessita. A *tag* `group if` neste caso depende do valor do argumento `scan_processing`: se esse argumento for `true`, o que está dentro do grupo é ativado, se for `false` então passa-se à frente para a próxima operação. No interior do grupo podem-se observar diferentes parâmetros que estão a ser definidos, onde são passados o nome e o valor dos mesmos.

Depois desta análise, tal como no Kobuki, retiraram-se algumas conclusões também acerca dos ficheiros que seriam incompatíveis pela ativação de nós com o mesmo nome. Assim, na Tabela 4.2 encontram-se todos os *launch files* que se consideraram incompatíveis. Neste caso, apenas foram considerados incompatíveis os ficheiros que ativam dispositivo incompatíveis para realizar tele-operações.

Ficheiros incompatíveis	
`ps3_teleop`	`logitech`
`ps3_teleop`	`xbox360_teleop`
`logitech`	`xbox360_teleop`

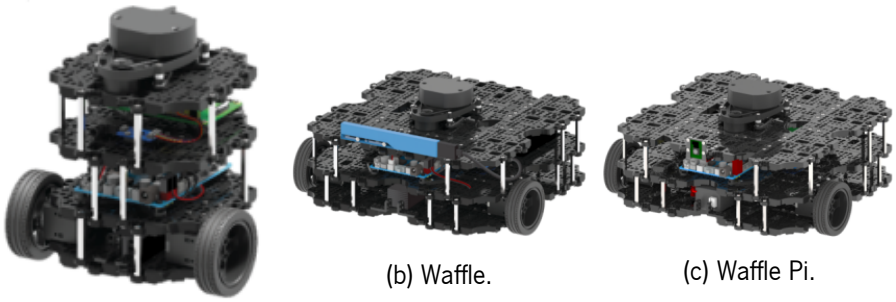
Tabela 4.2: Ficheiros incompatíveis no TurtleBot.

Neste sistema, tal como no anterior pode-se concluir que é utilizada uma abordagem composicional para programar o sistema. Como se viu são criados vários *launch files* independentes, que juntos formam o robô completo. Apesar do código apresentar argumentos em alguns ficheiros que influenciam quais os componentes que são ativados (algo típico numa abordagem anotativa), como é o caso do exemplo dado acima, este é um sistema onde a variabilidade é programada maioritariamente de forma composicional.

4.1.3 TurtleBot3

O sistema TurtleBot3[7], tal como os anteriores, apresenta um sistema de código aberto, o que facilita a sua análise. Tal como o TurtleBot, baseia-se num robô pessoal, sendo esta a sua terceira versão. Este é um pouco mais complexo e apresenta uma estrutura diferente. Foi realizada uma análise na sua documentação e *site* oficial e percebeu-se que existem três modelos para o robô - *Burger* (Figura 4.3a), *Waffle* (Figura 4.3b) e *Waffle Pi* (Figura 4.3c) - sendo que a escolha do modelo poderá alterar as funcionalidades do robô.

[7] https://github.com/ROBOTIS-GIT/turtlebot3



(a) Burger. (b) Waffle. (c) Waffle Pi.

Figura 4.3: TurtleBot3.

Para além da documentação do sistema, também foi analisado o repositório do código, o que facilitou a compreensão do sistema. Na lista que se segue, tal como foi feito nos sistemas anteriores vão ser descritas as diretorias que contêm *launch files* e os nomes dos próprios *launch files* com a respetiva função. Em todas as diretorias que vão ser aqui descritas existe uma outra diretoria chamada `launch` que contém todos os *launch files*. O mesmo acontece com todos os outros sistemas que já foram vistos e com os que ainda vão ser analisados neste capítulo.

- `turtlebot3_bringup` - contém os ficheiros necessários para iniciar o TurtleBot3. Nesta diretoria existem sete *launch files*:
  - `turtlebot3_core` - ativa um nó que lança o núcleo do sistema, ou seja, trata da configuração, publicações e subscrições;
  - `turtlebot3_lidar` - ativa um nó que lança um sensor que envia dados para efeitos de localização;
  - `turtlebot3_model` - inclui um outro *launch file* (`turtlebot3_remote`) e ativa nós que publicam mensagens acerca do estado das articulações do robô[8];
  - `turtlebot3_realsense` - este ficheiro apenas inclui outro que ativa um nó que lança uma câmara RealSense[9];
  - `turtlebot3_remote` - inclui um ficheiro que especifica o modelo do robô com URDF e ativa um nó que publica as posições 3D das articulações do robô;
  - `turtlebot3_robot` - a partir deste ficheiro são lançados os *launch files* `turtlebot3_core` e `turtlebot3_lidar`, o que resulta no lançamento do núcleo do robô com um sensor;
  - `turtlebot3_rpicamera` - ativa um nó que lança uma câmara Raspberry Pi[10].

[8] As articulações de um robô são as componentes móveis dos braços robóticos.
[9] https://www.intelrealsense.com
[10] https://www.raspberrypi.org/products

- `turtlebot3_example` - fornece exemplos básicos para o TurtleBot3. Esta diretoria apresenta bastante exemplos, contendo um total de sete *launch files* que correspondem a nós que vão ser ativados:
  - `interactive_markers` - permite a movimentação do robô com marcadores interativos, o que possibilita a mudança de posição ou rotação;
  - `turtlebot3_bumper` - ativa um sensor específico adicional (Touch_sensor TS-10[11]) que pode ser colocado no robô;
  - `turtlebot3_client` - permite que o TurtleBot3 se mova de forma personalizada, sendo possível movimento em forma de quadrado, triângulo e círculo;
  - `turtlebot3_cliff` - ativa um sensor específico extra (IR_sensor IRSS-10[12]) que pode ser colocado no robô;
  - `turtlebot3_illumination` - faz a ativação de um sensor adicional (LDR sensor Flying-Fish MH-sensor) quando lançado;
  - `turtlebot3_obstacle` - permite que o robô pare quando encontra um obstáculo;
  - `turtlebot3_pointop_key` - permite a inserção da posição do objetivo onde se pretende que o robô chegue;
  - `turtlebot3_sonar` - ativa um sensor específico adicional (Ultrasonic sensor HC-SR04) que pode ser colocado no robô.
- `turtlebot3_navigation` - esta diretoria contém os ficheiros necessários para iniciar a navegação do robô. É composta por três *launch files*:
  - `amcl` - ativa um nó que permite a localização do robô em relação a um mapa conhecido;
  - `move_base` - ficheiro que ativa um nó que, dada uma posição como objetivo, move a base móvel até essa posição;
  - `turtlebot3_navigation` - inclui os ficheiros `turtlebot3_remote`, `amcl` e `move_base`, desta forma faz a ativação completa da navegação do robô.
- `turtlebot3_slam` - contém os ficheiros necessários para ativar o SLAM *(Simultaneous Localization And Mapping)* que permite um mapeamento do espaço e a localização do robô nesse mesmo espaço. Para isso existem sete *launch files* nesta diretoria:
  - `turtlebot3_cartographer` - método de SLAM em 2D e 3D em várias plataformas e configurações de robôs. Este ficheiro inclui um ficheiro já aqui referido: `turtlebot3_navigation/move_base.launch`;

[11] https://emanual.robotis.com/docs/en/parts/sensor/ts-10
[12] https://emanual.robotis.com/docs/en/parts/sensor/irss-10

  - `turtlebot3_frontier_exploration` - método de SLAM que se baseia em receber os locais onde se pretende explorar, para movimentar a base. Neste ficheiro é incluído o ficheiro `gmapping.launch` que vai ser descrito a seguir e o `turtlebot3_navigation/amcl.launch`;
  - `turtlebot3_gmapping` - método de SLAM que recolhe as informações de mapeamento e localização a partir de um *laser* e da posição da base móvel. Aqui é ativado um nó de `gmapping` que ativa a funcionalidade;
  - `turtlebot3_hector` - método de SLAM baseado num sensor do tipo LiDAR (*Light Detection And Ranging*) sem necessidade de medição de distância percorrida e com poucos recursos computacionais. Este ficheiro ativa um nó que, consequentemente ativa esta funcionalidade de SLAM;
  - `turtlebot3_karto` - método de SLAM que recebe dados de distâncias de um sensor e constrói um mapa com a transformação das distâncias recebidas em coordenadas. Com o lançamento deste ficheiro é ativado um nó que permite esta funcionalidade;
  - `turtlebot3_manipulation_slam` - ficheiro que ativa alguns nós e inclui um ficheiro dependendo do método de SLAM que for escolhido;
  - `turtlebot3_slam` - com este ficheiro é possível ativar o método SLAM que se pretende, dependendo do método que o utilizador quer é incluído um dos ficheiros anteriores referentes a métodos SLAM. Para além disso é incluído o ficheiro `turtlebot3_remote`.

- `turtlebot3_teleop` - permite tele-operações a partir do teclado. Aqui existe apenas um *launch file*:
  - `turtlebot3_teleop_key` - neste ficheiro é ativado um nó que permite as tele-operações.

Neste sistema também foram encontrados alguns elementos que fazem parte dos *launch files*, tal como se pode observar pela Listagem 4.3 que pertence a um *launch file* deste sistema[13]. Os elementos identificados são os mesmos que nos sistemas anteriores daí não ser mencionado nenhum em particular como aconteceu anteriormente.

```
1 <launch>
2   <arg name="multi_robot_name" default=""/>
3   <arg name="set_lidar_frame_id" default="base_scan"/>
4   <include file="$(find turtlebot3_bringup)/launch/turtlebot3_core.launch">
5     <arg name="multi_robot_name" value="$(arg multi_robot_name)"/>
6   </include>
7   <include file="$(find turtlebot3_bringup)/launch/turtlebot3_lidar.launch">
8     <arg name="set_frame_id" value="$(arg set_lidar_frame_id)"/>
```

[13] `turtlebot3/turtlebot3_bringup/launch/turtlebot3_robot.launch`

```
9   </include>
10  <node pkg="turtlebot3_bringup" type="turtlebot3_diagnostics" name="turtlebot3_diagnostics"
        ↪ output="screen"/>
11 </launch>
```

Listagem 4.3: Exemplar de um *launch file* do sistema TurtleBot3.

Para a recolha de toda esta informação foi realizada uma análise exaustiva do repositório do sistema, do *site* oficial do mesmo e ainda foi consultado o *site* oficial do ROS, que apresenta também bastante informação acerca de nós e pacotes que por vezes são mencionados no código.

As conclusões relativas a ficheiros incompatíveis, que têm vindo a ser comuns no final de cada análise de sistema, ajudam também a perceber o sistema. Assim, na Tabela 4.3 estão todos os ficheiros incompatíveis que foi possível identificar. Todos estes pares de ficheiros apresentam pelo menos um nó com o mesmo nome, daí serem considerados incompatíveis.

Ficheiros incompatíveis	
turtlebot3_model	turtlebot3_navigation
turtlebot3_model	turtlebot3_manipulation_slam
turtlebot3_model	turtlebot3_slam
turtlebot3_navigation	turtlebot3_manipulation_slam
turtlebot3_navigation	turtlebot3_slam
turtlebot3_manipulation_slam	turtlebot3_slam
turtlebot3_remote	turtlebot3_manipulation_slam
move_base	turtlebot3_frontier_exploration

Tabela 4.3: Ficheiros incompatíveis no TurtleBot3.

Este é um sistema que também apresenta uma abordagem composicional, pela existência de *launch files* independentes em que cada um poderá representar uma característica do sistema. Para a criação do robô na sua forma completa terão que ser lançados vários desses ficheiros em conjunto.

4.1.4 Husky

O Husky[14] é uma base robótica que, tal como os sistemas anteriores, é suportado pelo ROS e tem o seu código disponível[15]. As principais características deste robô são: facilidade de utilização, robô todo o terreno, controlo de precisão e ainda a possibilidade de personalização. Essa personalização irá depender das tarefas que se pretende que o robô execute, podendo ser adicionados sensores, câmaras e manipuladores. O robô na sua forma mais simples encontra-se retratado na Figura 4.4.

[14] https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/
[15] https://github.com/husky/husky

Figura 4.4: Husky.

Muita da informação para a apresentação deste robô foi retirada da página oficial deste sistema que está referida acima. No entanto, para obter mais dados acerca das características deste robô explorou-se o seu código onde foram identificados os seguintes *launch files* e respectivas diretorias:

- `husky_base` - aqui é inicializado o *driver* do robô. Esta diretoria contém apenas um *launch file*:
  - `base` - este ficheiro ativa um nó que corresponde ao *hardware* básico do robô, para além disso inclui dois ficheiros importantes no funcionamento do sistema. Esses dois ficheiros vão ser explicados mais à frente (`control` e `teleop`).
- `husky_bringup` - nesta diretoria estão localizados os ficheiros correspondentes aos sensores possíveis do Husky. Existem sete *launch files* nesta diretoria:
  - `lms1xx` - ativa um nó correspondente a um *driver* de um sensor do tipo LiDAR;
  - `microstrain_gx3` - este ficheiro ativa um nó correspondente a um IMU (*Inertial Measurement Unit*) compatível com os sensores 3DM-GX2[16] e 3DM-GX3[17];
  - `microstrain_gx5` - este ficheiro apresenta vários argumentos, assim como duas *tags* `group` e um nó que corresponde também a um IMU compatível com sensores da linha 3DM-GX5[18];
  - `navsat` - ficheiro que ativa vários nós que dão origem à ativação de um GPS (*Global Positioning System*);
  - `realsense` - inclui um ficheiro relativo a uma câmara RealSense e ativa um nó de *scan* com *laser*;
  - `um6` - ficheiro que, quando lançado, ativa um sensor do tipo CHR-UM6 da linha IMU;
  - `um7` - ficheiro que ativa um sensor do tipo UM7 da linha IMU.
- `husky_control` - configurações de controladores. Aqui existem dois *launch files*:
  - `control` - ficheiro que faz a inclusão de alguns ficheiros, apresenta bastantes argumentos e ativa alguns nós, e tudo isto em conjunto ativa um controlador de parâmetros e de localização básica;

[16] https://www.microstrain.com/inertial/3dm-gx2
[17] https://www.microstrain.com/inertial/3dm-gx3-25
[18] https://www.microstrain.com/inertial-sensors/3dm-gx5-10

  - `teleop` - ficheiro que permite que haja tele-operações com o robô a partir de comandos Ps4 ou Logitech.

- `husky_description` - diretoria que apresenta apenas um *launch file* (`description`) que contém a descrição URDF do robô;

- `husky_gazebo` - contém os ficheiros necessários para lançar o robô em modo de simulação. Aqui existem seis ficheiros de lançamento:

  - `husky_empty_world` - ambiente de simulação onde o mundo é vazio e podem ser adicionados novos objetos, apenas existe um Husky e um plano infinito;
  - `husky_playpen` - este ficheiro apresenta uma outra forma de criar um ambiente com apenas um Husky e um plano infinito, tal como o anterior;
  - `multi_husky_playpen` - ambiente de simulação que apresenta três robôs Husky com um plano infinito;
  - `playpen` - ambiente de simulação vazio. Neste mundo apenas existe um plano infinito, não havendo um robô Husky;
  - `realsense` - ativa um nó de *scan* com *laser* a ser utilizado nos ambientes de simulação;
  - `spawn_husky` - este é um ficheiro que serve para ser incluído noutros que representem um mundo de simulação. O principal objetivo deste ficheiro é adicionar um Husky ao ambiente de simulação onde o ficheiro é incluído.

- `husky_navigation` - nesta diretoria estão ficheiros para o mapeamento autónomo e navegação do Husky. Aqui estão presentes oito *launch files*:

  - `amcl` - ativa um nó que ativa um sistema de localização probabilística para um robô num mapa conhecido;
  - `amcl_demo` - ficheiro que contém a inclusão de dois ficheiros (`amcl` e `move_base`) para criar uma demonstração da utilização dos mesmos;
  - `exploration` - ficheiro cujo conteúdo está comentado, porém esperava-se que ativasse um nó que implementa a exploração das fronteiras do mapa;
  - `exploration_demo` - esperava-se que, tal como o outro ficheiro `demo`, este incluísse o `exploration`, mas não é esse o caso. Este ficheiro utiliza o `gmapping` que vai ser explicado a seguir e `move_base`;
  - `gmapping` - este ficheiro ativa um nó que oferece SLAM (mapeamento e localização simultâneos) baseado em *laser*;
  - `gmapping_demo` - ficheiro que apenas inclui outros dois *launch files* (`gmapping` e `move_base`) para demonstrar o uso destes;

  - `move_base` - quando lançado, este ficheiro ativa um nó que, dado um objetivo, tenta alcançar esse ponto no mapa, ou seja, movimenta o robô até certo ponto;
  - `move_base_mapless_demo` - faz uma demonstração da utilização do ficheiro `move_base` através da sua inclusão.

- `husky_viz` - aqui encontram-se as configurações de visualização do Husky. Podem-se encontrar dois *launch files*:
  - `view_model` - visualizador do modelo do robô;
  - `view_robot` - visualizador do próprio robô.

Este é um sistema com bastante *launch files*, o que torna esta análise bastante longa, mais do que qualquer um dos sistemas anteriores. Pode-se perceber que é um sistema com bastantes características e, para além do trabalho que o robô pode fazer realmente, o Husky ainda permite uma simulação em ambientes próprios.

Na Listagem 4.4 pode-se encontrar um exemplo de um *launch file* deste sistema[19].

```
<launch>

 <arg name="joy_dev" default="$(optenv HUSKY_JOY_DEVICE /dev/input/js0)" />
 <arg name="joystick" default="true" />

 <group ns="joy_teleop">

   <group unless="$(optenv HUSKY_LOGITECH 0)" >
     <rosparam command="load" file="$(find husky_control)/config/teleop_ps4.yaml" />
   </group>

   <group if="$(optenv HUSKY_LOGITECH 0)" >
     <rosparam command="load" file="$(find husky_control)/config/teleop_logitech.yaml" />
     <param name="joy_node/dev" value="$(arg joy_dev)" />
   </group>

   <node pkg="joy" type="joy_node" name="joy_node" />

   <node pkg="teleop_twist_joy" type="teleop_node" name="teleop_twist_joy"/>
 </group>

</launch>
```

Listagem 4.4: Exemplar de um *launch file* do sistema Husky.

[19] `husky/husky_control/launch/teleop.launch`

Este é um ficheiro que permite ativar tele-operações a partir de um comando Ps4 ou Logitech. Estas duas opções podem ser determinadas a partir dos blocos condicionais nas linhas 9 e 13, respetivamente.

Pode-se ainda retirar algumas conclusões acerca dos ficheiros que são considerados incompatíveis pela ativação de nós com o mesmo nome. Na Tabela 4.4 encontram-se todos esses ficheiros. Como existem ficheiros com o mesmo nome mas em diretorias diferentes, para a sua distinção foi colocado o nome da diretoria a que pertencem antes do nome do ficheiro.

Ficheiros incompatíveis	
`husky_gazebo_realsense`	`husky_bringup_realsense`
`control`	`view_model`
`view_model`	`view_robot`
`um6`	`um7`

Tabela 4.4: Ficheiros incompatíveis no Husky.

Este sistema, tal como os anteriores apresenta uma abordagem composicional na programação da sua variabilidade, pois, tal como os outros sistemas faz a criação de *launch files* individuais para que, juntos possam criar o sistema completo. Isto faz com que seja o utilizador a lançar os ficheiros que pretende para obter o resultado que melhor o satisfaz. Durante a análise dos *launch files* encontrou-se, por vezes, algum comportamento condicional através de parâmetros, porém este é, tal como os anteriores, um sistema maioritariamente composicional.

4.1.5 Lizi

O sistema Lizi[20] é uma base robótica, tal como as anteriores. O principal objetivo deste sistema é o mapeamento, navegação e pesquisa, sendo apropriado para agir no exterior ou num espaço fechado. Este robô é, tal como os anteriores, compatível com o ROS e será o último sistema a ser analisado. A Figura 4.5 apresenta a aspeto deste robô.

Figura 4.5: Lizi.

Mais uma vez, para a análise do sistema foi visitado o respetivo repositório[21] com o código correspondente a este sistema. Numa primeira análise rápida, quando se vê apenas as diretorias e ficheiros sem entrar em cada um deles e analisá-los, este sistema parece ter o mesmo processo de criação que

[20] https://robotican.net/lizi/
[21] https://github.com/robotican/lizi

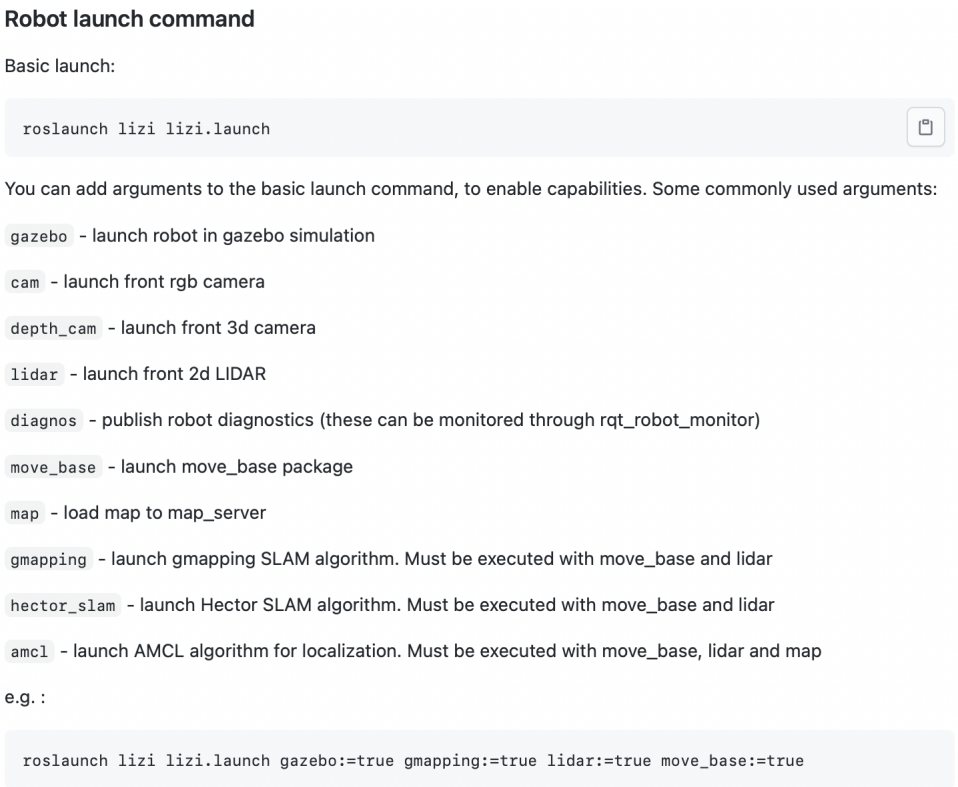
os anteriores, o que não é verdade. Este sistema tornou-se bastante mais fácil de analisar pois apresenta uma estrutura de código mais percetível e revela muita mais informação na sua documentação. Essa diferença na estruturação do código vai poder ser observada mais à frente com um exemplo de um *launch file*. De seguida vai ser apresentada uma lista das diretorias e ficheiros nelas existentes tal como foi feito nos restantes sistemas. Nesta lista também se vai perceber a maior diferença deste sistema.

- `lizi` - esta diretoria contém um *launch file* que pode ser considerado como o principal neste sistema. O ficheiro (`lizi.launch`) apresenta uma estrutura muito diferente do que se tem visto nos sistemas anteriores. O *launch file* possui uma grande quantidade de argumentos, sendo que a maioria é escolhida pelo utilizador quando lança o robô. Depois, dependendo dos argumentos que são escolhidos vão sendo incluídos ficheiros que ativam cada funcionalidade desejada.
- `lizi_control` - aqui é possível encontrar um ficheiro (`lizi_controllers.launch`) que ativa um nó com o poder de carregar e iniciar um conjunto de controladores, assim como parar e descarregar os mesmos.
- `lizi_description` - nesta diretoria podem-se encontrar dois ficheiros de teste:
  - `gazebo_test` - ficheiro que testa o robô num ambiente vazio;
  - `joints_test` - ficheiro que testa as articulações do robô.
- `lizi_hw` - diretoria com os ficheiros correspondentes ao *hardware* do sistema. Podem-se encontrar cinco *launch files*:
  - `d435_cam` - ficheiro que apresenta uma grande quantidade de argumentos e inclui um ficheiro referente a uma câmara;
  - `diagnostics` - a função deste ficheiro é a interpretação de mensagens de diagnóstico, processá-las, categorizar os dados e publicá-los num tópico específico;
  - `hokuyu_lidar` - ficheiro que ativa um sensor do tipo LiDAR;
  - `lizi_hw` - ficheiro que é incluído sempre que um ficheiro relacionado com o *hardware* é lançado;
  - `microsoft_cam` - este ficheiro ativa um nó referente a uma câmara USB (*Universal Serial Bus*).
- `lizi_navigation` - nesta diretoria podem-se encontrar os ficheiros necessários para que o robô possa realizar a sua locomoção, mapeamento e localização. Os cinco *launch files* existentes são os seguintes:
  - `amcl` - ficheiro que ativa um sistema de localização. Essa localização é realizada a partir da estimativa de posição em relação a um mapa conhecido;

- `gmapping` - este ficheiro permite SLAM baseado em *laser* de forma a construir um mapa do ambiente em que o robô se está a movimentar;
- `hector_slam` - ficheiro que ativa um nó para um sensor LiDAR baseado em SLAM, sem necessidade de medição de distância e com poucos recursos computacionais;
- `move_base` - neste *launch file* é ativado um nó que, dado um ponto no mapa, permite que o robô se desloque até esse mesmo ponto;
- `robot_localization` - este ficheiro, quando lançado, fornece uma estimativa do estado do robô através da utilização de sensores.

Com esta análise dos *launch files* já é possível perceber este sistema, assim como as suas características e funcionalidades. Como se pôde observar no primeiro ponto desta lista, existe um ficheiro principal que vai adicionando os ficheiros correspondentes às funcionalidades que o utilizador pretende. Esse mesmo utilizador pode escolher essas funcionalidades a partir dos argumentos que escolhe no momento do lançamento do robô.

Este sistema permite um ótimo entendimento do seu funcionamento apenas pela leitura do código, mas, para além disso, são ainda fornecidas bastantes informações importantes na documentação, tal como é possível verificar na Figura 4.6.


Robot launch command

Basic launch:

```
roslaunch lizi lizi.launch
```

You can add arguments to the basic launch command, to enable capabilities. Some commonly used arguments:

`gazebo` - launch robot in gazebo simulation

`cam` - launch front rgb camera

`depth_cam` - launch front 3d camera

`lidar` - launch front 2d LIDAR

`diagnos` - publish robot diagnostics (these can be monitored through rqt_robot_monitor)

`move_base` - launch move_base package

`map` - load map to map_server

`gmapping` - launch gmapping SLAM algorithm. Must be executed with move_base and lidar

`hector_slam` - launch Hector SLAM algorithm. Must be executed with move_base and lidar

`amcl` - launch AMCL algorithm for localization. Must be executed with move_base, lidar and map

e.g. :

```
roslaunch lizi lizi.launch gazebo:=true gmapping:=true lidar:=true move_base:=true
```

Figura 4.6: Documentação acerca da ativação do Lizi.

A partir desta informação relativa aos argumentos que estão disponíveis fica muito mais acessível a perceção do funcionamento do sistema, tanto para utilizadores como para programadores. Desta

documentação também é possível retirar informação sobre argumentos dependentes de outros, como é o caso dos três últimos argumentos que devem ser lançados com outros.

Tal como nos restantes sistemas robóticos aqui explorados e analisados, vai ser detalhado aqui um *launch file*[22] pertencente ao sistema. Na Listagem 4.5 está apenas parte do *launch file*, pois este apresenta grandes dimensões.

```
[...]
<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>

<include file="$(find espeak_ros)/launch/espeak_ros.launch" />

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

[22] lizi/lizi/launch/lizi.launch

```
37          </group>
38        </group>
39      </group>
40  </group>
41  [...]
```

Listagem 4.5: Parte de um exemplar de um *launch file* do sistema Lizi.

Tal como foi referido anteriormente, este é um ficheiro que apresenta bastantes argumentos. Nesta listagem não estão todos representados, mas podem ser vistos alguns, por exemplo, nas linhas 2, 6, 12, 18 e 19. Para além destes é possível observar mais sempre que é encontrada uma *tag* `group`. Isto significa que cada argumento pode representar uma funcionalidade do sistema. Quando o robô é lançado, este *launch file* testa os argumentos e quando estes são verdadeiros, tal como é percetível pelo código, é incluído um ou mais ficheiros que ativam uma determinada funcionalidade.

Como já é habitual na análise destes sistemas construiu-se uma tabela com os ficheiros que se consideraram incompatíveis pela presença de nós com o mesmo nome. No caso deste sistema apenas foram encontrados dois ficheiros com ativação de nós com o mesmo nome, esses ficheiros apresentam-se na Tabela 4.5.

Ficheiros incompatíveis	
`joints_test`	`lizi`

Tabela 4.5: Ficheiros incompatíveis no Lizi.

Este é um sistema um pouco diferente dos restantes, como se descreve acima. O facto de apenas incluir ficheiros dependendo do valor de argumentos leva a que a abordagem na programação da variabilidade possa ser considerada anotativa. Para efetuar a ativação do Lizi apenas é necessário lançar o *launch file* principal e definir os argumentos disponíveis. Esses argumentos irão dar origem depois ao lançamento de outros *launch files* que irão ativar as funcionalidades pretendidas.

4.2 Estratégias para Gestão da Variabilidade

Durante este capítulo foram explorados alguns sistemas ROS que ajudaram a perceber como é estruturado o código deste tipo de sistemas. Para além disso, foi também possível perceber qual a abordagem que a maioria deles utiliza para gerir a variabilidade.

Dos cinco sistemas analisados, quatro deles utilizavam principalmente uma abordagem composicional, enquanto que apenas um deles utilizava uma abordagem mais anotativa (Lizi). A abordagem utilizada nunca é única. Em alguns sistemas composicionais, por vezes são utilizadas argumentos, assim como no sistema anotativo também são utilizados ficheiros diferentes para representar as várias características. Manualmente, o sistema mais compreensível foi o Lizi, pois com as anotações tornou-se mais fácil de

perceber quais eram os argumentos que levavam a certas características, assim como encontrar os ficheiros correspondentes a cada uma delas. Os sistemas maioritariamente composicionais tornaram-se um pouco mais trabalhosos na medida em que foi necessário entender melhor o sistema e perceber melhor o que cada ficheiro fazia, para assim distinguir os ficheiros que realmente modificavam as funcionalidades, dos que apenas existiam para auxiliar esses.

Depois de toda a análise realizada e demonstrada na secção anterior, onde são também apresentados os sistemas estudados e conclusões dessas análises, podem-se obter algumas respostas quanto à gestão da variabilidade. Um dos principais elementos utilizados para gerir a variabilidade nestes sistemas, e identificado em todos eles, são os *launch files*. Em todos os casos estudados, os *launch files* têm sempre um papel muito importante na definição do que se poderá considerar uma possível característica (*feature*) do sistema.

Para além dos *launch files* foi encontrado mais um fator importante na gestão da variabilidade, os argumentos. Nos sistemas analisados, os argumentos foram muitas vezes cruciais para perceber algumas características do robô. No Lizi, por exemplo, os *launch files* são importantes para lançar certas *features*, mas o que determina quais os ficheiros a ser lançados são os argumentos. Assim, pode-se concluir que no Lizi, os argumentos são utilizados para facilitar a configuração do sistema pelo utilizador.

Outro elemento importante para a gestão da variabilidade encontrado a partir da análise dos sistemas anteriores é a ativação de nós. Os nós são bastante relevantes para perceber quais os *launch files* que são realmente importantes e quais aqueles que não podem ser lançados com outros. A partir destes pode-se perceber quais as características que são incompatíveis.

Com isto, é seguro concluir que os principais elementos para gestão da variabilidade em sistemas ROS são os *launch files*, pois todos os outros elementos aqui identificados fazem parte da estrutura de um ficheiro desse tipo. Estes ficheiros permitem a ativação dos sistemas e podem ser parametrizados por argumentos dados pelo utilizador. Assim, a partir dos *launch files* é possível fazer a gestão de quais as *features* que vão estar presentes dependendo da indicação do utilizador. É neles que se encontram os nós que o sistema vai ativar assim que for lançado, e é a partir deles que são determinadas as incompatibilidades e dependências das *features*. Apesar de tudo isto, é importante referir que existem outros possíveis tipos de variabilidade que não foram analisados por questão de simplicidade, como é o caso, por exemplo, do código fonte de cada nó existente. A partir de uma análise neste sentido poderia ser possível encontrar outras formas de gestão de variabilidade.

Outra das conclusões a que se chegou com esta análise foi o facto de existirem ficheiros com extensão `.launch.xml`. Estes apresentam uma estrutura semelhante com um ficheiro `.launch` mas apenas existem para ser incluídos. Apesar de ativarem nós e incluírem outros ficheiros, não é suposto lançá-los, mas sim incluí-los em *launch files*. Assim, ficheiros com extensão `.launch` servem para lançar partes do robô ou o robô completo e ficheiros `.launch.xml` apenas podem ser incluídos noutros.

No capítulo seguinte, todas estas conclusões vão ser utilizadas na definição de uma técnica para extrair automaticamente *feature models* de cada um destes sistemas. Essa técnica vai essencialmente analisar *launch files*, pois são os ficheiros mais relevantes para gestão da variabilidade em sistemas ROS.

Capítulo 5

Extrator de *Feature Models*

Neste momento, para se conseguir quantificar a variabilidade de um sistema ROS, através, por exemplo, de um *feature model*, é necessário analisar manualmente o código. Para que isso não seja necessário pensou-se em desenvolver uma ferramenta que ajude a gerar automaticamente os modelos de variabilidade. Em particular, pretende-se criar uma ferramenta capaz de construir um *feature model* a partir do código de um sistema ROS, podendo esse diagrama ser usado como ponto de partida na compreensão da respectiva variabilidade.

A ferramenta tem como primeiro objetivo conseguir encontrar *features* do sistema a partir do código deste. Depois de identificar as *features*, torna-se possível construir o *feature model*. Todo este processo implica uma análise dos *launch files*, o principal elemento para gestão da variabilidade, nomeadamente analisar como são construídos e como cada elemento é disposto no ficheiro, de maneira a se conseguir construir um *feature model* o mais preciso possível.

Decidiu-se que a linguagem utilizada para desenvolver a ferramenta seria o Python[1] por ser uma linguagem com características como a versatilidade, expressividade e simplicidade de sintaxe.

De seguida vai ser exposto ao detalhe todo o processo de extração de *features* e consequente construção do *feature model*, que funciona em três etapas. Cada uma das etapas aqui descritas vai apresentar os respetivos resultados, bem como a justificação de todas as decisões tomadas. Neste processo vão ser apenas demonstrados os resultados para um dos sistemas, sendo que para os restantes foi feito um processo igual, utilizando a ferramenta que se foi desenvolvendo. Durante cada etapa deste processo, até que se chegasse aos resultados pretendidos foram criadas novas ideias de como aprimorar o projeto e foram-se resolvendo problemas que se foram encontrando. Posto isto, cada fase aqui descrita vai ser bastante importante para perceber o caminho feito até à conclusão do projeto.

[1] https://www.python.org

5.1 Primeira Etapa

Visto que os *launch files* são o principal gestor da variabilidade e o que permite a execução dos sistemas, numa primeira fase de desenvolvimento o principal objetivo foi analisar bem esses ficheiros e retirar algumas informações que podem ajudar nas etapas seguintes. Assim procedeu-se ao desenvolvimento de uma ferramenta com as seguintes funcionalidades:

- Procura e contagem de *launch files*;
- Procurar a existência de nós em cada *launch file* encontrado;
- Encontrar os *launch files* incompatíveis pela ativação de nós com o mesmo nome;
- Encontrar os *launch files* incluídos noutros *launch files*;
- Construção de um *feature model* com estas informações.

As primeiras quatro funcionalidades foram pensadas para facilitar o trabalho posterior de criação do *feature model*. A procura de *launch files* é importante para se poder analisar cada um deles, tal como se fez manualmente e para poder perceber se devem ser considerados como *features* do modelo. A existência de nós dentro de cada *launch file* é relevante para perceber se esse ficheiro ativa algum nó, pois se esse for o caso será necessário entender se esse nó é único no sistema. Daí o terceiro ponto aqui destacado, que é a procura de ficheiros incompatíveis. Este ponto torna-se bastante relevante na parte de construção do modelo, pois estas incompatibilidades acabam por se tornar restrições do modelo. A quarta funcionalidade de busca de informações desenvolvida nesta fase é também fundamental para a criação de restrições, pois baseia-se na busca de *launch files* que dependem de outros, sendo que essa dependência é expressa através de `includes` no ficheiro. Com isto é possível criar um *feature model* onde as *features* são os *launch files* e as restrições são construídas a partir das incompatibilidades e dependências entre esses *launch files*. Aí entra a quinta e última funcionalidade desta etapa que se baseia na utilização das informações anteriores para a construção do *feature model*.

5.1.1 Procura e Contagem de *Launch Files*

Para a procura de *launch files* foi utilizada uma técnica que procura ficheiros com uma determinada extensão (neste caso `.launch`) dentro de diretorias. Para diferenciar ficheiros diferentes com nomes iguais foi utilizado o nome da diretoria onde cada um deles se encontra. Uma *feature* não deve iniciar com um algarismo, por isso esse problema foi resolvido colocando os algarismos que iniciam o nome no final. Na Tabela 5.1 encontra-se a esquematização destas duas soluções.

Antes	Depois
`3Dsensor`	`Dsensor3`
`Minimal`	`Kobuki_node_minimal`
`Minimal`	`Kobuki_auto_docking_minimal`

Tabela 5.1: Alteração do nome de *features* diferentes com nomes iguais e com algarismos no início.

A primeira linha da tabela é referente à alteração do nome quando este apresenta algarismos no início e a segunda e terceira linhas representam a mudança de nome de duas *features* com nome igual.

5.1.2 Procura da Existência de Nós

Para conseguir realizar a procura de nós nos *launch files*, o que se fez foi ir a cada ficheiro e, como o *launch file* tem um aspeto semelhante a um ficheiro `.xml`, fez-se uma busca pela *tag* `<node>`. Com isto, foram considerados todos os nós que poderiam ser lançados, mesmo aqueles que apenas se encontravam dentro de grupos condicionais. Desta forma foi possível retirar o nome de cada nó existente em cada *launch file*. Assim, conseguiu-se também relacionar cada ficheiro com todos os nós que este ativa.

5.1.3 Busca por *Launch Files* Incompatíveis

Com as informações anteriores acerca dos nomes dos ficheiros e dos nós que cada um ativa, para esta tarefa foi apenas necessário utilizar essa informação e comparar os nomes dos nós de *launch files* diferentes. Caso existissem *launch files* cujos nós tivessem o mesmo nome, então eram considerados incompatíveis, sendo essa informação guardada para a construção do modelo. Aqui os ficheiros incompatíveis já são guardados dois a dois de forma a facilitar o trabalho de construção do modelo.

5.1.4 Busca por *Launch Files* Dependentes

Para esta tarefa foi feito algo muito parecido com o realizado para descobrir os nós. Em cada ficheiros fez-se uma busca pela *tag* `<include>` e dentro dessa *tag*, se estivesse incluído um ficheiro `.launch` então considerava-se dependência. Esse procedimento foi realizado em todos os *launch files* e foi guardada toda essa informação, relacionando cada *launch file* com os ficheiros dos quais depende. Esta informação é também útil para a construção do *feature model*.

5.1.5 Construção do *Feature Model*

Dentre todas as tarefas desta etapa, a construção do *feature model* foi a menos complicada, pois apenas utiliza as informações recolhidas pelas tarefas anteriores para construir modelo. Na Secção 3.2 do Capítulo 3 foi descrita uma linguagem interessante para definir *feature models*, o TVL, e foi essa a linguagem

escolhida para fazer a construção dos *feature models* nesta ferramenta. Para isso teve-se em atenção todas as regras necessárias à criação de ficheiros desta linguagem, tal como descritos anteriormente.

Todos os ficheiros identificados foram convertidos em *features*. Os ficheiros que se consideraram incompatíveis deram origem a restrições do tipo a⟶!b (a negação é representada pelo ponto de exclamação). Uma restrição deste tipo significa que a escolha da *feature* a implica que a *feature* b não seja escolhida (quando uma existe a outra não pode existir). Em relação aos ficheiros que incluem outros e que foram considerados dependentes, esta relação é convertida em restrições de outro tipo a⟶b. Isto significa que a existência da *feature* a implica que a *feature* b exista no sistema. Isto acontece quando o ficheiro a inclui o ficheiro b, o que implica a sua existência conjunta sempre que o primeiro é selecionado.

Desta forma, toda a informação recolhida nas tarefas anteriores provou ser útil na construção do *feature model*. Na Listagem 5.1 e na Figura 5.1 encontra-se o resultado obtido desta etapa para o sistema Turtlebot. Na listagem apresenta-se o resultado que a ferramenta desenvolvida construiu, ou seja, um ficheiro TVL que representa o *feature model* para o sistema. Na figura está representado esse mesmo *feature model* de uma forma mais apelativa à visão humana. Este *feature model* foi construído manualmente utilizando o *plugin* featureIDE do Eclipse.

```
root Turtlebot_melodic {
   group allOf {
      opt Concert_minimal,
      opt Concert_client,
      opt Minimal,
      opt Dsensor3,
      opt Ps3_teleop,
      opt Keyboard_teleop,
      opt Logitech,
      opt Xbox360_teleop,
      opt Turtlebot2_bringup,
      opt Rgbd_sensor,
      opt Robot_state_publisher,
      opt Turtlebot_bringup,
      opt Depthimage_to_laserscan,
      opt Diagnostics
   }
   Ps3_teleop -> !Logitech;
   Ps3_teleop -> !Xbox360_teleop;
   Logitech -> !Xbox360_teleop;
   Turtlebot2_bringup -> !Turtlebot_bringup;
   Concert_minimal -> Minimal;
   Rgbd_sensor -> Dsensor3;
}
```

Listagem 5.1: Ficheiro TVL obtido na primeira etapa para o TurtleBot.

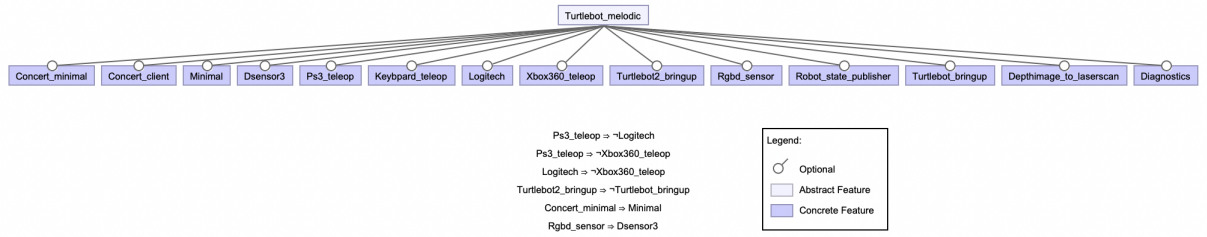


Figura 5.1: Representação gráfica do *feature model* resultante da primeira etapa para o TurtleBot.

Na Listagem 5.2 e na Figura 5.2 apresentam-se os resultados obtidos para o sistema Lizi. Tal como para o sistema anterior, a listagem mostra o resultado da ferramenta e a figura representa o modelo dessa listagem.

```
root Lizi_master {
   group allOf {
      opt Diagnostics,
      opt Lizi_hw,
      opt D435_cam,
      opt Hokuyu_lidar,
      opt Microsoft_cam,
      opt Gazebo_test,
      opt Joints_test,
      opt Lizi,
      opt Gmapping,
      opt Amcl,
      opt Move_base,
      opt Robot_localization,
      opt Hector_slam,
      opt Lizi_controllers
   }
   Joints_test -> !Lizi;
}
```

Listagem 5.2: Ficheiro TVL obtido na primeira etapa para o Lizi.

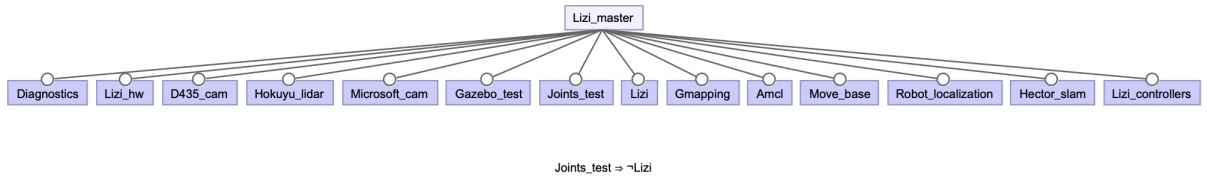


Figura 5.2: Representação gráfica do *feature model* resultante da primeira etapa para o Lizi.

Como se pode ver em ambos os sistemas, o `group allOf` foi convertido, na representação, numa relação and, sendo que todas as *features* `opt` no ficheiro são *features* opcionais na representação.

Nas etapas seguintes foi realizada uma separação entre sistemas com abordagens composicionais e anotativas, daí a utilização destes dois sistemas para exemplificar os resultados obtidos. Desta forma será possível acompanhar as melhorias realizadas para suportar cada uma das abordagens de programação da variabilidade.

5.2 Segunda Etapa

Como foi visto anteriormente, os argumentos são também usados na gestão da variabilidade. Por isso, nesta segunda etapa foi feita uma análise aos argumentos dos *launch files*. As funcionalidades desenvolvidas nesta etapa foram bastante importantes na construção do *feature model* de sistemas que apresentam uma abordagem anotativa. Neste caso, o único sistema desse tipo identificado no capítulo anterior é o Lizi, portanto vai ser utilizado esse sistema para ilustrar o que foi feito nesta etapa. O que se implementou nesta etapa foi:

- Procura de argumentos;
- Pesquisa acerca dos argumentos que deverão tornar-se *features* e os que não apresentam atributos para isso, assim como a criação de uma ligação entre estes e os *launch files* que dependem deles;
- Procura das diretorias referentes a cada *launch file*;
- Construção do *feature model* baseado nestas informações, incluindo também as da etapa anterior.

Para a criação do *feature model*, tal como na etapa anterior, tiveram que se recolher algumas informações relevantes. Essas informações são traduzidas pelas primeiras três funcionalidades. A procura por argumentos serve para que se possam utilizar na criação do modelo pois existem argumentos que podem ser considerados *features*. A segunda funcionalidade é importante na escolha dos argumentos que realmente se poderão considerar *features*. Para isso foram definidas algumas regras, pois a quantidade de argumentos existentes em cada *launch file* é demasiado grande para que qualquer argumento seja uma *feature*. As regras de escolha de argumentos passam pela inclusão de *launch files* (explicado mais à frente). Para além disso, vai existir uma relação entre os argumentos e alguns *launch files* e essa relação é importante para a criação do modelo. A terceira funcionalidade de pesquisa (procura de diretorias) visa ajudar a criar uma hierarquia no modelo. As diretorias, no modelo, são utilizadas para agrupar as *features* num grupo que as defina de maneira a obter um *feature model* mais completo e explícito. Com estas informações é possível construir um *feature model* mais preciso que utiliza as informações da etapa anterior e as novas, obtidas nesta etapa. O *feature model* que será resultante daqui terá as diretorias como *feature* pai, sendo os *launch files* de cada uma as *features* filho. Depois disso, os argumentos que se considerarem *features* estarão num nível inferior ao *launch file* a que pertencem.

5.2.1 Procura de Argumentos

Para a procura de argumentos foi utilizada a mesma técnica da procura de nós. Nos *launch files*, quando um argumento está a ser definido tem uma *tag* `<arg>` que contém no seu interior várias informações desse argumento, tais como o nome e o valor *default*. Para a concretização desta funcionalidade é apenas necessário guardar o nome do argumento e o ficheiro a que pertence. No final, ter-se-á uma lista com todos os argumentos de cada *launch file*. A grande quantidade de argumentos existentes em sistemas ROS levou à necessidade de perceber quais os argumentos que realmente podem ser considerados *features*, daí a próxima funcionalidade.

5.2.2 Promoção de Argumentos a *Feature* e Respetiva Ligação com *Launch Files*

Nesta funcionalidade utilizaram-se os resultados obtidos na anterior de maneira a perceber se um certo argumento deveria tornar-se *feature* aquando da construção do *feature model*. Esta é uma informação importante na medida em que reduz significativamente o número de argumentos a tratar como *feature* e simplifica, mais tarde, a construção do *feature model*. Para a escolha dos argumentos que deveriam tornar-se *features* foram estabelecidos alguns critérios:

- Um argumento é considerado *feature* quando causa a inclusão de *launch files* que ativam nós;
- Um argumento é também considerado *feature* quando provoca a inclusão de um *launch file* que, mesmo não ativando nós, inclui outros que o façam.

Na Figura 5.3 está representado o que foi descrito na lista anterior. Um argumento é considerado *feature* sempre que implique a inclusão de um *launch file* que ative um nó, mas quando isso não acontece, se esse ficheiro incluir outros ficheiros que ativem nós, então também é considerado *feature*.

Figura 5.3: Representação dos critérios que tornam um argumento numa *feature*.

Quando um argumento se encaixa num destes critérios é promovido a *feature*. Para isso é criada uma lista com todos os argumentos que são *features* correspondente a cada *launch file*.

Para a busca da ativação de nós, sempre que um argumento implicava a inclusão de um outro *launch file*, acedia-se a esse mesmo ficheiro e procurava-se pela ativação de nós. Caso não fosse encontrado

nenhum nó, procurava-se por inclusões: caso existissem realizava-se o mesmo procedimento, caso contrário o argumento não seria considerado *feature*. Para além destes casos observou-se um outro caso relevante: quando é incluído um ficheiro externo ao projeto, e portanto inacessível, considerou-se que o argumento seria na mesma uma *feature*, pois esse ficheiro incluído pode corresponder a uma funcionalidade importante do sistema.

Assim como os argumentos, também é importante guardar os ficheiros que são incluídos a partir desses argumentos e associá-los a estes. Para clarificar esta ideia, na Listagem 5.3 está representado apenas um excerto do código do ficheiro principal do sistema Lizi.

```
1  [...]
2  <group if="$(arg robot_localization)">
3        <include file="$(find lizi_control)/launch/lizi_controllers.launch" >
4           <arg name="enable_mbc_odom_tf" value="false"/>
5        </include>
6  </group>
7  [...]
8  <group if="$(arg robot_localization)">
9        <include file="$(find lizi_navigation)/launch/robot_localization.launch" />
10 </group>
11 [...]
```

Listagem 5.3: Excerto do *launch file* principal do Lizi.

Como se pode observar, neste caso é testada a validade do argumento `robot_localization`. No caso do valor do argumento ser *true*, o ficheiro `lizi_controllers.launch` é incluído. O mesmo se verifica no `group if` seguinte, com o mesmo argumento: caso o valor seja *true* o ficheiro `robot_localization.launch` é incluído e fica assim relacionado com o argumento que permitiu a sua inclusão. Assim, o que a ferramenta desenvolvida faz nesta fase é o seguinte:

- Quando o argumento é considerado *feature* pelos critérios demonstrados antes, cria-se uma ligação entre esse argumento e o ficheiro incluído.

Esta é uma funcionalidade interessante quando chega o momento de construção do modelo, pois estas relações entre *launch files* e os argumentos que permitem a sua inclusão vão ser convertidas em algo que ajuda na precisão do modelo. Essa transformação vai ser explicada de seguida, onde se refere o que foi feito para a construção do *feature model* a partir das informações recolhidas.

5.2.3 Procura de Diretorias

Esta procura de diretorias baseia-se na análise do caminho que se faz até chegar a um determinado *launch file*. Para o entendimento do funcionamento dos sistemas ROS foi realizada uma pesquisa onde foram analisados vários sistemas, sendo que apenas cinco foram minuciosamente explorados e aqui

referidos. Aquando dessa investigação percebeu-se que existiam alguns aspetos em comum entre todos os sistemas: todos os *launch files* podem ser encontrados dentro de diretorias com o nome `launch`. No entanto, essas diretorias não se encontram na raiz do projeto, mas sim dentro de uma outra diretoria que geralmente simboliza algo em comum entre todos os *launch files* que esta contém. Este facto é percetível durante o Capítulo 4 ao longo da análise dos *launch files* de cada sistema.

Com esta observação decidiu-se utilizar as diretorias para dar alguma facilidade de leitura ao modelo, criando uma melhor organização das *features*. Para isso, o que se fez foi encontrar as diretorias correspondentes às *features* previamente identificadas, ou seja, encontrar as diretorias de topo onde estão inseridos os *launch files* que foram considerados *features*. Durante a análise dos sistemas percebeu-se que nem sempre os *launch files* estão directamente dentro da diretoria `launch`. Na Figura 5.4 são apresentados exemplos de dois dos sistemas que foram analisados.

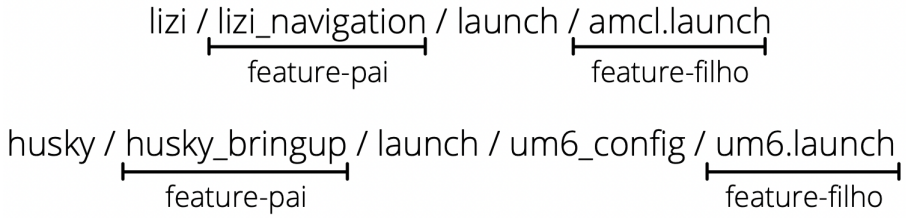


Figura 5.4: *Path* para um *launch file* considerado *feature* em dois sistemas diferentes.

Como se pode observar, na raiz do projeto encontra-se sempre uma diretoria onde se encontra outra com o nome `launch`, mas dentro dessa podem não estar diretamente todos os *launch files*. No caso do Lizi isso acontece, mas no caso do Husky o mesmo não se passa. O método utilizado para contornar estas diferenças foi o seguinte:

- Na *path* de cada *launch file* procurar a diretoria `launch`;
- Considerar com *feature* pai da *feature* associada ao *launch file* a diretoria imediatamente antes dessa.

Deste modo é possível encontrar todas as diretorias que têm *launch files*, de maneira a poder utilizá-las para a criação da hierarquia e organização no *feature model* a construir.

Para além da associação de cada *launch file* à diretoria correspondente, foi construída também uma lista com todas essas diretorias para facilitar a construção do modelo, como se vai poder observar na secção seguinte.

5.2.4 Construção do Modelo

Para a construção do *feature model*, tal como na etapa anterior, foi utilizada toda a informação recolhida. Esta etapa foi concebida principalmente para sistemas que utilizam uma abordagem anotativa no seu código, como é o caso do Lizi, o que não significa que não possa ser utilizada nos outros sistemas.

Durante a construção do modelo verificaram-se duas particularidades:

- Existem *launch files* com o mesmo nome de alguns argumentos;
- Existem diretorias que apenas contêm um *launch file*.

O primeiro ponto acarreta um problema pois não podem existir *features* com o mesmo nome no modelo, e tanto os ficheiros como os argumentos podem ser considerados *features*. O segundo ponto identifica uma situação que poderá levar a uma melhoria no *feature model* a construir. Para lidar com estas duas situações fez-se o seguinte:

- Aquando da construção do modelo, foi acrescentado, no final do nome da *feature* associada a cada *launch file* a terminação `_launch`. Deste modo ficou possível perceber quais as *features* que eram referentes a *launch files* e as referentes a argumentos, com a vantagem de garantir a inexistência de *features* com o mesmo nome;
- Sempre que uma diretoria com apenas um *launch file* era encontrada, essa diretoria era descartada e ficava apenas o *launch file* como *feature*. A inserção da diretoria nestes casos não iria facilitar a leitura do mesmo.

Tendo definido estas soluções pode-se então construir o modelo. O procedimento é bastante parecido ao da etapa anterior visto que é utilizada a informação recolhida e a estrutura do modelo não é muito diferente. Assim, o método desenvolvido para a construção do modelo foi a seguinte:

- Em primeiro lugar coloca-se a *feature* raiz do modelo, que tal como anteriormente, corresponde ao nome do projeto;
- Abaixo da raiz, dentro de um `group allOf` foram colocados os nomes das *features* originárias das diretorias ou ficheiros (no caso da diretoria apenas conter um ficheiro);
- Ainda dentro da raiz foram colocadas todas as incompatibilidades e dependências que foram recolhidas na etapa anterior. Neste campo, a novidade é a existência de novas dependências, ou seja, as dependências de ficheiros em relação a argumentos. A criação de ligação entre argumentos e os *launch files* incluídos é aqui utilizada para criar uma dependência do tipo `argumento` → `ficheiro`;
- Criação da hierarquia. Para tal, em primeiro lugar coloca-se o nome das diretorias que se considerou *features* e dentro destas, num `group someOf` todos os *launch files* que contêm. Decidiu-se optar por um `group someOf` para que pelo menos um dos ficheiros tivesse que ser escolhido, assim o facto de existir a diretoria não irá prejudicar o número de produtos válidos. Este procedimento é repetido até que não existam mais diretorias para explorar;

- No final do modelo são colocados os grupos de argumentos que foram considerados *features*, assim coloca-se o nome do *launch file* e de seguida, num `group allOf` todos os argumentos desse ficheiro que foram considerados *feature*. Neste caso foi escolhido criar um `group allOf`, pois existe a possibilidade de não escolher nenhum argumento no ficheiro. Este procedimento é repetido em todos os ficheiros com argumentos considerados *features*.

Na Listagem 5.4 encontra-se o resultado obtido nesta etapa para o sistema Lizi. Todos os sistemas foram testados com este método mas apenas o Lizi e o Husky tiveram resultados interessantes. Para os restantes sistemas, como não utilizam os argumentos para incluir *launch files*, obteve-se um *feature model* diferente do obtido na etapa anterior, apenas com melhorias na hierarquia.

```
root Lizi_master {
   group allOf {
      opt Lizi_hw,
      opt Lizi_description,
      opt Lizi_launch,
      opt Lizi_navigation,
      opt Lizi_controllers_launch
   }
   Joints_test_launch -> !Lizi_launch;
   Diagnos -> Diagnostics_launch;
   Amcl -> Lizi_controllers_launch && Amcl_launch;
   Robot_localization -> Lizi_controllers_launch && Robot_localization_launch;
   Depth_cam -> D435_cam_launch;
   Cam -> Microsoft_cam_launch;
   Lidar -> Hokuyu_lidar_launch;
   Gmapping -> Gmapping_launch;
   Hector_slam -> Hector_slam_launch;
   Move_base -> Move_base_launch;
}
Lizi_hw {
   group someOf {
      Diagnostics_launch,
      Lizi_hw_launch,
      D435_cam_launch,
      Hokuyu_lidar_launch,
      Microsoft_cam_launch
   }
}
Lizi_description {
   group someOf {
      Gazebo_test_launch,
      Joints_test_launch
   }
```

```
}
Lizi_navigation {
   group someOf {
      Gmapping_launch,
      Amcl_launch,
      Move_base_launch,
      Robot_localization_launch,
      Hector_slam_launch
   }
}
Lizi_launch {
   group allOf {
      opt Diagnos,
      opt Amcl,
      opt Robot_localization,
      opt Depth_cam,
      opt Cam,
      opt Lidar,
      opt Gmapping,
      opt Hector_slam,
      opt Move_base
   }
}
```

Listagem 5.4: Ficheiro TVL obtido na segunda etapa para o Lizi.

Na Figura 5.5 está a representação visual do modelo no código anterior, onde é possível verificar os ajustes que foram feitos.

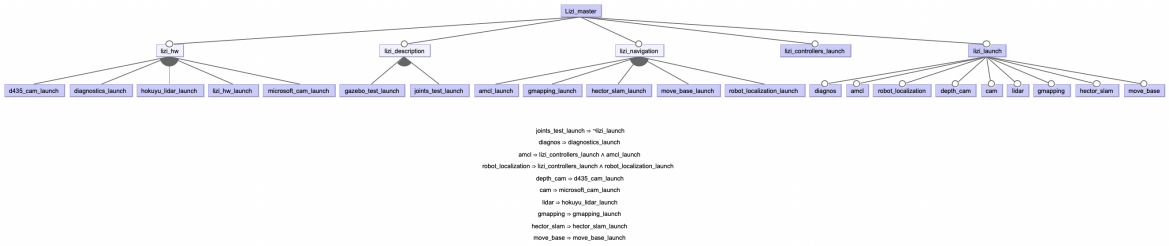


Figura 5.5: Representação gráfica do *feature model* resultante da segunda etapa para o Lizi.

Como é possível observar, no resultado da ferramenta são utilizados os `group allOf` e `someOf` que na representação são convertidos em relações `and` e `or` respetivamente. Quando um grupo é do tipo `someOf`, isto significa que tem que ser escolhida uma das *features* abaixo, ou seja, a *feature* pai correspondente a essas apenas existe para as agrupar, e não vai ter impacto nos produtos finais. Assim, na conversão para a representação sentiu-se a necessidade de colocar essa *feature* pai como uma *feature* abstrata pois não simboliza nenhuma característica específica do sistema. Ainda sobre a representação gráfica pode-se referir que inclui também todas as restrições encontradas pela ferramenta.

Com esta representação torna-se mais fácil perceber a diferença que faz a existência ou não das diretorias para a organização e compreensão do sistema. Nesta etapa já é possível visualizar uma árvore com hierarquia e organização, enquanto que na primeira etapa apenas estavam referidos os *launch files* existentes.

5.3 Terceira Etapa

Nesta etapa também são tratados os argumentos, mas desta vez numa perspectiva mais voltada para os sistemas com abordagens composicionais. Tal como nas etapas anteriores, vão ser retiradas algumas informações dos sistemas que vão ser depois utilizadas para a construção do *feature model*. Antes da recolha dessas informações foi necessário perceber a tática utilizada pelos sistemas composicionais na utilização dos argumentos. Na Listagem 5.5 é possível verificar que nestes sistemas são utilizados os argumentos diretamente na inclusão de outros ficheiros.

```
1 <include file="$(find turtlebot_bringup)/launch/includes/3dsensor/$(arg 3d_sensor).launch.xml"
    ↪ >
```

Listagem 5.5: Inclusão de um ficheiro dependente de um argumento (retirado do TurtleBot).

Como se pode observar neste exemplo, o argumento `3d_sensor` é utilizado neste `include` de maneira a ser substituído pelas várias opções. Ora, essas opções de substituição têm que ser descobertas de alguma forma, primeiro teve que se fazer uma análise manual e só posteriormente é que se passou para o desenvolvimento na ferramenta. Assim, foram-se procurar os ficheiros que serviriam para substituir naquele `include` e encontrou-se o que está representado na Figura 5.6.

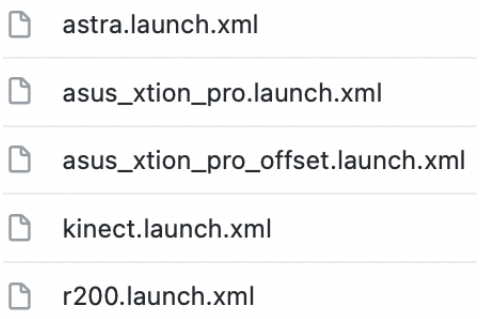


Figura 5.6: Conteúdo da diretoria onde estão os ficheiros substituíveis no argumento.

Com isto, pode-se concluir que as opções de substituição do argumento `3d_sensor` são: `astra`, `asus_xtion_pro`, `asus_xtion_pro_offset`, `kinect` e `r200`. Com isto em mente, foi criado um procedimento que ajudou a reunir mais informações acerca da forma como são utilizados os argumentos neste tipo de sistemas, e que é o seguinte:

- Procurar os *launch files* com inclusões que, dependendo do valor do argumento varia o ficheiro incluído, assim como a extração do nome do ficheiro e do argumento;

- Procurar os valores possíveis dos argumentos presentes nas inclusões;
- Criação de ligação entre os argumentos e os *launch files*, assim como a criação de uma associação entre os argumentos e as respetivas opções de substituição que irão acabar por ser consideradas *features*, tal como os argumentos;
- Verificação das igualdades de nomes de argumentos, *launch files* e opções de substituição;
- Construção do modelo.

Toda esta recolha de informação é importante, pois é a partir dela que vai ser construído o *feature model* e assim tentar chegar a uma versão o mais precisa possível das *features* existentes num sistema.

Aqui, a procura dos *launch files* com inclusões onde o ficheiro incluído varia com o valor do argumento é necessária para encontrar os argumentos que deverão ser considerados *features*. Para além disso é importante perceber quais são as opções para esses argumentos, pois tal como foi visto anteriormente, cada argumento apresenta vários valores possíveis e esses valores também vão ser referidos no modelo, daí o segundo ponto presente na lista acima. A criação de uma ligação entre os ficheiros e os argumentos e também com os possíveis valores desses argumentos é importante para a criação da hierarquia correta no modelo. Desta forma será possível perceber quais os valores que cada argumento pode ter, assim como em que ficheiro está presente esse argumento. A verificação de nomes iguais é muito importante em qualquer etapa de desenvolvimento desta ferramenta pois é necessário que todas as *features* do modelo tenham nomes diferentes, pois de outra forma será impossível fazer a contagem dos produtos e pode haver confusão na identificação das *features*. Para a construção do modelo todas estas informações vão ser utilizadas da seguinte maneira: tal como foi feito na etapa anterior, todas as diretorias vão estar dentro do grupo da raiz, sendo que depois disso cada uma vai ter espaço para os seus ficheiros, depois disso vai ser iniciada a parte dos argumentos e dos respetivos valores, o que vai criar um modelo mais preciso do que os obtidos na primeira etapa.

5.3.1 Procura de *Launch Files* com Inclusões Parametrizadas

As inclusões que se procuram durante esta etapa não são iguais às encontradas durante a primeira etapa, mas sim do tipo do que está representado na Listagem 5.5, vista anteriormente. Isto é, estas inclusões têm que conter a *tag* `arg` que vai servir para fazer a parametrização dependendo do valor dado ao argumento. Assim, é importante descobrir quais são os *launch files* que possuem este tipo de inclusões para que, aquando da construção do modelo seja possível criar a hierarquia correta.

O método que se utilizou para a procura e guarda dos nomes dos ficheiros foi a seguinte:

- Procura-se no *launch file* se existem `includes`;
- Se existirem tem que se verificar se é uma inclusão normal ou se utiliza argumentos para criar várias oportunidades de inclusão;

- Caso se verifique que é uma inclusão com base em argumentos guarda-se o nome do *launch file* que se está a analisar;
- Se a inclusão for normal e o ficheiro incluído for um *launch file* então não é necessário implementar nada, pois esse caso já foi considerado nas etapas anteriores;
- Se for uma inclusão normal mas o ficheiro incluído é um outro tipo de ficheiro, como por exemplo `.launch.xml`, terá que se ver se esse ficheiro apresenta inclusões;
- Caso tenha inclusões é necessário averiguar o tipo de inclusão, seguindo o processo já descrito. No caso de haver inclusões parametrizadas com argumentos, o nome a registar é sempre o do ficheiro onde se iniciou a busca, pois é esse que será considerado uma *feature* e indiretamente dá origem a essa inclusão.

Na Figura 5.7 está representado este método de forma mais sucinta com uma espécie de fluxograma. Aqui é importante perceber que o ficheiro do qual se retira o nome é sempre o *launch file* inicial, ou seja, aquele onde se inicia a análise.

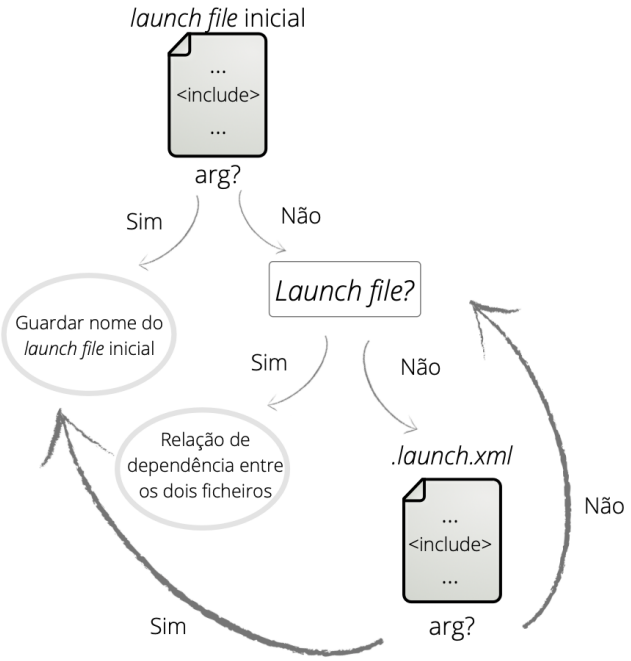


Figura 5.7: Método de procura e guarda de nomes de ficheiros.

Assim que se encontra um `include` que corresponda aos critérios anteriores, o nome do argumento utilizado para a inclusão é guardado juntamente com o nome do ficheiro. Isto permite que, mais tarde, seja possível criar uma hierarquia desde o ficheiro que contém a inclusão até ao argumento.

5.3.2 Procura dos Valores Possíveis dos Argumentos

Para procurar os valores possível de um argumento foi necessário analisar muito bem a forma como são utilizados. Tal como foi visto na Listagem 5.5 o nome do argumento encontra-se na *path* para o ficheiro

que se quer incluir. Na Listagem 5.6 apresentam-se dois exemplos dessa utilização no TurtleBot.

```
1 <include file="$(find turtlebot_bringup)/launch/includes/3dsensor/$(arg 3d_sensor).launch.xml"
      ↪ >
2 ----------------------------------------------------------------------------------------------
3 <include file="$(find turtlebot_bringup)/launch/includes/$(arg base)/mobile_base.launch.xml">
```

Listagem 5.6: Dois exemplos de inclusão de um ficheiro dependente de um argumento (retirados do TurtleBot).

O primeiro caso já foi visto antes e corresponde ao caso em que os ficheiros presentes na diretoria `3dsensor` representavam os valores que o argumento `3d_sensor` pode tomar. No entanto, o segundo caso é um pouco diferente, o que implicou generalizar a forma como se trata este tipo inclusão. Neste segundo caso os valores possíveis que o argumento `base` pode tomar estão representados pelas diretorias dentro da diretoria `includes` que contenham no seu interior um ficheiro chamado `mobile_base.launch.xml`. Essa pesquisa foi realizada manualmente ainda em fase de análise e chegou-se à conclusão que existem três opções para este argumento, tal como se pode observar na Figura 5.8.

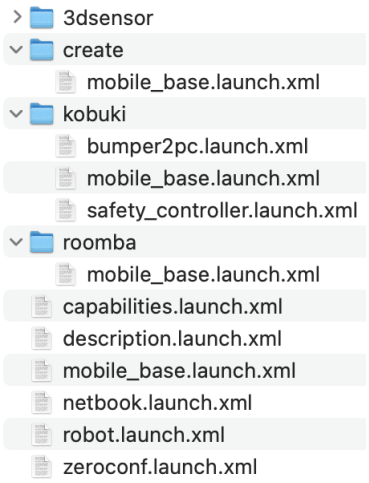


Figura 5.8: Conteúdo da diretoria `includes` do sistema TurtleBot.

Com esta figura pode-se facilmente perceber que dentro da diretoria `includes` existem quatro diretorias, ou seja, o argumento `base` tem pelo menos quatro opções de substituição. No entanto, uma delas (`3dsensor`) já foi analisada anteriormente e apresenta apenas ficheiros com nomes diferentes do pedido (`mobile_base.launch.xml`), por isso apenas sobram três para valor do argumento. Procedeu-se portanto à sua abertura e pode-se verificar pela figura que, de facto, existe no seu interior um ficheiro com o nome exigido.

O que se fez para tornar esta análise automática foi ir à *path* para o ficheiro a incluir, substituir o `$(arg nome_do_argumento)` por um '*' e depois disso fazer a pesquisa de todas as diretorias que se encaixem nestes critérios.

5.3.3 Criação de Ligação entre *Launch Files*, Argumentos e Valores

Após recolher as informações acerca dos *launch files*, argumentos e das opções de valores que estes podem tomar, é possível criar uma ligação entre estes elementos. Esta ligação vai ser muito útil quando se for construir o modelo pois permite definir a hierarquia no mesmo. Assim, o que se fez foi criar um dicionário em Python que permite reunir toda a informação utilizando etiquetas para identificar os índices, ao contrário das listas cujos índices são números inteiros ordenados. Desta forma, durante a construção do modelo será possível chegar aos valores pretendidos a partir dos índices personalizados. Com esta ligação obtiveram-se os seguintes resultados representados na Listagem 5.7, onde é possível perceber quais os nomes dados aos índices e quais os resultados obtidos para cada campo.

Uma característica que também é importante notar é a existência de dois argumentos com o mesmo nome e, consequentemente opções de valores com nomes iguais também. Isto significa que construir um *feature model* com as informações que se tem no momento iria originar um erro, pois iam existir *features* diferentes com o mesmo nome, o que não pode acontecer.

```
1 {'ficheiro': '3dsensor', 'arg': '3d_sensor', 'filhos': ['asus_xtion_pro_offset', 'kinect', '
      ↪ asus_xtion_pro', 'astra', 'r200']}
2 {'ficheiro': 'concert_client', 'arg': 'base', 'filhos': ['kobuki', 'roomba', 'create']}
3 {'ficheiro': 'minimal', 'arg': 'base', 'filhos': ['kobuki', 'roomba', 'create']}
```

Listagem 5.7: Resultado da ligação dos *launch files* aos argumentos e valores respetivos com existência de nomes iguais.

5.3.4 Verificação da Existência de Nomes de *Features* Iguais

Na tarefa anterior encontrou-se um problema que envolvia a existência de *features* diferentes com o mesmo nome. Para resolver esse problema teve que se recorrer a uma técnica para alterar esses nomes de maneira a distinguir essas *features*. Para fazer essa distinção, o que se fez foi acrescentar o nome do ficheiro correspondente no final do nome do argumento, assim conseguiu-se obter argumentos com nomes diferentes. Apesar desta mudança podem continuar a existir *features* com o mesmo nome, nomeadamente nas opções de substituição dos argumentos, que no dicionário representado anteriormente se encontram na chave `'filhos'`. O que se fez para resolver isso foi exatamente o mesmo que para os argumentos, garantindo-se assim um nome diferente para cada *feature*. A inexistência de nomes iguais previne que haja ambiguidades no momento de identificação das *features*.

Na Listagem 5.8 pode-se verificar a mudança de nomes dos argumentos e opções que antes estavam iguais. O facto de os nomes ficarem muito maiores pela adição do nome do ficheiro pode ser confuso nesta fase, mas não vai ser um problema quando se construir o modelo.

```
1 {'ficheiro': '3dsensor', 'arg': '3d_sensor', 'filhos': ['asus_xtion_pro_offset', 'kinect', '
      ↪ asus_xtion_pro', 'astra', 'r200']}
```

```
2 {'ficheiro': 'concert_client', 'arg': 'base_concert_client', 'filhos': ['kobuki_concert_client
    ↪ ', 'roomba_concert_client', 'create_concert_client']}
3 {'ficheiro': 'minimal', 'arg': 'base_minimal', 'filhos': ['kobuki_minimal', 'roomba_minimal',
    ↪ 'create_minimal']}
```

Listagem 5.8: Resultado da ligação dos *launch files* aos argumentos e valores respetivos depois da alteração dos nomes.

5.3.5 Construção do Modelo

Nesta etapa, tal como nas anteriores, as informações recolhidas são utilizadas e distribuídas pelo *feature model*. O método utilizado para construir um modelo sintaticamente correto foi bastante similar às etapas anteriores. Algumas das informações que se encontram nesta etapa forçam a uma estrutura um pouco diferente de forma a criar um modelo mais compreensível e simples. Esta é uma etapa que foi planeada para sistemas cuja abordagem de programação é composicional, por isso, o sistema que vai ser aqui utilizado como exemplo será o TurtleBot.

Na lista a seguir estão apresentados todos os passos necessários até chegar ao resultado final, assim como uma breve explicação do porquê de cada decisão:

- Em primeiro lugar, na raiz do modelo, como já é habitual, colocou-se o nome do projeto como *feature*, sendo qualquer *feature* descendente desta;
- Logo abaixo da raiz, num `group allOf`, são colocadas todas as diretorias que apresentam *launch files*, que são *features* auxiliares para melhorar a organização do modelo. Nesta etapa, tal como na anterior, as diretorias que apenas contiverem um *launch file* são descartadas, sendo que neste nível do modelo aparecerá então a *feature* correspondente ao mesmo;
- Depois, ainda dentro da raiz são adicionadas as restrições do sistema, ou seja, as incompatibilidades e dependências;
- Já fora da *feature* raiz, começa a parte em que são detalhadas as diretorias. Cada uma tem um `group someOf` onde se colocam todos os *launch files* que, depois de analisados, se consideraram *features*. Estes ficheiros estão dentro de um grupo `someOf` para ser escolhido pelo menos um, pois a diretoria apenas existe para agrupar *features* e facilitar a leitura do modelo;
- Depois de todas as diretorias estarem descritas serão detalhados os ficheiros e os seus argumentos. Para cada *launch file* os respetivos argumentos são colocados dentro de um `group allOf`. Para cada argumento, são agrupadas todas as suas opções para valores num `group oneOf`. Os argumentos estão dentro de um `group allOf`, no entanto, esses argumentos não têm a *tag* `opt` tal como antes, pois é obrigatória a escolha do argumento. Por outro lado, os valores possíveis estão num `group oneOf` o que implica que tem que ser escolhido um e um só valor.

Na Listagem 5.9 pode-se observar o resultado obtido pela ferramenta para o sistema TurtleBot. Nesta etapa, tal como nas anteriores, todos os sistemas foram testados, mas apenas o TurtleBot e o TurtleBot3 beneficiaram dos mecanismos implementados, pois são os únicos que usam os argumentos para parametrizar inclusões. O Lizi, como se viu, não utiliza os argumentos da mesma forma. O Husky também utiliza os poucos argumentos que tem da mesma forma que o Lizi, e no Kobuki os argumentos que utiliza não se revelaram importantes na variabilidade do sistema (não afetam a ativação de nós).

```
root Turtlebot_melodic {
   group allOf {
      opt Turtlebot_bringup,
      opt Turtlebot_teleop,
      opt Providers
   }
   [...]
}
Turtlebot_bringup {
   group someOf {
      Concert_minimal_launch,
      Concert_client_launch,
      Minimal_launch,
      Dsensor3_launch
   }
}
Turtlebot_teleop {
   group someOf {
      [...]
   }
}
Providers {
   group someOf {
      [...]
   }
}
Dsensor3_launch {
   group allOf {
      D_sensor3 {
         group oneOf {
            Asus_xtion_pro_offset,
            Kinect,
            Asus_xtion_pro,
            Astra,
            R200}}}}

Concert_client_launch {
```

```
38    group allOf {
39       Base_concert_client {
40          group oneOf {
41             Kobuki_concert_client,
42             Roomba_concert_client,
43             Create_concert_client}}}}
44
45 Minimal_launch {
46    group allOf {
47       Base_minimal {
48          group oneOf {
49             Kobuki_minimal,
50             Roomba_minimal,
51             Create_minimal}}}}
```

Listagem 5.9: Ficheiro TVL obtido na terceira etapa para o TurtleBot.

Algumas partes deste ficheiro foram cortadas (espaços onde se encontram `[...]`) para minimizar o tamanho do ficheiro. O primeiro corte foi realizado nas restrições que já tinham sido observadas na Secção 5.1 e os outros são relativos aos ficheiros existentes em duas diretorias.

Na Figura 5.9 pode-se ver a representação deste ficheiro sem partes cortadas, sendo possível observar todas as *features* e compreender melhor as decisões tomadas.

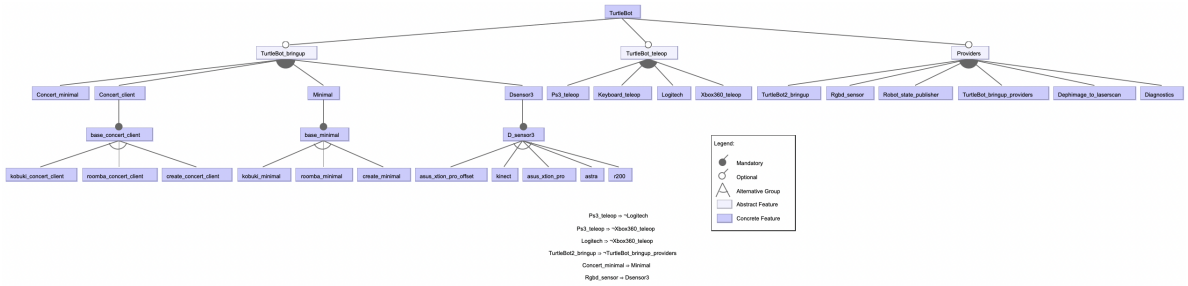


Figura 5.9: Representação gráfica do *feature model* resultante da terceira etapa para o TurtleBot.

Capítulo 6

Refinamento Manual e Avaliação dos Resultados

Neste capítulo vai ser descrito o refinamento manual que foi realizado nos *feature models* obtidos com a técnica automática descrita no capítulo anterior. Esse refinamento tem o objectivo de melhorar a precisão dos *feature models*, por forma a obter modelos que melhor quantifiquem a variabilidade dos sistemas. Alguns dos refinamentos têm por objectivo embelezar os mesmos. O embelezamento serve apenas para tornar os modelos mais apelativos e compreensíveis, sendo que não altera o número de produtos finais.

Os *feature models* refinados também serão importantes para perceber se a ferramenta de extração foi desenvolvida com sucesso. Para isso vão ser analisados os resultados antes e depois do refinamento manual, para se poder perceber o quão diferentes são as duas soluções. Deste modo também será possível apontar eventuais problemas da ferramenta que possam ser corrigidos no futuro.

6.1 Refinamento Manual

Nesta secção vai ser realizado um aperfeiçoamento dos *feature models* obtidos automaticamente, de maneira a melhorar a precisão dos mesmos. Todas as decisões tomadas serão devidamente justificadas para que seja percetível o motivo que levou a cada escolha. Este refinamento irá ainda incluir uma simplificação dos modelos em termos visuais, recorrendo a operações de reestruturação (*refactoring*) que deixam o número de produtos finais inalterado. Já foram realizados vários estudos acerca de métodos de reestruturação de *feature models* que preservam o número de produtos finais [4, 25], sendo que as reestruturações aplicadas nesta fase são inspiradas nesses estudos.

6.1.1 Kobuki

Automaticamente foi obtido um *feature model* bastante grande, principalmente devido à existência de ficheiros `standalone`. Apesar disso, também se encontraram bastantes restrições, o que ajuda a reduzir

o número de produtos finais. No entanto, existem algumas alterações que se podem efetuar para que este modelo se torne ainda mais exato e correto. Uma dessas alterações é a eliminação de uma *feature* que a ferramenta considerou, mas que apenas serve para teste, o que faz dela irrelevante como funcionalidade do sistema. A *feature* que se removeu foi a `Gyro_perf_launch` por apenas apresentar um nó de teste e, por isso, pode ser eliminada. Com esta remoção, tanto o número de *features* como o de produtos finais fica mais reduzido.

Outra *feature* que se conseguiu eliminar foi a `Auto_dock_with_safe_keyop_launch`. Isto foi possível pois com a análise ao conteúdo do *launch file* concluiu-se que esta *feature* é igual a duas outras juntas (`Kobuki_auto_docking_minimal_launch` e `Safe_keyop_launch`). Como a *feature* pode ser substituída por essas duas, decidiu-se retirar pois apresentam as mesmas dependências e incompatibilidades e o mesmo se passa com os nós que ativam.

No interior de alguns *launch files* foram encontrados comentários no código que despertaram atenção para algo que levou a melhorar o modelo resultante. O tipo de comentário encontrado foi o que está representado na Listagem 6.1.

```
1 <!--
2  Requires kobuki_node/minimal.launch to be started before.
3 -->
```

Listagem 6.1: Comentários encontrados que ajudaram na modelação.

Como se pode ver, é evidente a necessidade de um ficheiro em particular para que os ficheiros sejam realmente funcionais. Essa necessidade deve-se ao facto de ser preciso um `nodelet manager` para que o sistema funcione corretamente. O `kobuki_node/minimal.launch` activa um desses `managers` e, sendo o *launcher* do Kobuki por defeito, é esse o referido, nos comentários, para ser lançado previamente.

Este tipo de comentários levou a uma análise mais profunda, onde se percebeu que, quando os argumentos de um `nodelet` contêm um `nodelet manager` com uma designação que não está presente nesse ficheiro, então vai ser necessário lançar outro que contenha essa designação. Na Listagem 6.2 está parte do *launch file* `kobuki_auto_docking/minimal.launch` que ilustra esta situação.

```
1 <launch>
2  <node pkg="nodelet" type="nodelet" name="dock_drive" args="load kobuki_auto_docking/
       ↪ AutoDockingNodelet mobile_base_nodelet_manager">
3   [...]
4  </node>
5 </launch>
```

Listagem 6.2: Exemplo de um *launch file* com necessidade de lançamento prévio de um `nodelet manager`.

Apesar de, nos comentários, estar a sugestão da necessidade do *launch file* `kobuki_node/minimal.launch` existem outros que activam o `nodelet manager` que tem que ser lançado previamente. Neste caso, dentro da diretoria `kobuki_node`, existe um outro *launch file* - `robot_with_tf.launch` - que é bastante parecido ao `minimal.launch` e que também ativa o `nodelet manager`. Com isto, percebe-se que é possível o lançamento de um ficheiro ou do outro, pois qualquer um dos dois ficheiros apresenta o `nodelet` necessário para obter uma configuração válida do robô. Na Figura 6.1 podem-se ver as restrições do modelo resultantes da análise dos argumentos de cada `nodelet`, sendo visível quais os ficheiros (agora convertidos em *features*) onde se encontrou esse tipo de necessidade.

Kobuki_auto_docking_minimal_launch ⇒ Kobuki_node_minimal_launch v Robot_with_tf_launch

Bump_blink_app_launch ⇒ Kobuki_node_minimal_launch v Robot_with_tf_launch

Keyop_launch ⇒ Kobuki_node_minimal_launch v Robot_with_tf_launch

Safe_keyop_launch ⇒ Kobuki_node_minimal_launch v Robot_with_tf_launch

Random_walker_app_launch ⇒ Kobuki_node_minimal_launch v Robot_with_tf_launch

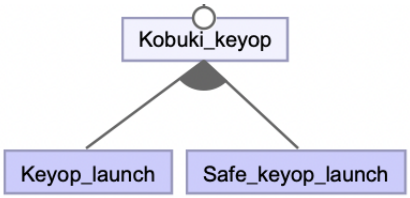
Safe_random_walker_app_launch ⇒ Kobuki_node_minimal_launch v Robot_with_tf_launch

Figura 6.1: Restrições resultantes da conversão dos comentários dos *launch files*.

Na figura podem-se ver várias *features* que necessitam de um `nodelet manager` e que por isso estão presentes nestas restrições.

Para além dos comentários não foram encontradas mais informações na documentação que ajudem na simplificação do modelo, o que leva a que seja realizado um refinamento mais conservador, de maneira a não interferir com os potenciais produtos finais. Desta forma, apenas é possível alterar o modelo para que este fique mais compreensível. Para isso identificaram-se situações em que era possível utilizar métodos de *refactoring*, sendo de seguida aplicados esses métodos.

Analisando o modelo, percebe-se que existem relações que podem ser simplificadas. Na Figura 6.2 pode-se verificar que as *features* `Keyop_launch` e `Safe_keyop_launch` estão ligadas por uma relação `or`, o que significaria que poderiam ser selecionadas as duas ou apenas uma delas, mas tal não é possível. A escolha das duas *features* em simultâneo é travada por uma restrição existente no modelo que mostra que a existência de uma implica a inexistência da outra. Ora, esta restrição dita que apenas uma delas pode ser escolhida, ou seja, poderiam ser ligadas por uma relação `xor`.



Keyop_launch ⇒ ¬Safe_keyop_launch

Figura 6.2: Relação `or` entre duas *features* e restrição que impede a coexistência.

Para tornar o modelo mais limpo, pode-se alterar a relação entre as duas *features* para uma relação `xor` e eliminar a restrição pois esta acaba por se tornar redundante. Estas alterações resultam numa simplificação do modelo, tal como mostra a Figura 6.3. Desta forma o modelo torna-se mais agradável para quem o está a analisar, levando a uma melhor compreensão do mesmo.

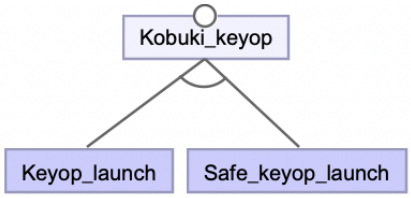


Figura 6.3: Relação `xor` entre as duas *features*.

Esta segunda relação torna-se mais percetível, pois rápido se percebe que as duas *features* são incompatíveis, apenas uma delas pode ser escolhida. No caso da relação anterior, primeiramente pensa-se que as duas podem ser utilizadas em conjunto, e só depois de analisar as restrições é que se torna visível que isso não é possível. Esta simplificação poupa tempo a quem está a analisar.

Uma análise completa do modelo revelou mais duas ocorrências deste tipo de relação travada por restrições, portanto procedeu-se à alteração do modelo tal como visto na situação anterior. As *features* envolvidas nestes dois casos foram:

- `Kobuki_random_walker`:
  - `Random_walker_app_launch`;
  - `Safe_random_walker_app_launch`;
  - `Kobuki_random_walker _standalone_launch`.
- `Kobuki_controller_ tutorial`:
  - `Bump_blink_app_launch`;
  - `Standalone_launch`.

Este tipo de situações acontece pois a ferramenta agrupa os ficheiros que pertencem à mesma diretoria em relações `or`, mas por vezes esses ficheiros são incompatíveis entre si, o que leva à adição das restrições, resultando, na verdade, em relações `xor` entre as *features*.

Para além destas alterações, foram ainda realizadas outras mudanças estéticas no modelo. A ferramenta de construção automática desenvolvida ainda não realiza este tipo de simplificação e embelezamento do modelo, mas pode ser algo a implementar no futuro. O que se alterou para que a leitura do modelo ficasse mais acessível foi o número de restrições, pois existem restrições que por vezes se podem juntar. Isto é possível pois a *feature* envolvida do lado esquerdo da restrição é a mesma, o que permite a junção.

Na Figura 6.4 está exemplificado um desses casos onde várias restrições podem ser convertidas numa só, desde que a *feature* existente do lado esquerdo da restrição seja a mesma. Na figura 6.5 está o resultado da simplificação que, como se pode ver, contém todas as *features* envolvidas e apenas utiliza uma linha no modelo, tonando-o mais claro. A passagem de uma figura para a outra é feita com a conjunção (representada no modelo pelo símbolo $\wedge$), permitindo a junção das várias restrições, numa só.

Robot_with_tf_launch ⇒ ¬Kobuki_node_minimal_launch
Robot_with_tf_launch ⇒ ¬Compact_launch
Robot_with_tf_launch ⇒ ¬Kobuki_bringup_launch
Robot_with_tf_launch ⇒ ¬View_model_launch

Figura 6.4: Restrições antes do refinamento.

Robot_with_tf_launch ⇒ ¬Kobuki_node_minimal_launch ∧ ¬Compact_launch ∧ ¬Kobuki_bringup_launch ∧ ¬View_model_launch

Figura 6.5: Junção das restrições anteriores numa só.

O tipo de alterações representadas acima, tanto a de mudança de relação como a redução do número de restrições são apenas estéticas, pois não reduzem nem aumentam o número de produtos finais ou *features*. De modo a tentar reduzir o número de produtos finais que poderiam estar menos corretos, apenas foi possível a remoção de duas *features* como foi explicado anteriormente e a introdução de novas restrições. A falta de documentação neste sistema fez com que não fosse possível uma maior redução dos produtos finais que podem não ser completamente válidos.

6.1.2 TurtleBot

Para este sistema foi alcançado um modelo bastante grande e que requer bastante atenção por parte de quem pretende analisá-lo. Felizmente, é possível fazer algumas alterações para facilitar a sua compreensão. Algumas alterações, para além de ajudarem na compreensão, ajudam também a reduzir o número de *features* (que na prática podem ser consideradas repetidas).

Como se viu anteriormente, no Capítulo 5, este é um sistema que apresenta argumentos que se tornaram *features*, no entanto, existem *launch files* que apresentam os mesmos argumentos e que a ferramenta considera diferentes (por questões de simplicidade da estratégia implementada). Contudo, isso faz com que o modelo origine repetições de produtos finais. Assim procedeu-se a uma alteração nas *features* que resultaram de argumentos, ficando desta forma o modelo mais pequeno e mais compreensível. O procedimento utilizado para essas alterações foi o seguinte:

- Criação de uma *feature* chamada `arguments` no nível abaixo da raiz;
- Identificação das *features* que representam argumentos e quais delas representam argumentos iguais;

- Colocar as *features* identificadas anteriormente, assim como as suas opções de substituição de valores, abaixo da *feature* `arguments`;
- Criação de restrições que traduzam a relação entre esses argumentos e os *launch files* a que pertencem.

Na Figura 6.6 está um excerto do modelo antes das alterações, que, como se pode ver, acaba por ser bastante confuso e onde existem dois conjuntos de *features* que na prática são duplicadas.

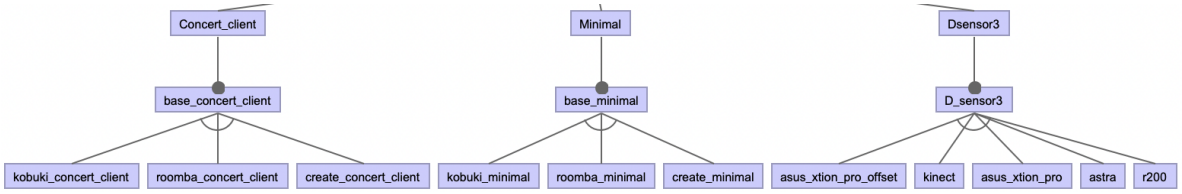


Figura 6.6: *Feature model* com repetição de *features*.

Neste exemplo conseguem-se identificar *features* que representam apenas um argumento, como é o caso da `base_concert_client` e `base_minimal`, incluíndo também as *features* filho. Depois disso é importante perceber quais as restrições que se deve colocar para que o modelo fique correto. Assim, percebeu-se que cada argumento deve implicar a existência dos ficheiros que os invocam numa relação `or` e que todos os ficheiros que os invocam devem implicar a existência deles. Esta estratégia fica mais percetível com a alteração do exemplo anterior, que está presente na Figura 6.7.

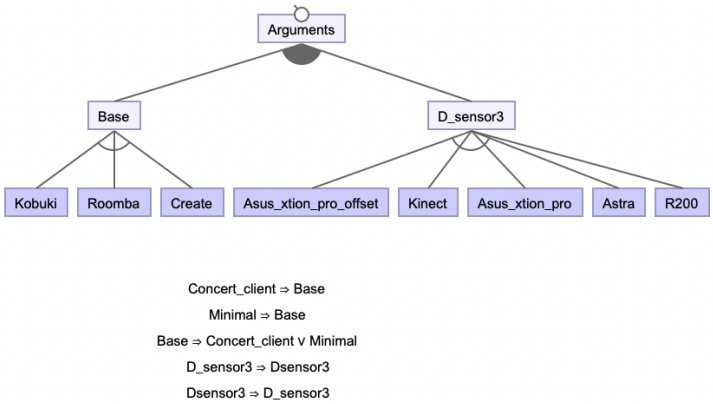
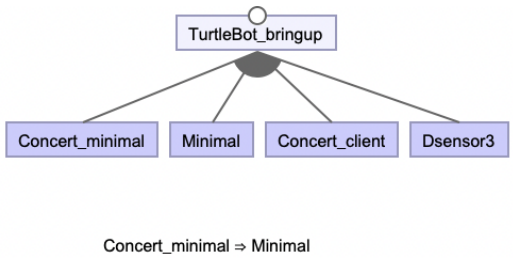


Figura 6.7: *Feature model* sem repetição de *features*.

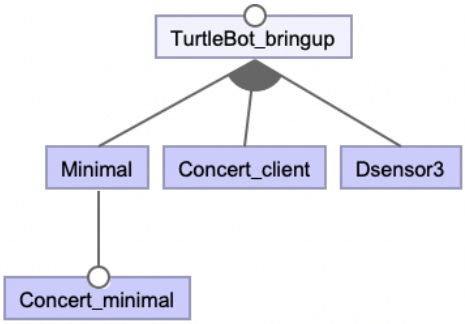
Pode-se ver que em relação à figura anterior, esta é muito mais clara e simples de analisar, mas a maior vantagem desta nova organização é a inexistência de *features* repetidas. É possível ver que cada argumento só pode ser escolhido na companhia de pelo menos um dos ficheiros que o invoca. Assim, quando um desses ficheiros é escolhido tem que se escolher também os argumentos que este invoca, tal como explicado anteriormente. Esta alteração no modelo permitiu reduzir o número de produtos finais válidos, visto que antes existiam *features* iguais, apenas com nomes diferentes, o que resultava, por vezes, em produtos repetidos.

Para além dessa alteração nos argumentos, fez-se também uma alteração na localização de uma *feature* e a respetiva eliminação de uma restrição que se iria tornar redundante com essa mudança. Na Figura 6.8a é apresentado um fragmento do modelo antes dessa mudança. Enquanto que o resultado dessa alteração está representado na Figura 6.8b.

Na primeira figura pode-se ver que a *feature* `Concert_minimal` implica a `Minimal`, mas como estão as duas ao mesmo nível e pertencem à mesma *feature* pai, pode-se considerar a `Concert_minimal` como *feature* filho da `Minimal`. Assim, quando a *feature* filho é escolhida, a *feature* pai tem que estar escolhida antes. Na segunda figura já se pode verificar a eliminação da restrição e a mudança de lugar da *feature* `Concert_minimal`.



(a) Parte do *feature model* antes da alteração.



(b) Parte do *feature model* depois da alteração.

Figura 6.8: Alteração de lugar de uma *feature* e consequente eliminação de uma restrição.

Esta foi uma alteração que apenas modificou o aspeto do *feature model*, mas que em nada alterou os produtos finais. Tal como no sistema anterior, fez-se também uma alteração nas restrições do modelo de maneira a reduzir o número destas e a tornar a sua leitura mais acessível. Essa alteração passa pela junção de restrições sempre que possível.

As únicas alterações, neste sistema, que alteraram o número de produtos finais foram aquelas relacionadas com os argumentos, todas as restantes apenas serviram para tornar o modelo mais legível. Algo que dificultou este trabalho de refinamento foi a falta de documentação, que parece ser algo comum neste tipo de sistemas. Sem documentação, as alterações que se fazem são mais conservadoras por receio de inviabilizar produtos que poderiam ser válidos, daí as reduzidas alterações feitas.

6.1.3 TurtleBot3

Este sistema foi o que resultou no maior modelo, pois é o que apresenta mais *features*. É um sistema bastante parecido, na sua programação, ao sistema anterior e por isso, algumas das alterações realizadas no TurtleBot foram também feitas neste. Um dos problemas encontrados no modelo obtido automaticamente é a existência de restrições redundantes, o que pode tornar este sistema mais confuso. Na Figura 6.9 pode-se ver quais as restrições envolvidas nas redundâncias. Este tipo de problemas aparece porque a ferramenta desenvolvida apenas deteta complicações que tornem o modelo incorreto.

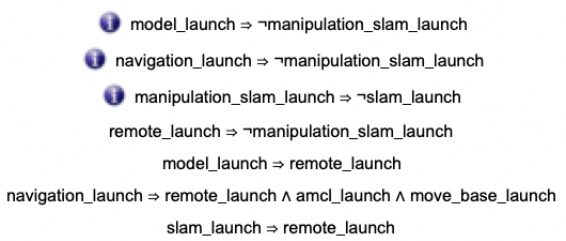


Figura 6.9: Restrições redundantes.

Como se pode verificar, as três primeiras restrições podem ser eliminadas pois são redundantes em relação ao resto das restrições. O que acontece é o seguinte:

- A 4º e 5º restrições tornam a 1º redundante;
- A 4º e 6º restrições tornam a 2º redundante;
- A 4º e 7º restrições tornam a 3º redundante.

Assim, foram eliminadas as primeiras três restrições de maneira a obter um conjunto de restrições mais reduzido e sem informação repetida.

Tal como no sistema anterior, encontrou-se uma forma de tentar reduzir o número de produtos finais que poderiam não ser totalmente corretos, ou até mesmo, repetidos. Assim, criou-se uma *feature* com o nome `arguments` onde foram colocadas as *features* que representam argumentos dos ficheiros. Depois disso criaram-se as respetivas restrições de forma a permitir apenas a existência dos argumentos quando os ficheiros correspondentes são escolhidos. Nas Figuras 6.10 e 6.11 estão o antes e depois das alterações, respetivamente.

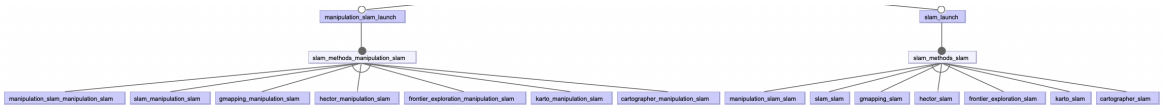


Figura 6.10: *Feature model* com *features* repetidas.

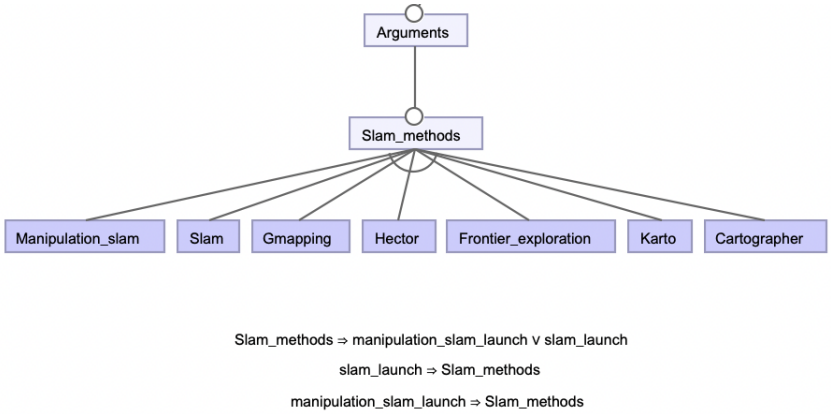
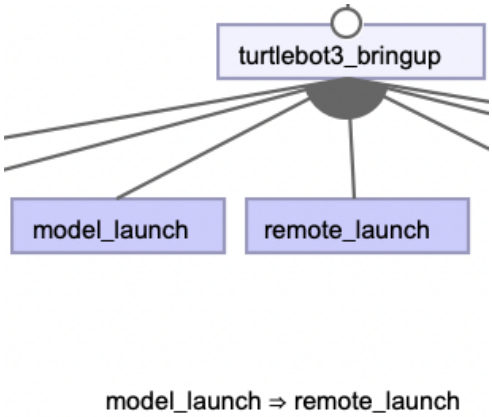


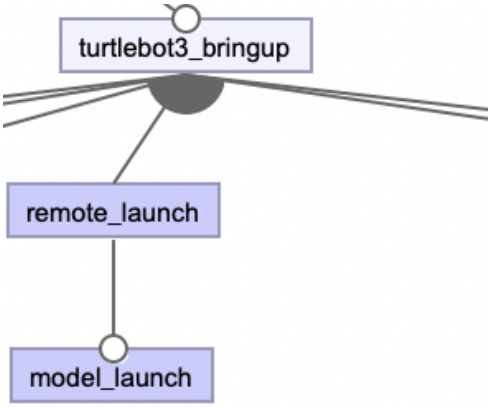
Figura 6.11: *Feature model* com as *features* repetidas eliminadas.

Como é possível observar, a mudança visual do modelo também é bastante grande: onde antes existiam duas *features* equivalentes agora existe apenas uma. Desta maneira foi possível melhorar a qualidade dos produtos finais, sem a presença de produtos repetidos e por vezes não possíveis. Com a configuração anterior era possível criar um produto onde se escolhiam os dois ficheiros que invocam os argumentos com argumentos diferentes e, na prática, isso não é possível.

Encontrou-se ainda outra oportunidade de mudança no modelo, tal como no anterior, que foi a existência de uma restrição que pode ser eliminada, colocando uma *feature* como pai de outra. Neste caso, as *features* envolvidas são `remote_launch` e `model_launch`. Nas Figuras 6.12a e 6.12b é possível perceber a transformação efetuada para a eliminação da restrição.



(a) Parte do modelo antes da alteração.



(b) Resultado de parte do modelo depois das alterações.

Figura 6.12: Processo de eliminação de uma restrição e mudança de lugar de uma *feature*.

Depois desta alteração, apenas foi possível realizar mais uma, aquela que foi possível em todos os sistemas até agora, o embelezamento e simplificação das restrições. Tal como o Kobuki e o TurtleBot, o TurtleBot3 também apresenta restrições que podem ser condensadas de maneira a facilitar a sua leitura.

De todas as alterações efetuadas neste sistema, a única que ajuda a reduzir o número de produtos finais é a que agrupa as *features* que representam argumentos. As restantes alterações apenas apresentam benefícios de embelezamento e simplicidade de leitura do modelo. Algo que ajudaria a realizar este refinamento manual seria a documentação, que é escassa, também neste sistema, tal como aconteceu nos anteriores. Assim sendo, fizeram-se as alterações que se acharam mais evidentes e necessárias, tentando sempre não comprometer os produtos finais.

6.1.4 Husky

Este é um sistema de grandes dimensões que, à primeira vista parece ser do mesmo tipo que todos os anteriores. No entanto, pelos resultados obtidos automaticamente, conclui-se que este é um sistema que apresenta uma programação parecida com o Lizi. Utiliza argumentos da mesma forma, apesar de ser em número muito mais reduzido. Olhando apenas para o modelo do Husky, parece que não é possível realizar nenhuma alteração que ajude a simplificar o modelo nem a reduzir o número de produtos finais.

No entanto, analisando com atenção o código presente nos *launch files* deste sistema é possível perceber algumas alterações possíveis. Nas Figuras 6.13a e 6.13b estão dois exemplos de *launch files* que apenas servem para incluir outros, ou seja, uma configuração com esse *launch file* convertido em *feature* nada mais é do que a junção de outras.

```
<launch>

  <!--- Run gmapping -->
  <include file="$(find husky_navigation)/launch/gmapping.launch" />

  <!--- Run Move Base -->
  <include file="$(find husky_navigation)/launch/move_base.launch" />

  <!-- Run Frontier Exploration -->
  <!-- <include file="$(find husky_navigation)/launch/exploration.launch"

</launch>
```

(a) *Launch file* `exploration_demo`.

```
<launch>

  <!--- Run gmapping -->
  <include file="$(find husky_navigation)/launch/gmapping.launch" />

  <!--- Run Move Base -->
  <include file="$(find husky_navigation)/launch/move_base.launch" />

</launch>
```

(b) *Launch file* `gmapping_demo`.

Figura 6.13: *Launch files* que apenas incluem outros.

Tal como estes, existe mais um ficheiro do mesmo tipo, por isso, no total, foram retiradas três *features* do modelo. Desta forma é possível reduzir tanto o número de *features* como o número de produtos finais.

Entrando agora na parte em que é realizado *refactoring* de modo a, apenas embelezar o modelo, encontraram-se duas situações onde pode ser realizado esse refinamento. As duas situações já foram vistas em sistemas anteriores, uma delas está representada na Figura 6.14.

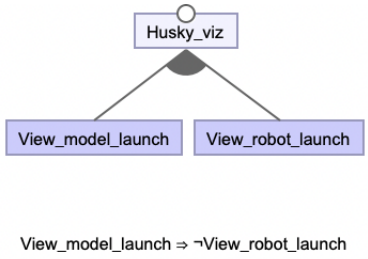


Figura 6.14: Excerto do modelo antes da transformação (relação `or` com restrição).

Como já se viu anteriormente, quando existe algo semelhante ao representado na figura anterior, pode-se alterar a relação entre as duas *features* e eliminar a restrição que impede a existência de uma quando a outra é escolhida. O resultado da transformação está representado na Figura 6.15, onde a visualização do modelo se torna mais limpa e o modelo mais claro.

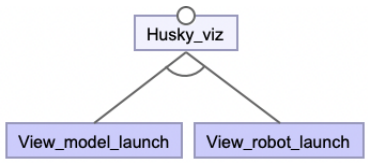
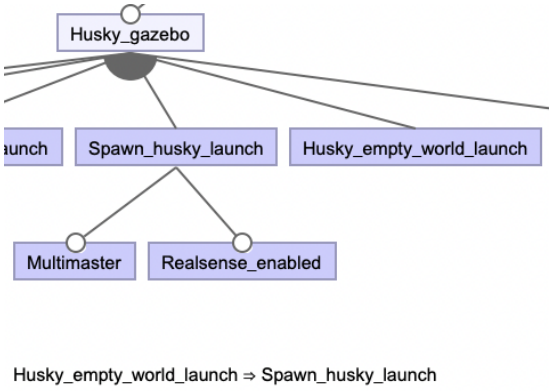


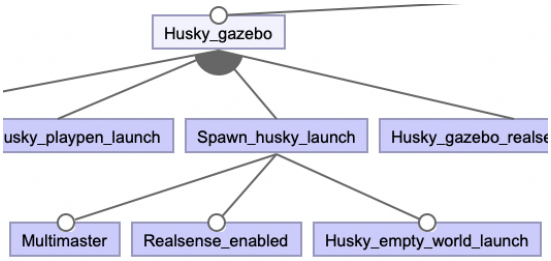
Figura 6.15: Excerto do modelo depois da transformação (relação `xor`).

Outra das situações que se identificou foi a existência de uma restrição que pode ser eliminada com a mudança de lugar de uma *feature*, isto só é possível porque a hierarquia existente não é quebrada, tal

como foi visto anteriormente nos sistemas TurtleBot e TurtleBot3. Assim, nas Figuras 6.16a e 6.16b estão o antes e o depois, respetivamente, da eliminação da restrição e mudança de lugar da *feature* em questão. Essa *feature* é a `Husky_empty_world_launch` que implica a existência da `Spawn_husky_launch` e que, por isso, pode passar a filha dessa.



(a) Hierarquia com a restrição, antes da mudança de lugar.



(b) Resultado das alterações no modelo.

Figura 6.16: Transformação do modelo através da alteração de lugar de uma *feature* e eliminação de uma restrição.

Para além destas alterações, fez-se também aquela que já é usual, a junção de restrições sempre que possível. Este é um sistema que não permitiu muitas alterações, pois tal como os sistemas anteriores não apresenta documentação específica neste sentido. A documentação que existe é, principalmente sobre o *hardware* dos sistemas, sendo que o *software* fica, em algumas situações, confuso de entender.

6.1.5 Lizi

De todos os sistemas analisados, este é o que apresenta uma documentação que ajuda a perceber o sistema, podendo esta ser utilizada para melhorar a precisão do modelo. Assim, encontrou-se informação relevante que foi convertida em restrições e desta maneira foi possível reduzir o número de produtos finais válidos. Na Figura 6.17 estão as frases da documentação que se aproveitaram e converteram em restrições.

`gmapping` - launch gmapping SLAM algorithm. Must be executed with move_base and lidar

`hector_slam` - launch Hector SLAM algorithm. Must be executed with move_base and lidar

`amcl` - launch AMCL algorithm for localization. Must be executed with move_base, lidar and map

Figura 6.17: Documentação do Lizi utilizada para o modelo.

Com estas expressões pode-se concluir que existem três argumentos que devem ser executados com outros, o que pode ser convertido, no modelo, em implicações. O resultado dessa conversão a restrições

encontra-se na Figura 6.18. Como se pode observar teve que se adaptar cada expressão ao modelo, sendo que a inexistência de algumas *features* leva à omissão de certas informações da documentação.

$$\text{gmapping} \Rightarrow \text{move\_base} \wedge \text{lidar}$$
$$\text{hector\_slam} \Rightarrow \text{move\_base} \wedge \text{lidar}$$
$$\text{amcl} \Rightarrow \text{move\_base} \wedge \text{lidar}$$

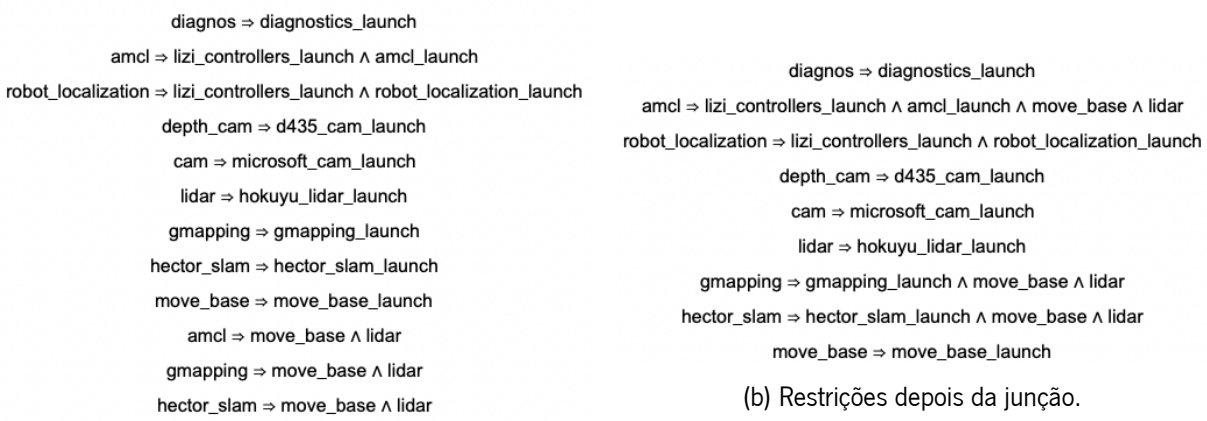
Figura 6.18: Restrições convertidas da documentação do Lizi.

Estas alterações foram importantes para criar relação entre os diferentes argumentos, gerando restrições que eliminam alguns produtos finais que antes iriam estar errados. Tal como essa, fez-se outra que também mexeu com os produtos finais, que foi a eliminação dos ficheiros de teste. Para isso fez-se uma análise dos *launch files* que estavam representados no modelo como *features*. Com isto foram encontrados dois ficheiros de teste (`gazebo_test.launch` e `joints_test.launch`), presentes na diretoria `lizi_description`, onde apenas existiam esses dois ficheiros. Assim que foram eliminados esses dois ficheiros não fez sentido a existência da diretoria vazia, pelo que foi também eliminada, pois não representa nenhuma *feature*, assim como qualquer restrição onde estas pertencessem.

Para além destas alterações feitas ao modelo, fizeram-se outras que se mostraram importantes para uma melhor compreensão do mesmo. Enquanto que as alterações anteriores alteraram o número de produtos válidos, estas novas apenas ajudam na leitura e compreensão do modelo.

Este sistema é, de facto, o mais diferente em termos de programação, tornando fácil para a ferramenta automática criar um modelo mais correto. Nos outros sistemas foi possível, a partir da mudança de lugar de *features*, a eliminação de algumas restrições, o que não é o caso deste sistema, pois já apresenta todas essas melhorias automaticamente. Os *launch files* não incluem outros, existe apenas um *launch file* que dependendo dos argumentos passados, vai incluindo os outros de maneira a depender da escolha do utilizador.

A única alteração estética que se fez neste sistema foi a junção de restrições quando a *feature* do lado esquerdo era a mesma e sempre que as relações eram compatíveis. Com isto, nas Figuras 6.19a e 6.19b estão as restrições antes e depois dessa junção.



(a) Restrições antes da junção.

(b) Restrições depois da junção.

Figura 6.19: Transformação das restrições com a junção.

Com esta junção de restrições é notório que o número de restrições diminuiu, o que torna o modelo mais limpo e mais compreensível. De notar que as restrições que levaram à necessidade desta junção foram as adicionadas manualmente, cuja informação se retirou da documentação.

Neste sistema foi possível utilizar a documentação para melhorar o modelo, algo que não foi possível nos sistemas anteriores. Assim foi possível criar um modelo mais exato e correto, que traduz melhor o sistema.

6.2 Análise e Avaliação dos Resultados

Esta secção vai servir para realizar uma análise e avaliação dos resultados obtidos automaticamente e depois do refinamento manual descrito na secção anterior. Os elementos dos modelos que vão ser utilizados para a análise dos resultados serão:

- Número de *features*;
- Numero de restrições;
- Número de produtos finais.

Com estes elementos será possível perceber quão grande é a variabilidade destes sistemas, assim como avaliar a precisão da ferramenta de extração automática. Assim sendo, a seguir apresentam-se três tabelas que representam o antes e depois de cada um dos elementos de análise. Na Tabela 6.1 encontra-se o número de *features* antes e depois do refinamento manual, onde na coluna "Alteração" o cálculo realizado foi: $Alteracão = Depois - Antes$.

Features			
Sistema	**Antes**	**Depois**	**Alteração**
Kobuki	28	26	-2
TurtleBot	32	29	-3
TurtleBot3	47	39	-8
Husky	34	31	-3
Lizi	27	24	-3

Tabela 6.1: Número de *features* antes e depois do refinamento manual.

Como se pode observar na tabela anterior, em todos os sistemas, depois do refinamento o número de *features* reduziu, o que, mais tarde, se vai traduzir numa redução no número de produtos finais. O número de restrições também reduziu na maior parte dos sistemas, esses números estão representados na Tabela 6.2, sendo que a coluna "Alteração" foi calculada da mesma forma que na tabela anterior.

Restrições			
Sistema	**Antes**	**Depois**	**Alteração**
Kobuki	31	11	-20
TurtleBot	6	9	+3
TurtleBot3	14	10	-4
Husky	16	10	-6
Lizi	10	9	-1

Tabela 6.2: Número de restrições antes e depois do refinamento manual.

A maioria dos sistemas apresentaram uma redução de restrições, no entanto, o Kobuki foi o sistema que manifestou uma maior e significativa redução. Esta redução deve-se principalmente ao facto de, durante o refinamento manual, ter-se juntado as restrições de acordo com algumas regras vistas na secção anterior. Como o resultado automático devolveu um modelo com um número muito elevado de restrições, foi possível fazer essa junção e assim reduzir o número destas. Contrariamente à maior parte dos sistemas, o TurtleBot mostra um aumento no número de restrições, que pode ser explicado pela alteração nas *features* que representam argumentos do sistema. Quando se alterou essas *features* foram acrescentadas algumas restrições, situação que, combinada com o já reduzido número de restrições do modelo automático, resultaram num aumento desse número.

A tabela que talvez expressa melhor a diferença entre os resultados completamente automáticos e com o refinamento manual é a que exibe o número de produtos finais válidos de cada um desses modelos. A Tabela 6.3 apresenta uma coluna "Redução", que exprime qual a percentagem de redução do número de produtos finais. Os valores desta coluna foram calculados da seguinte forma: $Reducão(\%) = \frac{Antes - Depois}{Antes} \times 100.$

Produtos Finais			
Sistema	**Antes**	**Depois**	**Redução**
Kobuki	45 568	7 776	82,93%
TurtleBot	59 136	33 792	42,85%
TurtleBot3	n/a	n/a	n/a
Husky	n/a	n/a	n/a
Lizi	121 916	25 646	78,96%

Tabela 6.3: Número de produtos válidos finais antes e depois do refinamento manual.

Nesta tabela é possível perceber o quanto se conseguiu reduzir os produtos finais com o refinamento manual. É aqui que também se verifica a quantidade de variabilidade que estes sistemas permitem. Em alguns deles, o número de produtos finais válidos são tantos que a implementação Java do TVL não foi capaz de contá-los e deu como resultado um *time-out*. Pode-se concluir pela tabela que, nos sistemas testados, houve uma grande redução dos produtos finais depois do refinamento manual, o que se torna positivo. No entanto, não se pode concluir que estes valores e os modelos estejam totalmente corretos, pois o ideal seria contactar os programadores destes sistemas e perceber o que poderia ser melhorado. Os resultados automáticos também não são totalmente corretos, pois existem alterações que apenas é possível fazer manualmente visto que requerem uma análise da documentação dos sistemas.

A grande redução de produtos finais deveu-se principalmente à eliminação de algumas *features* e à adição de restrições durante o refinamento. Todas essas decisões foram devidamente explicadas sendo que nunca se retirou nem adicionou algo sem que se tivesse a certeza de que não iria inviabilizar um produto potencialmente válido.

Com estes resultados conclui-se ainda que, de acordo com os *launch files* existentes nos repositórios dos sistemas, as *features* que foram retiradas durante a fase de refinamento são principalmente de teste ou que podiam ser substituídas pela junção de outras. Assim sendo, nunca foi retirada uma *feature* chave dos sistemas, o que faz com que, de acordo com a informação dos sistemas, não existem produtos finais fora dos resultantes, tanto antes como depois do refinamento.

Capítulo 7

Conclusão

Este capítulo resume o trabalho realizado durante esta tese e responde à pergunta que incentivou a pesquisa e, consequentemente, levou à realização do presente documento. Também serão mencionados os principais contributos e referido o que se pretende realizar como trabalho futuro.

7.1 Considerações Finais

Nesta tese está presente toda a pesquisa realizada para que fosse possível perceber como é gerida a variabilidade em sistemas ROS, assim como o desenvolvimento de uma ferramenta capaz de construir *feature models* de sistemas ROS apenas a partir do seu código. Tal como a pesquisa, também está aqui referido todo o processo de desenvolvimento dessa mesma ferramenta, assim como os resultados obtidos e a sua análise.

Durante toda a pesquisa efetuada antes da análise, ficou-se a conhecer o ROS e todos os seus componentes, onde se identificaram como principais os nós, tópicos, mensagens, serviços e parâmetros. Algo que se verificou que era bastante importante na criação de sistemas ROS foram os *launch files*, pois permitem a utilização dos componentes e é a partir deles que se consegue lançar os sistemas.

Para ajudar na resposta ao problema principal que originou esta tese, foi efetuada uma investigação em *Software Product Lines*, o que levou a vários outros temas relevantes. Um tópico muito importante que se abordou foi a variabilidade na robótica, que permite a adaptação do robô ao ambiente em que opera. Durante a exposição desse tema identificaram-se os principais fatores da origem da variabilidade tais como: requisitos do cliente, meio ambiente, componentes do robô e *middleware*.

Interpretar a variabilidade de um sistema não é um trabalho simples, por isso foram vistas algumas técnicas de modelação da variabilidade, sendo que os *feature models* foram o método escolhido para utilizar neste projeto. Para além disso referiu-se uma linguagem TVL e biblioteca associada, capaz de contar e identificar os produtos válidos de um *feature model*. O facto de existir muita variabilidade nalguns

sistemas leva a que seja necessário encontrar técnicas específicas para programar os mesmos, existindo duas abordagens comummente utilizadas: composicionais e anotativas. Cada uma apresenta as suas vantagens e desvantagens, cabendo portanto aos programadores escolher a abordagem que acham que mais se adequa ao sistema que pretendem programar.

Para entender melhor os sistemas ROS e para obter informações relevantes sobre esses sistemas foi realizada uma análise profunda em cinco sistemas ROS: Kobuki, TurtleBot, TurtleBot3, Husky e Lizi. Em todos estes sistemas foi elaborada uma análise do seu código e identificada a abordagem utilizada para a programação da variabilidade, chegando-se à conclusão que todos eles utilizam uma abordagem composicional, à exceção do Lizi que utiliza uma abordagem anotativa. Para além disso foram encontrados, para cada sistema, todos os *launch files* incompatíveis (ficheiros diferentes que ativavam nós com o mesmo nome), assim como eventuais ajudas no entendimento dos sistemas presentes na documentação dos mesmos. No final desta análise percebeu-se que os fatores que ajudam na gestão da variabilidade são principalmente os *launch files*, assim como os nós e os argumentos, respondendo assim à principal pergunta levantada nesta tese.

Adicionalmente desenvolveu-se uma ferramenta capaz de analisar sistemas ROS e construir *feature models*, de maneira a ajudar a quantificar a variabilidade de um sistema. Esta ferramenta analisa os *launch files* dos sistemas e a partir dessa análise vai criando um *feature model* na linguagem TVL. A construção da ferramenta foi desenvolvida por etapas, sendo que cada uma delas foi exposta, assim como o respectivo resultado para, pelo menos, um sistema.

Depois de todo o desenvolvimento da ferramenta e da sua utilização em todos os sistemas analisados, foram realizadas algumas alterações nos resultados obtidos. Essas alterações serviram principalmente para refinar o modelo obtido, de forma a obter resultados mais corretos. Para isso foi analisado todo o código dos sistemas e perceberam-se algumas peculiaridades que ajudaram nesse refinamento. Outras alterações focaram-se no embelezamento do *feature model* para que este não se tornasse tão confuso. A falta de documentação nestes sistemas foi algo que não ajudou neste processo, pelo que se fizeram poucas alterações de forma a não correr o risco de criar um modelo incorreto. Depois de tudo isto, fez-se uma avaliação dos resultados obtidos antes e depois do refinamento manual. Percebeu-se principalmente que a técnica automática ainda está longe de ter uma boa precisão. No que diz respeito ao refinamento manual, considerando toda a falta de documentação destes sistemas, obtiveram-se resultados que se acredita não estar assim tão longe do que é correto. É certo que há sistemas que apresentam melhores resultados do que outros, sendo que isso também se deve à forma como foram programados os sistemas e à estratégia de extração implementada na ferramenta automática. O sistema em que se conseguiu uma maior redução no número de produtos finais no refinamento manual foi o Kobuki, com uma redução de 82,93%, seguido do Lizi com 78,96% e do TurtleBot com 42,85%.

7.2 Principais Contributos

No decorrer desta tese procurou-se tratar um assunto bastante importante e ainda desconhecido para a comunidade científica da robótica, a gestão da variabilidade em sistemas ROS. Dentro deste tema foram aprofundados dois subtemas e para os quais foi encontrada uma possível solução:

- **Fatores que gerem a variabilidade dos sistemas ROS** - a resposta a esta dúvida é algo que pode ser considerado importante para a robótica para se poder criar ferramentas que ajudem nessa gestão. Esta resposta foi dada no Capítulo 4 depois da análise de todos os sistemas;
- **Viabilidade da extração automática de *feature models*** - como foi possível observar no Capítulo 5, foi possível o desenvolvimento de uma ferramenta automática que ajuda a criar *feature models* a partir da análise de *launch files*. Acredita-se que a ferramenta desenvolvida constitui um contributo relevante para a robótica.

Outro dos contributos desta tese foi o refinamento manual realizado nos resultados obtidos automaticamente com a ferramenta. Esse refinamento contribui na medida em que ajuda a traçar objetivos de melhoramento da ferramenta para que esta se torne mais precisa.

Para além destes, durante esta tese foram realizadas inúmeras análises a cinco sistemas ROS que funcionaram como casos teste para a ferramenta desenvolvida. Com isto, aumentou-se o conhecimento sobre estes sistemas, o que facilita a sua utilização como casos de estudo em novas investigações, não sendo necessário repetir a análise já aqui efetuada.

7.3 Trabalho Futuro

Embora se acredite que este tese apresenta um bom contributo para a comunidade da robótica, podem ainda ser feitas algumas melhorias ao trabalho realizado. Como trabalho futuro pretende-se melhorar a ferramenta de modo a criar *feature models* mais precisos do que se está a obter neste momento. Alguns pontos nos quais se pretende trabalhar são os seguintes:

- Melhorar a ferramenta de extração automática de modo a que, nos sistemas anotativos, sejam considerados *feature* os argumentos que afetam o lançamento de nós, pois neste momento apenas se estão a considerar aqueles que afetam a inclusão de *launch files*;
- Completar a ferramenta com a funcionalidade de conseguir encontrar *launch files* que precisem de um `nodelet manager` específico e procurar os ficheiros que lançam esse tipo de `manager`, criando restrições que traduzam essas necessidades;
- Sempre que, em sistemas composicionais, existam argumentos, criar uma *feature* onde sejam colocados todos esses argumentos e respetivas opções de substituição, criando sempre as restrições necessárias;

- Tornar os modelos obtidos pela ferramenta menos confusos, de maneira a que não seja necessário um refinamento manual tão exaustivo, como é o caso da junção das restrições sempre que possível;
- Aplicação da ferramenta de extração automática a sistemas ROS mais complexos, de maneira a quantificar a variabilidade dos mesmos.

Todas estas melhorias à ferramenta permitirão criar um produto mais robusto e que constitui um maior contributo para a comunidade ROS. A aplicação da ferramenta a outros sistemas pode ser vantajoso para a aprofundar a sua validação, podendo perceber como é o seu desempenho quando aplicada a sistemas mais complexos.

Algo que também se pretende fazer é a escrita de um artigo científico, onde se apresenta esta pesquisa e o desenvolvimento da ferramenta de extração, em conjunto com a discussão de alguns resultados relevantes.

Bibliografia

[1] M. Acher, P. Collet, P. Lahire e R. B. France. “FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models”. Em: *Science of Computer Programming* 78.6 (2013), pp. 657–681.

[2] F. Aleixo, M. Freire, D. Alencar, E. Campos et al. “A Comparative Study of Compositional and Annotative Modelling Approaches for Software Process Lines”. Em: *2012 26th Brazilian Symposium on Software Engineering*. IEEE. 2012, pp. 51–60.

[3] F. Aleixo, M. Freire, W. dos Santos e U. Kulesza. “A Model-driven Approach to Managing and Customizing Software Process Variabilities”. Em: *ICEIS 2010 - Proceedings of the 12th International Conference on Enterprise Information Systems*. Vol. 3. 2010, pp. 92–100.

[4] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba e C. Lucena. “Refactoring Product Lines”. Em: *Proceedings of the 5th international conference on Generative programming and component engineering - GPCE ’06*. 2006, pp. 201–210.

[5] S. Apel, D. Batory, C. Kästner e G. Saake. *Feature-Oriented Software Product Lines*. Springer, 2016, pp. 3–15.

[6] S. Apel e C. Lengauer. “Superimposition: A Language-Independent Approach to Software Composition”. Em: *International conference on software composition*. Springer. 2008, pp. 20–35.

[7] D. Batory, J. N. Sarvela e A. Rauschmayer. “Scaling Step-Wise Refinement”. Em: *IEEE Transactions on Software Engineering* 30.6 (2004), pp. 355–371.

[8] D. Batory. “Feature Models, Grammars, and Propositional Formulas”. Em: *International Conference on Software Product Lines*. Springer. 2005, pp. 7–20.

[9] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki e A. Wasowski. “A Survey of Variability Modeling in Industrial Practice”. Em: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*. 2013, pp. 1–8.

[10] Q. Boucher, A. Classen, P. Faber e P. Heymans. “Introducing TVL, a Text-based Feature Modelling Language”. Em: *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS’10), Linz, Austria, January*. 2010, pp. 27–29.

[11] A. Classen, Q. Boucher, P. Faber e P. Heymans. “The TVL Specification”. Em: *PReCISE Research Center, University of Namur* (2010).

[12] J. Coplien, D. Hoffman e D. Weiss. “Commonality and Variability in Software Engineering”. Em: *IEEE software* 15.6 (1998), pp. 37–45.

[13] A. Cunha, A. Santos, N. Macedo, R. Arrais e F. N. Dos Santos. “Mining the Usage Patterns of ROS Primitives”. Em: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2017, pp. 3855–3860.

[14] K. Czarnecki e U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Jan. de 2000.

[15] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang e P. Heymans. “Feature Model Extraction from Large Collections of Informal Product Descriptions”. Em: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 2013, pp. 290–300.

[16] C. Dietrich, R. Tartler, W. Schröder-Preikschat e D. Lohmann. “A Robust Approach for Variability Extraction from the Linux Build System”. Em: *Proceedings of the 16th International Software Product Line Conference-Volume 1*. 2012, pp. 21–30.

[17] M. Galster, D. Weyns, D. Tofan, B. Michalik e P. Avgeriou. “Variability in Software Systems — A Systematic Literature Review”. Em: *IEEE Transactions on Software Engineering* 40.3 (2013), pp. 282–306.

[18] S. García, D. Strüber, D. Brugali, A. Di Fava, P. Schillinger, P. Pelliccione e T. Berger. “Variability Modeling of Service Robots: Experiences and Challenges”. Em: *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems*. 2019, pp. 1–6.

[19] T. L. Harman e C. Fairchild. *ROS Robotics by Example*. Packt Publishing Ltd, 2016.

[20] P. Haumer. “Eclipse Process Framework Composer”. Em: *Eclipse Foundation* (2007).

[21] C. Kästner e S. Apel. “Integrating Compositional and Annotative Approaches for Product Line Engineering”. Em: *Proc. GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*. 2008, pp. 35–40.

[22] C. Kästner, S. Apel e M. Kuhlemann. “Granularity in Software Product Lines”. Em: *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE. 2008, pp. 311–320.

[23] C. Kästner, T. Thüm, G. Saake, J. Siegmund, T. Leich, F. Wielgorz e S. Apel. “FeatureIDE: A Tool Framework for Feature-Oriented Software Development”. Em: *2009 IEEE 31st International Conference on Software Engineering*. IEEE. 2009, pp. 611–614.

[24] S. Nadi, T. Berger, C. Kästner e K. Czarnecki. “Mining Configuration Constraints: Static Analyses and Empirical Results”. Em: *Proceedings of the 36th international conference on software engineering*. 2014, pp. 140–151.

[25] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulesza e P. Borba. “Investigating the Safe Evolution of Software Product Lines”. Em: *Proceedings of the 10th ACM international conference on Generative programming and component engineering*. 2011, pp. 33–42.

[26] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng et al. “ROS: An Open-Source Robot Operating System”. Em: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[27] S. She, R. Lotufo, T. Berger, A. Wasowski e K. Czarnecki. “Reverse Engineering Feature Models”. Em: *Proceedings of the 33rd International Conference on Software Engineering*. 2011, pp. 461–470.

[28] J. Siegmund, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich e G. Saake. “Do Background Colors Improve Program Comprehension in the# ifdef Hell?” Em: *Empirical Software Engineering* 18.4 (2013), pp. 699–745.

[29] H. Spencer e G. Collyer. “# ifdef Considered Harmful, or Portability Experience with C News”. Em: *{USENIX} Summer 1992 Technical Conference ({USENIX} Summer 1992 Technical Conference)*. 1992.

[30] J. Van Gurp, J. Bosch e M. Svahnberg. “On the Notion of Variability in Software Product Lines”. Em: *Proceedings Working IEEE/IFIP Conference on Software Architecture*. IEEE. 2001, pp. 45–54.

[31] T. Ziadi, L. Frias, M. da Silva e M. Ziane. “Feature Identification from the Source Code of Product Variants”. Em: *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE. 2012, pp. 417–422.