# Variability Analysis for Robot Operating System Applications

André Santos*, Alcino Cunha†‡, Nuno Macedo†§, Sara Melo‡ and Ricardo Pereira‡
**Vortex-CoLab*, Vila Nova de Gaia, Portugal
†*INESC TEC*, Porto, Portugal
‡*University of Minho, Braga, Portugal*
§*University of Porto*, Porto, Portugal

***Abstract*—Robotic applications are often designed to be reusable and configurable. Sometimes, due to the different supported software and hardware components, as well as the different implemented robot capabilities, the total number of possible configurations for a single system can be extremely large. In these scenarios, understanding how different configurations coexist and which components and capabilities are compatible with each other is a significant time sink both for developers and end users alike. In this paper, we present a static analysis tool, specifically designed for robotic software developed for the Robot Operating System (ROS), that is capable of presenting a graphical and interactive overview of the system's runtime variability, with the goal of simplifying the deployment of the desired robot configuration.**

***Index Terms*—software engineering, variability, software product lines, software analysis**

## I. INTRODUCTION

Robots are evolving at an astounding pace in recent years. Developments in software controllers and hardware components led us from mostly static machines, performing repetitive tasks, to autonomous vehicles and assistants, capable of adapting to a multitude of scenarios. This shift is associated with highly modular designs, now pursued by many robot makers, that enable users to switch software and hardware components as they see fit – be it for cost reduction, or simply to enable different capabilities. As long as the different components agree on a communication interface and protocol, the same robotic components that are assembling goods in a factory today, could be helping doctors save lives tomorrow.

The use of middlewares, such as the Robot Operating System (ROS) [1], was crucial to standardize communication interfaces, as well as managing some of the complexity associated with lower-level tasks. ROS is very lenient in its design, enabling highly dynamic and reconfigurable applications based on independent components and message exchange channels. However, while key to achieve modularity, this level of flexiblity raises an array of well-known problems in software engineering, namely *variability management*.

The research leading to these results has received funding support from the projects: “STEROID - Verification and Validation of ADAS Components for Intelligent Vehicles of the Future” from the European Union Financial Support (FEDER) under grant agreement No. 69989; “NORTE-06-3559-FSE-000046 - Emprego altamente qualificado nas empresas – Contratação de Recursos Humanos Altamente Qualificados (PME ou CoLAB)” financed by the Norte's Regional Operational Programme (NORTE 2020) through the European Social Fund (ESF); and National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.

Variability essentially describes the *features* of a system, points where components, parameters, or capabilities may change due to configuration. As the number of features increases and their interaction becomes more complex, development becomes unmanageable without a unified view of the commonalities and variation points among the acceptable configurations of a system (i.e., its *variants* or *products*), leading to the concept of Software Product Line (SPL) development [2]. Here, *domain engineering* is fundamental, which includes domain *analysis* – identifying and analysing the interaction between different features, including detecting invalid configurations – domain *design* – defining a generic system architecture shared by all variants – and domain *implementation* – defining how variability is resolved to obtain executable applications.

This work focuses precisely on variability management in the ROS ecosystem, a domain where the issue is evident – robots such as the TurtleBot2, a simple mobile robot with a few different sensors, already come with hundreds of packaged configurations. We first studied how variability manifests in a ROS application and how it is often expressed. We found out that variability management is a mostly manual and error-prone process, both for developers and (perhaps more so) for users. This served as motivation for us to develop a methodology and tool where ROS applications are interpreted as SPLs.

The proposed technique shows that it is possible, albeit with some assumptions and limitations, to automatically extract variability points (features) from source code, namely *launch files* (commonly used both in ROS and ROS2), and identify feature conflicts that would lead to invalid configurations. This extracted information is displayed graphically to support the developer in domain analysis. To aid in domain design, the tool presents the system's computation graph – its runtime architecture – annotated with *presence conditions*, aiding the user in understanding which runtime entities are tied to particular features. Users can select features and see live updates on the shape of the resulting graph. Users are also alerted when a given configuration is potentially invalid, according to a limited set of rules. Lastly, the tool supports domain implementation by generating a configuration artefact (a launch command) that deploys the selected variant.

For evaluation purposes, we have studied four open-source robotic systems in depth and assessed how the tool handled source code from those real-world applications. The technique was implemented on top of the HAROS [3], [4] framework for the development of high-assurance ROS applications, providing a friendly interface for ROS developers. This is also a first step towards supporting variability-aware analyses that would not otherwise scale if applied individually to each of the (many possible) configurations.

The remainder of this paper is structured as follows. Section II provides some necessary background on software variability. Section III complements it, providing background on ROS and how variability may manifest in ROS applications according to our preliminary study. We follow with Section IV, describing our approach and tool implementation, which are evaluated in Section V. Lastly, Section VI compares our work to prior approaches in the same domain, and Section VII lays down our conclusions and directions for future work.

## II. SOFTWARE VARIABILITY

Variability is a concept that goes hand in hand with the concept of Software Product Lines (SPL) [2]. An SPL is a family of (software) products, also called *variants*, that stem from a common base. Each variant supports a different set of capabilities, components or parameters, collectively referred to as *features*. Understanding and adopting an SPL-oriented development process reduces the overall complexity of building modular designs. Variability, then, measures in what ways a system can be reconfigured, by adding, removing or replacing its features.

According to [5], there are four main sources of variability in robotics: the clients, who require different capabilities for different applications; the different environments in which a robot is expected to operate; the range of supported hardware and software components; and the supported middlewares, with each requiring its own interface code for the same component. So it is clear how a modern and modular robotic system could be interpreted as an SPL with an overwhelming number of theoretically possible products.

Feature-oriented domain analysis is one of the most common approaches to variability management [6]. Here, a *feature model* provides a hierarchical representation of all the available features in the system. It also establishes relations of dependency and conflict between features, such as the selection of one feature requiring the inclusion (or exclusion) of another. In particular, child features require always the selection of their parent feature, while siblings often represent different, but not necessarily exclusive, options for the same parent feature – e.g., different hardware components for an abstract *Sensor* feature. Feature models also admit relations between features that are not in the same hierarchical line, resulting in *cross-tree constraints*. Given a set of constraints and hierarchical relations, one can easily determine whether a given selection of features is a consistent (or valid) product.

During design and implementation, the naïve approach to variability is to *clone-and-own*, duplicating code and then changing it as necessary. This strategy has an obvious deficiency in terms of maintainability, which makes *compositional* and

```
<launch>
  <arg name="vel_topic" default="vel_cmd" />
  <arg name="use_teleop" default="false" />
  <param name="max_vel" type="double" value="0.8" />
  <include file="$(find example)/launch/diagnostics.launch" />
  <node name="controller" pkg="robot_teleop" type="kb_teleop"
      if="$(arg use_teleop)">
    <remap from="vel_cmd" to="$(arg vel_topic)" />
  </node>
  <node name="controller" pkg="robot_planner"
      type="laser_planner" unless="$(arg use_teleop)">
    <remap from="vel_cmd" to="$(arg vel_topic)" />
  </node>
</launch>
```

Fig. 1. Example ROS launch file.

*annotative* approaches better suited choices [7]. The former views features as isolated modules and then composes them to obtain the final product. The latter uses annotations in the artefacts to identify what is related to each feature. For design, this can be achieved, for instance, through stereotypes in UML models. For implementation, a common example is the use of conditional compilation – e.g., `#ifdef` preprocessor directives in C code. Each approach has its benefits and trade-offs; neither is a *de facto* choice.

While, ideally, SPL engineering would be adopted from the beginning of the development (in a, so called, proactive approach), in practice it is often only adopted later in the process as the complexity of the products increases, often without any prior domain analysis nor feature model to help manage the existing variability. This led to the development of many reverse engineering methods and tools for automatic extraction of feature models from source code [8].

## III. THE ROBOT OPERATING SYSTEM

ROS builds upon established concepts in software engineering, easing its learning curve and making it more appealing as a prototyping and development platform.

In terms of source code, software is organized in *packages*, the basic build and distribution units. Many packages are open source and freely available for reuse. Some are even collected in official indices called *distributions*. The latest distributions of ROS, for example, list more than 4 000 packages.

At runtime, a ROS system functions based on the concept of *computation graph*: a distributed network of black-box components (*nodes*) that communicate by exchanging messages (mostly) over publisher-subscriber channels (*topics*). ROS is multilingual, with its two main languages being C++ and Python. In order to assemble a system (i.e., define its computation graph), developers often use *launch files* – XML files where one is able to declare: *(i)* which nodes the ROS infrastructure should run; *(ii)* what unique names to assign to each node; *(iii)* which parameters to pass to each node instance; and *(iv)* how to redirect communication channels, to connect nodes. A single system can be composed of multiple launch files to be executed in parallel.

The example in Fig. 1 illustrates the typical features of a ROS/ROS2 XML launch file. The `arg` tag declares an argument

– essentially a local variable that can be used throughout the launch file. In the example, only `default` values are specified, which will be used if the user does not provide alternative values via command line. The **`param`** tag declares a global parameter that can be accessed at runtime by any ROS node. Launch files can **`include`** the contents of other launch files, enabling a form of composition. The **`node`** tag is self-explanatory – it instantiates a node into the system. Each node is given a unique `name` and launches an executable (`type`) from a ROS package (`pkg`). The same node `type` may be used multiple times in the computation graph, as long as each `name` is unique. The nested **`remap`** tags redirect communication channels of the respective nodes. In this example, whenever the nodes access `vel_cmd`, they will be transparently redirected to the value of the `vel_topic` **`arg`** instead. More interestingly, these nodes are conditional. The first node is only launched **`if`** the value of `use_teleop` is true. Conversely, the second node is always launched, **`unless`** `use_teleop` is true.

An empirical study [9] shows that command-line **`arg`**, runtime **`param`** and channel **`remap`** are widely used in the ROS ecosystem. With these features, developers can write modular code that is not tied to a specific application; each component can be reconfigured as necessary during system orchestration. File inclusion and conditional statements are also used, in a similar measure. Additional mechanisms, used to a lesser extent, are, e.g., environment variables (not shown in Fig. 1).

Our preliminary study of ROS systems – among which are popular robots in the community, such as the TurtleBot2[1], Husky[2] and Lizi[3] – has shown that developers tend to prefer a compositional approach to variability. For each capability there are multiple launch files with slightly different configurations, from which a selection is launched in parallel. For example, one common launch file starts the robot's base, while, for example, a second launch file, picked from a set of provided launch files, starts navigation components with the correct map for a given area. On average, only about 20% of all nodes are unique [9]. The abundant code duplication in launch files is confirmed by other studies as well [10]. Lizi is an exception; it leans towards the annotative method, having fewer but highly parameterisable entry points that use conditional file inclusion (depending on user-supplied **`arg`**) to load features.

## IV. VARIABILITY ANALYSIS FOR ROS

The attentive reader might have noticed that, in our brief introduction to how variability manifests and how it is handled in ROS applications, in Section III, we have never mentioned what, exactly, constitutes a *feature* in ROS (or an *application*, for that matter). There are no formal definitions for such concepts in the official ROS documentation, a fact that poses our first challenge when automating variability analysis. ROS works with dynamic and open-ended networks, which makes it harder to define up front the full architecture of a system. A single robot might represent a full application in itself, if acting alone, but it could also be just a component of a larger application in the context of a robot swarm, for example. As such, we assume that the intention of the developers of a robotic application is essential to identify the packages and components that compose it. We base our approach on the fact that developers almost always provide (one or more) launch files to be used as the entry points to run a ROS application. With that in mind, we can start designing what a *ROS feature model* could look like.

[1]https://wiki.ros.org/Robots/TurtleBot

[2]https://wiki.ros.org/Robots/Husky

[3]https://wiki.ros.org/lizi

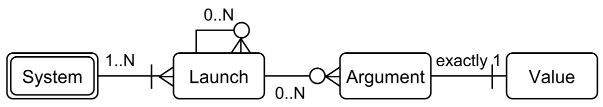


Fig. 2. Feature metamodel for a ROS application.

### A. ROS Feature Modelling

We consider a ROS *application* to be a set of nodes to be launched, along with their configuration. Nodes are major actors in the ROS computation graph, and their configuration dictates which parameters and communication channels will be available at runtime. In other words, an application is the set of necessary building blocks to compose a specific system product or computation graph.

We consider a *feature* of an application to be any configuration option that ultimately affects the structure of the computation graph[4]. As we have seen in Section III, nodes, parameters and communication channels are the core components of the ROS computation graph. Channels are determined by the set of launched nodes, without user intervention. Nodes and parameters are, in turn, determined by the selected launch files. The resulting configuration options a user has – namely, launch files and their corresponding command-line arguments – make the features of a ROS application.

Launch files are features for the obvious reason that they group together and enable all other components; they are the entry points of an application. Command-line arguments (**`arg`**), despite not being features *per se* – in the traditional sense of an observable component or functionality – can be set with a user-provided value, which is then used instead of hard-coded literals to determine the inclusion or exclusion of other features and components. For example, `$(arg use_teleop)` in Fig. 1 determines which of the two nodes will be instantiated. To the top-level set of launch files that a user wants to deploy in parallel, to instantiate a concrete application, we call a *system*. This feature hierarchy is shown in Fig. 2.

Standard feature model restrictions are supported over the presented structure. The selection of a child feature imposes the selection of its parent – selecting the arguments of a launch file entails the selection of that launch file. Features can be grouped in *xor-groups*, where exactly one child feature must be selected, e.g., exactly one of the alternative values for an argument. Cross-tree constraints of *inclusion* – when a launch

[4]Variability may also affect the internal behaviour of the nodes, but that is beyond the scope of this work.

file includes another one – and *exclusion* – when launch files conflict with each other due to launching nodes or parameters with the same name – are also supported. Section IV-D explains how all these are automatically extracted from source code.

## B. ROS Variational Architectures

Besides the feature model just described, we provide the SPL's computation graph. In previous work, Santos et al. [11] proposed an architectural model for representing computation graphs of ROS applications. Here, these are extended by allowing the annotation of runtime entities with *presence conditions*, propositional formulae over the features of the extracted feature model and the components they spawn. This allows the user to quickly identify, in a single model, which elements are mandatory or optional, and under which configurations. For instance, if a `node` is spawned by a launch file $a$, its presence condition is simply $a$. If it is launched conditionally, with `if`, when an argument $x$ has value $v$, and assuming feature $a\_x\_v$ denotes that $v$ is assigned to $x$ in $a$, its presence condition is $a\_x\_v$. If two launch files, $a$ and $b$, spawn a `node` with the same name, $n$, we have to calculate a *super-product* that merges both occurrences of $n$, disjoining its presence conditions (in this case, $a \vee b$, if launched unconditionally). Note that the feature model would actually have an exclusion constraint forbidding $a$ and $b$ to be selected simultaneously due to the node name clash. Presence conditions are simplified when elements are merged, and if they evaluate to false (resp. true), elements are forcibly excluded from (resp. added to) the final product.

All extracted entities keep a series of attributes pertaining to their configuration in the launch files. For example, a node maintains attributes such as its name, package and executable type – extracted from the `node` tag itself – but it also keeps a set of name remappings – extracted from child `remap` tags – that it should apply to redirect communication channels. These attributes, too, are kept variational, i.e., annotated with presence conditions, so that we may calculate their final value when merging selected features. For example, a `node` of `type` $t_1$ in launch file $a$ and `type` $t_2$ in file $b$ can only determine its final `type` once either $a$ or $b$ are selected.

## C. Multi-step configuration

Once the feature model and the variability-aware computation graph are created, the user may perform multi-step configuration by iteratively selecting/rejecting features. Each feature can be in one of three states: **selected**, if it should be part of the system; **deselected**, if it should be excluded; or **unselected**, if no choice has yet been made. This state is determined by the user selections, but can also be affected by the feature model constraints (e.g., if an argument value $a\_x\_v$ is selected by the user, its parent launch file $a$ is also automatically selected). In parallel, the computation graph is also refined as the feature selection evolves. For example, if a `node` has presence condition $a \vee b$, and the user selects launch file $a$, the presence condition becomes true and the `node` becomes unconditionally present in the application.

**Algorithm 1** Feature model extraction

```
Input: a set of launch files $lfiles$
Output: a feature model $fm$
  $lns \leftarrow \{\}$
  $fm \leftarrow$ FM("$root$")
  for all $xml \in lfiles$ do
      $f\_lch \leftarrow xml$.GETNAME()
      $fm$.ADDCHILD("$root$", $f\_lch$)
      for all $arg \in xml$.GETELEMENTS("$arg$") do
          if AFFECTSCG($xml, arg$) then
              $f\_arg \leftarrow arg$.GETFIELD("$name$")
              $fm$.ADDCHILD($f\_lch, f\_arg$)
              $fm$.ADDXOR($f\_arg$, GETVALUES($xml, arg$))
      for all $inc \in xml$.GETELEMENTS("$include$") do
          $fm$.ADDINCLUDES($f\_lch, inc$.GETFIELD("$file$"))
      for all $elem \in xml$.GETELEMENTS("$node$") $\cup$
      $xml$.GETELEMENTS("$param$") do
          $pc \leftarrow f\_lch \wedge$ GETCONDITION($xml, elem$)
          $ln \leftarrow \{pc\} \cup lns$.GET($f\_lch$)
          $lns$.PUT($elem, ln$)
  for all $elem \in lns$.KEYS() do
      if SIZE($lns$.GET($elem$)) $> 1$ then
          for all $l_1, l_2 \in lns$.GET($elem$) do
              $fm$.ADDEXCLUDES($l_1, l_2$)
```

Once no unselected features are left, a single concrete product has been configured and can be launched. By then, the conditional presences are resolved, leading to an architecture without any variation points. The concrete call to `roslaunch` that runs such system – calling the selected launch files with the selected argument values – is then provided to the user.

## D. Extraction

Constructing feature models is a laborious and error-prone task, as mentioned in Section II, but its value is undeniable, especially when aided by automatic extraction procedures.

Our approach to ROS variability revolves around launch files and nodes, since both are concrete source code artefacts that can be analysed. Using static analysis to extract communication channels from ROS nodes has been done by Santos et al. [11] and implemented in HAROS [3], [4], a framework for quality assurance of ROS applications. This framework analyses launch files in order to build a computation graph, but it does not support the feature model structure we propose. We extended its capabilities so that: *(i)* we act on a global view over all launch files; *(ii)* features and attributes are variational; *(iii)* presence conditions are logic formulae, rather than unresolved strings; *(iv)* cross-tree constraints are supported; and *(v)* the computation graph can be calculated as the user selects features.

The process of extracting the feature model is depicted in Algorithm 1. Given a set of launch files, the algorithm iteratively processes each file and creates a feature model. Here, procedure FM creates a new feature model with the given root feature, ADDCHILD adds a parent/child relationship, ADDXOR adds a xor-group to a parent feature, and ADDINCLUDES and ADDEXCLUDES add inclusion and exclusion cross-tree constraints, respectively.

For each launch file, the algorithm calculates a unique name with GETNAME and adds that feature to the feature model. XML tags are processed in order, mimicking the behaviour of

```
<launch>
  <arg name="model" default="kobuki" />
  <arg name="use_teleop" />
  <arg name="vel_topic" value="$(arg model)/vel_cmd" />
  <node name="$(arg model)" pkg="example" type="$(arg
     model)_driver">
    <remap from="vel_cmd" to="$(arg vel_topic)" />
  </node>
  <include file="$(find example)/launch/control.xml">
    <arg name="use_teleop" value="$(arg use_teleop)" />
    <arg name="vel_topic" value="$(arg vel_topic)" />
  </include>
</launch>
```

Fig. 3. Example arguments that may affect the computation graph.

the ROS infrastructure, `roslaunch`, and preserving its semantics. We abstract some of the procedures to retrieve XML elements and fields as GETELEMENTS and GETFIELD, respectively.

Starting with the `arg` tag, only those that affect the architecture are relevant, a test we abstract by AFFECTSCG. If its `value` is set, it is not considered a feature, since every reference to it can be directly replaced. Otherwise, it is an input argument and a feature candidate. It will be effectively a feature if it is used – directly in the launch file or transitively via `include` – *(i)* to instantiate a `node` or `param`; *(ii)* to `remap` communication channels; or *(iii)* to resolve a Boolean condition that, directly or transitively, affects nodes, parameters or communication channels. Note that while an `arg` with a set `value` may not be a feature in itself, it can depend on command-line `arg` that, then, may become transitive features. For example, take the listing in Fig. 3. Assume that the included `control.xml` is the launch file in Fig. 1. The first two `arg` are input arguments, while `vel_topic` is not. The `model` `arg` determines the `name` and `type` of a node, so it affects the computation graph and, thus, is a feature. Both `use_teleop` and `vel_topic` are passed on via `include`, and we know that `use_teleop` determines the nodes launched in that file, making it also a feature. On the other hand, `vel_topic` affects the computation graph by determining the ultimate name assigned to a communication channel with the `remap` tag. However, its value depends on `model`. If `model` was not a feature in itself, it would transitively become one at this point.

Calculating the possible values an `arg` can take, a process abstracted by GETVALUES, can be done in some cases, although it may not be trivial. The simplest case is when the `arg` is used to fully replace the value of an attribute that, by definition in the ROS launch XML specification, can only take values from a given enumeration. One such case is the `type` attribute of a `param`, that can only be one of `"bool"`, `"string"`, `"int"`, `"double"`, `"yaml"` or `"auto"`. Another relatively easy case is an `arg` that is used only in Boolean conditions, which makes it `"true"`, `"false"`, `"0"` or `"1"`. If the `arg` is used in place of the executable `type` for a given node (e.g., `model`), and we have access to the corresponding package in the file system, we can scan the available executables and produce a set of possible values. Lastly, if the `arg` is used in a file path expression, e.g., `"path/to/$(arg file).launch"`, we enumerate all possibilities by inspecting the package's file structure.

The next step processes `include` tags, adding inclusion constraints that force the selection of the called launch file. Note that the same file may be included in multiple launch files, so there is not a simple parent/children relationship between launch files.

Finally, the `node` and the `param` tags are handled in similar fashion. Procedure GETCONDITIONS calculates the presence conditions of these elements, based on the parent launch files and required argument values. Here, the main goal is to identify if elements with the same name are being launched in multiple launch files, causing a conflict between them. In that case, an exclusion constraint is added to the feature model.

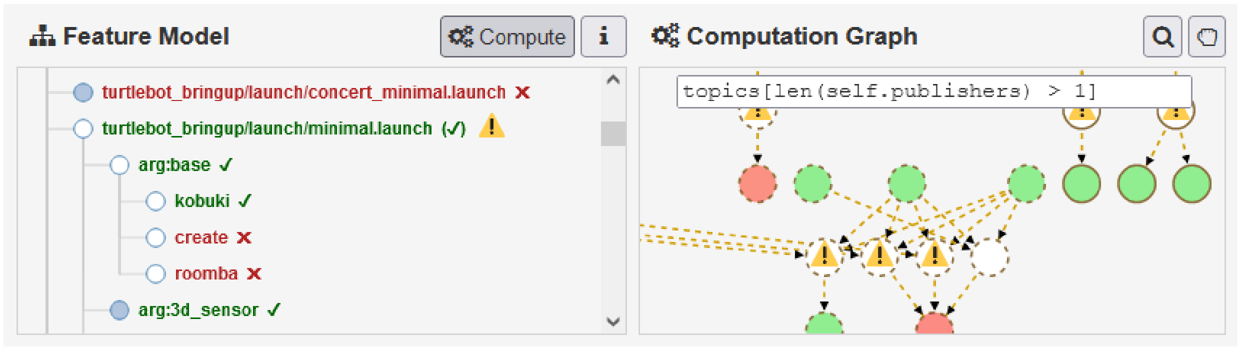
## E. Tool Support and Visualization

Our prototype implementation[5] is, in essence, split into three stages: *(i)* executing HAROS, to produce JSON files with the intra-node models; *(ii)* executing a command-line tool, `harosvar`, that analyses all launch files in a set of packages and produces JSON files containing their feature models; and *(iii)* executing `harosviz`, a modified HAROS visualization server. This server differs from the original in terms of its model visualization. Whereas the original displays a series of pre-computed (static) computation graphs, based on user-provided lists of launch files, we now display an interactive feature model with live computation graph calculation, as illustrated in Fig. 4.

On the left-hand side of this figure, we see an excerpt of the extracted feature model, displayed as a hierarchical tree. Dark filled circles represent collapsed sub-trees, while white circles represent expanded trees. The topmost visible layers correspond to launch files, with one being deselected (in red, with the cross mark in front) and one being selected (in green, with the check mark in front). A parenthesized check mark means that the selection of that feature is derived, in this case because the user selected one of the launch file's arguments (in the second layer), which requires the parent launch file to also be present. If an argument has a default value, it is always displayed among the options. When the tool is unable to calculate an argument's possible values, the user has the option of manually inserting a value of their choice. Lastly, to make evident problematic configurations, we add a warning symbol in front of launch files for which there are potential conflicts, determined by exclusion constraints. The user can click on the symbol to get a list of conflicting files and their conditions. As the user selects features, these conflicts may become impossible (e.g., the selected argument values never lead to the launch of conflicting nodes), and the warning disappears. Once a product is fully configured, the respective `roslaunch` command is provided to the user.

On the right-hand side we can see an excerpt of the variability-aware ROS computation graph that would result from the feature selection on the left side. White circles represent ROS nodes, while coloured circles represent the communication channels. The warning symbols are a visual indication of potential conflicts at the node level as occurring in

[5]https://github.com/git-afsantos/harosvar


Feature Model
turtlebot_bringup/launch/concert_minimal.launch ✗
turtlebot_bringup/launch/minimal.launch (✓)
arg:base ✓
kobuki ✓
create ✗
roomba ✗
arg:3d_sensor ✓
Computation Graph

```
topics[len(self.publishers) > 1]
```

Fig. 4. Screen capture of the prototype visualization tool.

the feature model; in this case, multiple ROS entities clashing with the same name. Drawings in dashed lines represent entities that are not fully resolved with the current feature selection. Since all features start as unselected, the initial computation graph tends to be large, to contain multiple potential conflicts, and to be mostly drawn in dashed lines. As features are selected and deselected, the graph becomes smaller and drawn mostly in solid lines. Users can inspect the attributes, conflicts and conditions of each element. While most of the graph rendering is reused from HAROS, the formatting of presence conditions and the display of potential conflicts are novel.

The visualizer also improves upon HAROS regarding computation graph analysis. HAROS allows users to define custom queries, presumably representing problematic patterns, such as topics with multiple publishers. Queries would run for all manually declared configurations, one by one, and results could be inspected in the visualizer, highlighting affected elements. Our approach turns again towards *live* computation. As the computation graph is resolved, queries are evaluated in real time against the current graph, as shown in Fig. 4 (affected entities are highlighted in red). Moreover, queries may also consider the presence conditions of elements, e.g., collecting only elements that are affected by certain features. Queries are provided in PyFLWOR syntax, a Python-like query language and system based on XQuery and XPath. The example shows the selection of topics with more than one publisher, defined by the query on the top, `topics[len(self.publishers) > 1]`.

## V. Evaluation

The evaluation of variability analysis and model extraction processes often takes into account various performance aspects, both functional and non-functional. Regarding the former, we are interested in assessing how effective the proposed technique is in detecting variability points in launch files. Moreover, we are also interested in the complexity of the resulting feature models, which are evidence of the variability level of the ROS system and represent the configuration effort that would be expected of the user. Non-functional metrics are often time measurements, which in this work would regard the performance of the feature model and computation graph extraction (e.g., parsing and interpretation of files); and their

TABLE I
Evaluation results regarding launch files

| System | LF | LV | LC | $LC_U$ | $LC_A$ |
|---|---|---|---|---|---|
| Kobuki | 21 | 4 | 17 | 17 | 0 |
| TurtleBot2 | 53 | 15 | 38 | 36 | 2 |
| Lizi | 14 | 2 | 12 | 12 | 0 |
| Husky | 137 | 37 | 100 | 98 | 2 |

TABLE II
Evaluation results regarding command-line arguments

| System | A | ACG | AS | $AS_D$ | $AS_V$ | IF | $IF_M$ |
|---|---|---|---|---|---|---|---|
| Kobuki | 18 | 0 | 0 | 0 | 0 | 0 | 0 |
| TurtleBot2 | 195 | 23 | 19 | 15 | 5 | 6 | 2 |
| Lizi | 66 | 12 | 1 | 1 | 0 | 20 | 4 |
| Husky | 391 | 82 | 40 | 40 | 12 | 57 | 4 |

update during multi-step configuration. Thus, this section aims to answer the following research questions.

RQ1 Does the execution time of the technique scale for realistic ROS systems?
RQ2 How effective is the technique in automatically extracting feature models?
RQ3 How complex are the extracted feature models of realistic ROS systems?

To answer these questions, we applied our technique to four distinct mobile ROS robots with which we have experience: Kobuki, TurtleBot2, Lizi and Husky. These robots have received contributions from the ROS community over the years, and are iconic subjects used in robotics research and education.

Table I and Table II summarise the results of our experiments, considering a variety of metrics. Table I is focused on metrics pertaining to launch files, namely how many there are (**LF**); the number of files that never lead to conflicts with other files (**LV**); the number of files that have, at least, one potential conflict (**LC**); and, among **LC**, those that have at least one conflict that is unconditional ($\mathbf{LC}_U$), and those whose conflicts always depend on the values of their `arg` ($\mathbf{LC}_A$). Table II shows our findings related to command-line `arg`, such as their number across all launch files (**A**); how many affect the structure of the computation graph in some measure (**ACG**); the number of `arg`, among **ACG**, that require non-Boolean values (**AS**);

and, among those, how many provide a default value ($\mathbf{AS}_D$) and how many have computed values besides their defaults ($\mathbf{AS}_V$). In addition, we calculate the number of times arguments from **ACG** affect an `if` or `unless` condition (**IF**), as well as the maximum number of times a single `arg` does so ($\mathbf{IF}_M$).

Note that we do not aim to measure the efficacy of the extracted models against a ground truth representing the developers' intentions. There is no reliable method to obtain this ground truth, other than studying the test subjects in depth – reading source code, documentation, contacting the developers, etc., which we leave for a future study.

Regarding RQ1, we did not find any performance bottleneck in the proposed approach. Parsing and interpreting launch files is a fast process. There are no alignment issues when calculating the models – two ROS resources are the same if they have the same name. Calculating computation graphs proved to be almost immediate for all our experiments (ranging from 120 milliseconds to 3.68 seconds), which is also not surprising, given that the involved logic formulae are rather simple, due to the structured nature of launch files.

Regarding RQ2, we consider the overall precision and recall of the approach, resorting to manual observation to validate our results. Our technique is precise by construction, given that only launch files and arguments that affect the computation graph are considered features. Ranges of valid `arg` values are computed only in predictable, deterministic scenarios. All reported conflicts are also confirmed to be present.

In terms of completeness (recall), all features have been detected, according to our model. We have found that many `arg` (the difference between **A** and **ACG**) do not affect the computation graph. Instead, they are used to configure secondary aspects, such as navigation. Boolean `arg` are trivially detected and computed, based on `if` and `unless`. However, the tool was unable to compute any values besides the default for 43 out of 60 **AS** (71.67%). More than half of these (26) are ROS names for nodes and channels that must be provided by the user, seen in TurtleBot2, Lizi and Husky. The remaining arguments (17) pertain to third-party packages, indicating launch and data files to import. Since third-party packages are not part of the subject systems per se, we left their features out of our measurements. Recall that, when value inference fails, the user is still able to provide the intended value via the tool's interface. Note also that we are only concerned with the completeness of the variability extraction technique, which builds on the computation graphs automatically extracted with HAROS using intra-node source code analysis [11]. HAROS allows the user to provide additional information to fix incomplete graphs.

As for RQ3, looking at the number of launch files that realistic ROS systems carry, and considering that the vast majority may induce runtime conflicts, the utility of a tool such as we propose, to support the user in system configuration, becomes evident. It is true that, all things considered, the numbers of **ACG** do not come close to **A**. The true question, however, is not *how many* arguments affect the computation graph, but rather *how* they do so. Out of 117 **ACG**, 57 are Booleans, while 60 belong to **AS**. Most **AS** provide a default value, but most also fall out of the predictable cases for which our tool can compute valid ranges of values. This makes system configuration less intuitive for users, and requires more manual effort to validate the resulting products. We argue that simply being aware of those variability points and their connection to the computation graph is helpful for the user.

Concerning the 57 Boolean **ACG**, we can see that there is a total of 83 **IF**, i.e., there are more decision points than Boolean `arg`, meaning that some are used multiple times. Indeed, some `arg` are used a total of 4 times ($\mathbf{IF}_M$) throughout a single launch file and its inclusions. Fortunately, decision points concentrate at the topmost level; only 3 Boolean `arg` propagate down via `include`, while non-Boolean `arg` are passed down more often.

Overall, the Lizi system stands out in this corpus, in the way that it approaches launch files. It has the lowest **LF** because most of its variability resides in a single *main* launch file, with multiple `arg` that alter its behaviour. Its **IF** is notably high, relatively speaking, amounting to an average of 1.43 `if` per launch file – even though we know they are not distributed equally – while other systems show an average below 0.5. This is due to the other systems, in contrast, following a clone-and-own approach, having multiple similar launch files, changing only some aspects of their configuration. Husky, for instance, has 24 `arg` used in file paths. We are able to calculate half of them, while the other half relates to third-party packages. In reality, 10 of the missing 12 are five repetitions of the same two `arg`, spread over five very similar launch files. This repetition offers simplicity in the one hand, but might overwhelm users with the sheer number of launch files in the other.

## VI. Related Work

Variability has been traditionally implemented in C source code using preprocessor directives (`#ifdef`) to take advantage of conditional compilation. This problem has been tackled using different strategies, such as integrating build systems and their configurations in the analysis, as (major) sources of variability [12]; using mining techniques, in addition to static analysis, to find feature correlations and constraints [13]; or integrating multiple layers, scripts and file types into feature models, considering also that some pieces of information are not always Boolean [14]. While this is not a common problem in ROS, some concepts can be compared, and even transposed. For example, integrating the launch interpreter semantics into the extraction process is a core aspect of our approach, as is the merging of Boolean and non-Boolean information from multiple layers of variability into a single model.

The work of Assunção et al. [15] is an interesting contrast to ours. The driving motivation of their work is to extract feature models that are *variability-safe*, i.e., models where all possible feature combinations are structurally well-formed and unresolved references to undefined elements are forbidden. We, on the other hand, support unresolved references explicitly, and present them to users in the form of warnings and other indicators, so that users can take action.

Schlie et al. [16] focus on model extraction in systems where variability is implemented through clone-and-own. This is something that we anecdotally have seen in ROS, for instance by cloning launch files and changing the parameters they pass down to nodes. Rather than traditional feature models, they build a so-called 150% model; a type of model that represents variability at the implementation level, e.g., showing alternative code blocks when under conditional compilation. Likewise, we define features in terms of tangible source code artefacts, that can be traced back to their origin.

Lastly, in stark contrast to our approach, HyperFlex [17] is a Model-Driven Engineering (MDE) toolkit. Users of HyperFlex build architectural diagrams and feature models using mostly visual representations, which can then be instantiated to concrete implementations, configured via automatically generated code – which, in ROS, corresponds to automatically generating launch files. The tool supports building multiple feature models, targeting different levels of abstraction and, consequently, different users. For example, one feature model, intended for domain experts, might capture the functional variability for a low-level capability, such as perception or mobile manipulation, using vocabulary that is appropriate for this domain. Another feature model, intended for system integrators, would describe the robotic application at a high level of abstraction, referring to the lower-level features while hiding their internal complexity. The connections between levels of abstraction are defined with model-to-model transformations, that the tool also supports. The clear separation of concerns and the use of semantic vocabulary are compelling conveniences for the development of robotic applications – but they require an MDE approach from the start. Our feature models, on the other hand, operate at a much lower level of abstraction. Features correlate directly to source code artefacts without domain-specific meaning, a direct consequence of a reverse engineering approach. One major benefit, though, is that it requires no previous investment in modelling. It can be used by roboticists without training in software engineering, or by software engineers working with existing or legacy systems.

## VII. CONCLUSION

Variability analysis is a current and complex problem in the world of software engineering, but especially so in robotics. As developers shift deeper into modular designs, building a robotic system revolves largely around reusing, configuring and orchestrating third-party components. This can be seen in ROS launch files, files designed for the purpose of system orchestration, which are also a major source of variability.

In this paper, we have proposed a method to automatically extract a variability model of a ROS application, and presented a prototype tool that implements this approach. The interactive view shows how each feature will impact the system's architecture at runtime, as it calculates the resulting ROS computation graph in real time. The tool is also capable of identifying invalid configurations and allows users to execute queries over the model, to identify further issues.

XML launch files are supported by the original ROS and ROS2, but the latter favours Python launch files instead, which make analysis much more of a challenge. Regardless, it is a challenge that we intend to take on.

Another limitation of our approach is that static analysis is sometimes unable to extract a fully resolved computation graph, as evidenced during our experiments. The underlying tool for this, HAROS, requires user-provided extraction hints that supply the missing pieces of information. In our case, such hints must also take presence conditions into consideration, since the way a system's features are integrated and configured certainly has an effect on the final product. How to best approach this is a challenge that we intend to study further.

Lastly, we intend to explore how software analysis tools, such as HAROS, can be altered or reused in such a way that they are able to operate on variational models. The goal is to, ideally, analyse once, and determine all possible configurations that would cause an issue to manifest.

## REFERENCES

[1] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: An open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.

[2] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-oriented Software Product Lines*. Springer, 2016.

[3] A. Santos, A. Cunha, N. Macedo, and C. Lourenço, "A framework for quality assessment of ROS repositories," in *IROS*. IEEE, 2016, pp. 4491–4496.

[4] A. Santos, A. Cunha, and N. Macedo, "The high-assurance ROS framework," in *RoSE@ICSE*. IEEE, 2021, pp. 37–40.

[5] S. García, D. Strüber, D. Brugali, A. D. Fava, P. Schillinger, P. Pelliccione, and T. Berger, "Variability modeling of service robots: Experiences and challenges," in *VaMoS*. ACM, 2019, pp. 8:1–8:6.

[6] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski, "A survey of variability modeling in industrial practice," in *VaMoS*. ACM, 2013, pp. 7:1–7:8.

[7] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *ICSE*. ACM, 2008, pp. 311–320.

[8] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, and A. Egyed, "Reengineering legacy applications into software product lines: a systematic mapping," *Empir. Softw. Eng.*, vol. 22, no. 6, pp. 2972–3016, 2017.

[9] A. Santos, A. Cunha, N. Macedo, R. Arrais, and F. N. dos Santos, "Mining the usage patterns of ROS primitives," in *IROS*. IEEE, 2017, pp. 3855–3860.

[10] P. Estefo, R. Robbes, and J. Fabry, "Code duplication in ROS launchfiles," in *SCCC*. IEEE, 2015, pp. 1–6.

[11] A. Santos, A. Cunha, and N. Macedo, "Static-time extraction and analysis of the ROS computation graph," in *IRC*. IEEE, 2019, pp. 62–69.

[12] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann, "A robust approach for variability extraction from the Linux build system," in *SPLC*. ACM, 2012, pp. 21–30.

[13] B. Zhang and M. Becker, "RECoVar: A solution framework towards reverse engineering variability," in *PLEASE*. IEEE Computer Society, 2013, pp. 45–48.

[14] S. El-Sharkawy, S. J. Dhar, A. Krafczyk, S. Duszynski, T. Beichter, and K. Schmid, "Reverse engineering variability in an industrial product line: Observations and lessons learned," in *SPLC*. ACM, 2018, pp. 215–225.

[15] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, and A. Egyed, "Extracting variability-safe feature models from source code dependencies in system variants," in *GECCO*. ACM, 2015, pp. 1303–1310.

[16] A. Schlie, S. Schulze, and I. Schaefer, "Recovering variability information from source code of clone-and-own software systems," in *VaMoS*. ACM, 2020, pp. 19:1–19:9.

[17] D. Brugali and N. Hochgeschwender, "Managing the functional variability of robotic perception systems," in *IRC*. IEEE Computer Society, 2017, pp. 277–283.