FeatX: Controlled Robot Configuration with Flexible Feature Binding

Jude Gyimah[1], Henriette Knopp[1], Thorsten Berger[1], and Patrizio Pelliccione[2]

Abstract— **Robotic systems often need to be configured for different dynamic execution environments, hardware, or non-functional properties, such as energy consumption. Configuration options, a.k.a. features, can be used to enable, disable, and calibrate different parts of the system, ranging from whole subsystems over components to lines of code. While configuration mechanisms are abundant in robotics, they are limited in expressiveness, and the configuration files are often distributed over the codebase in different artifacts, challenging the consistent declaration and enforcement of dependencies. In addition, robotic systems require flexibility, since features need to be activated or changed at different times in the lifecycle of the system, which can cause intricate dependencies, especially when they depend on other static or dynamic features. To prevent misconfiguration and undefined behavior, such configuration spaces need to be properly declared and managed.**

We present FeatX, a model-based configuration technique. It uses and extends feature models, but accounts for the specific needs in robotics. Specifically, it allows declaring features, their dependencies, as well as the allowed binding times and binding modes, while the configurator enforces correct configuration and reconfiguration, considering intricate semantics of such models. We designed the syntax and semantics of the FeatX language and implemented them in the configurator. Our prototype is implemented for ROS2 with a command-line interface (*ros2cli*). We evaluated it upon realistic (re-)configuration scenarios.

Index Terms— **robots, configuration, feature models, ROS2**

I. Introduction

Service robots, "a category of task-performing robots designed for personal or professional use" [1] are increasingly widespread. They integrate diverse capabilities and are constantly expected to be self-adaptive and reconfigurable enough to sense, think, and act in a given context [2], [3], [4], [5]. Consider platforms such as TIAGo [6], [7], which need to be highly customizable towards different use cases that require perception, navigation, human-robot interaction, and advanced control in healthcare, logistics, and manufacturing. Similarly, in unmanned aerial vehicle (UAV) operations, drones are expected to rapidly detect an oncoming aerial object, gracefully shut down motor functions via an automatic or manual trigger, and deploy a parachute in anticipation of a high-velocity crash.

To this end, most such robots need to be configurable, i.e., have the possibility to enable or disable desired *features* (a.k.a. skills or supported actions) as necessary [6], [7], [2]. This also includes reconfiguration: dynamically starting or shutting down system functionality at runtime to adapt to evolving requirements, detected failures, and unexpected environmental changes [8], [9]. We use the term *feature* to refer to functionalities, components, configuration options (often Boolean) that can be selected or deselected on demand [10]. For example, core features of TIAGo, such as SLAM-based navigation and manipulation, in certain instances, may remain enabled throughout build time and runtime, whereas optional features, such as inventory scanning, fall detection, or teleoperation can be activated on demand during execution. Such features require implementation techniques that allow them to be included or excluded at compile time through preprocessing or static binding, as well as during program loading or runtime execution via dynamic binding [11], [12].

Unfortunately, consistently handling such configurations is difficult and has largely been managed with ad hoc mechanisms in robotics [2]. While the current state-of-the-art (i.e., official, custom and community-contributed solutions) offers some support for implementing configuration binding, consistently managing dependencies and constraints in tandem with these bindings has always been challenging [13], [14], [15], [16], [17]. To the best of our knowledge, there is no configuration language expressive enough to model features, dependencies, and such binding modalities. While ROS offers various configuration mechanisms, they neither provide a central overview, nor allow modeling dependencies and binding modalities, let alone consistent configuration and reconfiguration verification. For instance, when reconfiguring a system, selecting and deselecting dependent features that may be changeable (***dynamic***) or unchangeable (***static***) at compile time (***early***) or at runtime (***late***), in the absence of an effective mechanism for managing this flexibility, can easily lead to misconfigurations and system failures.

We present FeatX [18], a model-based configuration technique around a domain-specific language (DSL) [19] to centrally manage the configuration space of robots. The DSL enables specification of features, their dependencies, as well as binding modalities that determine when and how features are activated. These specifications are processed by a configurator tool. Building on traditional feature models—a familiar concept in robotics [20], [21], [22]—we formally define the DSL syntax and semantics and implement them within the configurator to ensure consistent and correct management of system reconfiguration. Our prototype is integrated into the ***ros2cli*** command-line tools, ensuring a seamless workflow when building highly reconfigurable robotic systems with ROS. We evaluated FeatX with case studies and tested our implementation in a Gazebo-simulated warehouse scenario.

II. Motivation and Background

We illustrate the need for better configuration support in robotics, particularly flexible and controlled feature binding, which is challenging to realize consistently, as we will show.

[1] Faculty of Computer Science, Chair of Software Engineering, Ruhr University, Bochum, Germany [jude.gyimah, henriette.knopp, thorsten.berger]@rub.de

[2] Gran Sasso Science Institute (GSSI), L'Aquila, Italy patrizio.pelliccione@gssi.it

We also introduce the necessary background for the remainder.

A. Illustrative Robotics Example

Figure 1 illustrates a complex robot system as a feature model (explained shortly, in Sect. II-C). Consider the feature *CPU* that implements generic CPU control code (e.g., scheduling). It could have the options ARM or X86. Since processor choices typically cannot be changed at runtime, it is expected that both *ARM* and *X86* implementations would be selected and compiled into the system at compile time (henceforth referred to as bound ***early***). Implementation techniques for this could be the C preprocessor with conditional compilation directives (e.g., #IFDEF) or a configurable build system (similar to KBuild from the Linux kernel [23], [24]). In this way, only the source codes of selected features are compiled. Given the conditions of such an implementation and considering that the CPU hardware is not swappable at runtime, bindings of these CPU features must be ***static***, so changing a CPU selection demands recompilation.

Consider the feature *ImageDisplay* loaded at runtime to track and visualize an autonomous robot's trajectory within a reconstructed 3D operating environment. Such a feature does not need to be activated at compile time, since it is only required when there is motion. Depending on the concrete implementation and the hardware, unloading it at runtime might be difficult. In such a case, this feature can be loaded ***late***, but bound as a ***static*** feature—it can only be loaded at runtime and persists until system shutdown.

Consider the feature *Camera* for which multiple options, such as *RaspberryPiCamera* and *IntelRealSenseR200*, are made available at runtime to support perception requirements. Both features should be bound ***dynamic***, so the actual cameras used can be changed at runtime. For instance, when no depth estimation is necessary, the *RaspberryPiCamera* suffices. However, since both features impose some startup overhead for their first start (but not for subsequent starts), they should be loaded early already when the robot system starts up. So, one would implement both features in a way that they can be loaded and unloaded (i.e., as ROS plugins), but a configuration mechanism should allow configuring them to be bound ***dynamic*** and ***early***, the former to allow them to be unloaded (and loaded again) freely at runtime (***late***), without their startup overhead.

Finally, consider the feature *HardwareAcceleration*, which can potentially be configured as ***dynamic*** and ***late*** (at runtime). It can be loaded at runtime to boost *GPU* operations when high-definition image processing is required. Such a feature often has its startup time tied to the execution state of the system because there is an associated startup cost (i.e., loading GPU drivers, allocating memory or initializing contexts), which depends on the timing of the feature's activation. For example, if loaded early, memory and performance will be negatively impacted, as opposed to scheduled ***late*** (runtime) loading for specialized reasons. This implies that this feature should be implemented as a plugin that can be toggled at runtime (***late***) and unloaded once it is no longer required.

TABLE I: Possible feature binding and configuration mechanisms that can be used with ROS

Binding mechanism	Binding type	ROS1	ROS2
launch files	static	✓	✓
parameter files (YAML)	static	✓	✓
c++ preprocessor	static	✓	✓
manual composition	static	✗	✓
dynamic_reconfigure	dynamic	✓	✗
pluginlib	dynamic	✓	✓
component plugin	dynamic	✗	✓
runtime parameters	early and late	✓	✓
static member function callback	early	✓	✓
virtual functions	early	✓	✓
boost::bind	early	✓	✗
std::bind	early	✗	✓
polymorphism	late	✓	✓
importlib	dynamic	✓	✓

B. Binding and Configuration Mechanisms

Table I shows possible binding mechanisms—implementation techniques to realize the flexible binding of features in robotics. They can be categorized by time as ***compile or startup time*** or ***runtime***, and by mode, i.e., ***static*** (not changeable after binding) or ***dynamic*** [25] (changeable after binding). In ROS, the build time workflow, initiated by build systems `catkin` or `colcon`, configures targets, compiles, links and installs node implementations. A common static mechanism is conditional compilation, which as explained above, is often realized with a configurable build system and the C preprocessor. Configuration parameters are dynamic; changing their values can always change execution at runtime. Dynamic configuration trades static optimization of systems (e.g., memory footprint or performance) for flexibility.

Considering our examples, the ***static early*** *CPU* features could be implemented via conditional preprocessor statements that compile all features that are selected in the robot's configuration into the main program at build time. The feature *Graphics* that requires dynamic binding could be realized with a plugin library for easy runtime loading and unloading. A simpler implementation could be with runtime parameters, which allow ***dynamic*** binding, either ***early*** at system startup time or ***late*** at runtime.

We distinguish between **binding mechanisms** as the implementation techniques that determine the possible binding times (***early*** or ***late***) and binding modes (***static*** or ***dynamic***) of features, and the actual **configuration**—the selection of features activated in the robot system. Once a robot system is implemented, users should be able to configure features. They should be able to select them, while also selecting the actual binding time (whether activated ***early*** or ***late***) and binding mode—of course these need to be allowed by the actual implementation. For instance, the feature *ImageDisplay* can be configured as ***static late***, which implies that it is loaded at runtime, but cannot be unloaded until system shutdown. *RaspberryPiCamera* can be configured as ***dynamic early*** bind, ensuring that it is automatically loaded at startup, while it can be unloaded at runtime later on. Finally, *HardwareAcceleration* configured as ***dynamic late***, so loading it only when computational demands justify it.

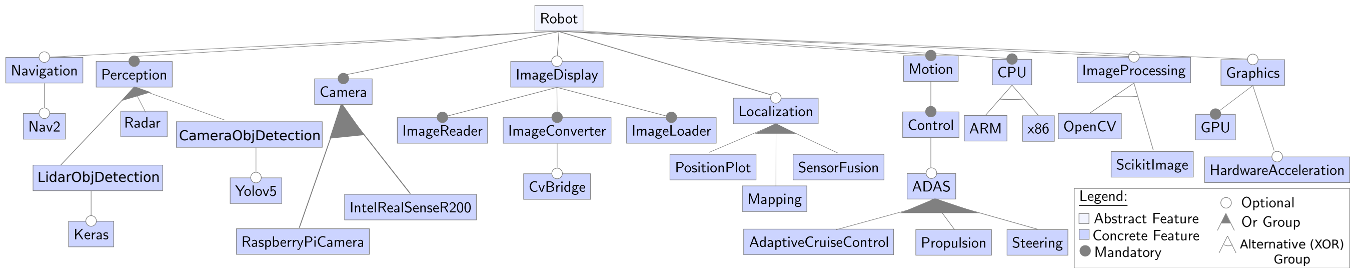

Fig. 1: Modeled features of an example robotic system

Limitations. Implemented features often depend on other features, since they need others' functionality. Such dependencies (especially transitive dependencies) can be intricate, since the code implementing a feature can be scattered [26] across the codebase, or tangled with other features' code. As such, features and configuration options need to be modeled centrally, capturing their dependencies, so that configurator tools can assure that concrete configurations (i.e., concrete selections of features) adhere to those dependencies. In fact, real-world dependencies can be complex [17].

Middlewares such as ROS provide configuration and binding mechanisms, such as runtime parameters, launch files, or plugin libraries, while generic techniques, such as feature toggles or the C++ preprocessor can be used as well. The many different mechanisms cause the following limitations.

> **Limitation 1:** The specifications of configurations options is also distributed over different types of files, which challenges keeping a centralized overview of a robot's configuration spaces, as well as declaring features and their dependencies when declared in different mechanisms. FeatX addresses this challenge by focusing on a central model where configuration spaces can be declared.

Many mechanisms diverge in expressiveness and also prevent keeping an overview of a robot's configuration space.

> **Limitation 2:** Existing mechanisms are insufficient, often lacking the possibility to formally declare dependencies. The mechanisms' expressiveness also varies greatly, while each is tailored towards enforcing a limited set of binding modalities. The FeatX DSL builds upon the well-established notation of feature models, which allow declaring features and dependencies in a hierarchical and intuitive notation.

C. Feature Models

Feature models are intuitive, tree-like representations of features and their dependencies [27], [28]. The typical graphical notation is shown in Fig. 1. Features are labels representing a configuration option or a part of the system [10], [17]. Features are organized in a hierarchy; they can be mandatory (always enabled, which helps structuring the model) or optional (they can be selected or deselected); they can be part of a feature group, typically OR (at least one feature should be selected) or XOR (only one feature should be selected). Additionally, so-called cross-tree constraints, which cannot be expressed using these means, are added as expressions (typically using Boolean logic).

Feature models are input to interactive configurator tools, such as Gears, pure::variants, FeatureIDE, HyperFlex [29], or the Linux kernel configurator. These tools support users creating valid configurations (concrete selections of features, i.e., mappings of features to values), often in an interactive process. While the visual syntax shown in Fig. 1 is intuitive and used in some tools, it does not scale well with the number of features. Like various tools (e.g., Linux kernel configurator with the Kconfig language) FeatX relies on a textual syntax.

Over decades, many extensions have been provided to improve the expressiveness of feature models [30]. Several extensions address runtime configuration [31], [32], [5], feature types (e.g., integer, String) and more expressive dependencies. **Limitations.** While feature models address the limitations of the mechanisms discussed above, they are not sufficient to handle the intricacies of the different binding times and modes required in robotics illustrated above. This challenges the consistent handling of dependencies. For instance, even dynamic features can become static when depending on other static features, which can be hard to enforce over transitive dependencies. To the best of our knowledge, no extension of feature models or similar configuration languages supports specifying binding times and modes per feature.

> **Limitation 3:** Feature models lack support to consistently handle binding times and modes, to prevent invalid configurations and potentially unsafe robotic behavior. Addressing this gap requires designing languages that allow specified features together with their allowed binding times and modes, to be consistent with the actual implementation, as well as carefully designed and formally defined semantics—to be implemented in the configurator tools integrated with robotics middleware. FeatX provides such a DSL and a configurator integrated with ROS, supporting configuration and reconfiguration scenarios.

III. FeatX Overview and Prototype

FeatX supports roboticists in centrally modeling, configuring, and reconfiguring the configuration space of robots. It consists of a DSL and a configurator that actuates a set of binding mechanisms to actually bind the features enabled in a configuration. FeatX can be used by (i) *robot manufacturers* who build a robotic software platform and define the features in the FeatX feature model; (ii) *developers* who implement features with concrete binding mechanisms and declare them in the feature model by defining their hierarchy, dependencies, and allowed binding times and modes (consistent with the binding mechanisms implemented); and (iii) *operators and end-users*, who create initial configurations and configure

```
fr0b0@fr0b0-Latitude-5340: ~/ros2_ws
fr0b0@fr0b0-Latitude-5340:~/ros2_ws$ ros2 featx start_config
[INFO] [1772706861.000434295] [configurator]:  *Feature (navigation): bindings n
ot consistent
[INFO] [1772706861.000617579] [configurator]:   --- (1) issue(s) detected ---
[INFO] [1772706861.000888650] [start_config]: Fix issue(s), build and rerun the
command again
fr0b0@fr0b0-Latitude-5340:~/ros2_ws$ 
```

Fig. 2: ros2cli extended with FeatX

robots for a target runtime environment; as well as they reconfigure robots as needed.

While our main contribution is independent of a concrete middleware, we realized our prototype by integrating it with ROS2. Figure 2 shows the command-line interface of FeatX. It is implemented using ***ros2cli***, ROS2's command-line interface, which comprises a collection of terminal tools that can be used to inspect and control ROS2 components (e.g., nodes, topics, services, parameters, packages). It provides default sub-commands, such as ros2 node and ros2 topic, to facilitate utility operations directly from the shell. However, its interface can also be extended with custom commands linked to bespoke applications controlled via the console. Thanks to this integration, FeatX does not require extra tooling to enforce control from the command line.

Recall that ROS offers many configuration and binding mechanisms, both native and third-party, as was shown in Table I. FeatX incorporates some of these mechanisms such that when a `load` or `unload` command is valid (i.e., its bindings and dependencies allow it), the processing element implemented as a class is instantiated or destroyed. Note that FeatX could either replace mechanisms or be used together with them. From the feature model, it would be easy to generate custom configuration files in one of the existing mechanisms.

Our prototype implementation uses and extends plugin-based binding mechanisms implemented upon `pluginlib` (for C++) and `importlib` (for Python), to realize and enforce all binding times and modes. **Dynamic** features can be implemented as such plugins (see Listing 1). **Static** features can be implemented as conditional compilation directives using conditional source-code inclusion through the C++ preprocessor (see Listing 2). Our current prototype provides guidelines for using the C++ preprocessor, while a future implementation could easily generate header files with one macro per feature according to the current configuration.

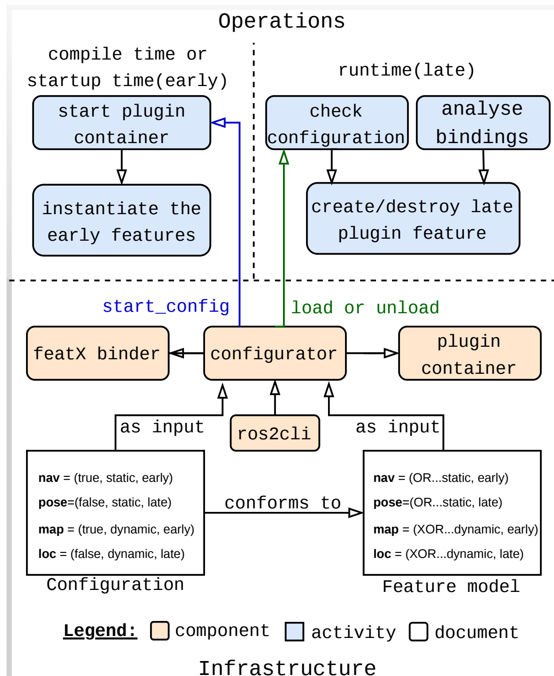

Fig. 3: FeatX infrastructure and operations

```
1  #include "rclcpp/rclcpp.hpp"
2
3  namespace featx_plugin{
4      class Nav2: public rclcpp::Node{
5          public:
6              Nav2(const rclcpp::NodeOptions
                   &options):Node("nav2", options){
7                  RCLCPP_INFO(this->get_logger(), "Hello
                       nav2");
8  }};}
9
10 #include "rclcpp_components/register_node_macro.hpp"
11 RCLCPP_COMPONENTS_REGISTER_NODE(featx_plugin::Nav2)
```

Listing 1: Example of a feature implemented as a plugin

```
1  #ifdef ARM
2      #include "../include/robot/arm_cpu.hpp"
3  #elif !defined(ARM)
4      #pragma message("ARM CPU not selected")
5  #endif
6
7  #ifdef x86
8      #include "../include/robot/x86_cpu.hpp"
9  #elif !defined(x86)
10     #pragma message("x86 CPU not selected")
11 #endif
```

Listing 2: Example of a conditionally compiled feature

Figure 3 provides an overview of FeatX. The bottom shows a feature model (see Listing 3 for an excerpt of its textual syntax) that declares all features, together with their allowed binding times and modes, and all dependencies. This shows an example feature definition. Configurations (see Listing 4 for an excerpt of its textual syntax) conform to the model. The FeatX DSL is an external DSL processed by a JSON parser. As shown in Listing 3, for each feature, a name, group dependency, optionality, inclusion or exclusion constraints, and the allowed bindings are specified. Similarly, to configure declared features, entries that specify a selection status, binding time and binding mode are created separately in a configuration file (see Listing 4). In the middle of Fig. 3, the main components are shown, while at the top, the workflow of activating and deactivating early and late features is illustrated.

```
1 { "name": "navigation",
2   "group": "",
3   "isOptional": false,
4   "includes": "",
5   "excludes": "",
6   "bindingTimeAllowed": "Early",
7   "bindingModeAllowed": "Static" }
```

Listing 3: A single FeatX feature definition

```
1 { "name": "navigation",
2   "isSelected": true,
3   "bindingMode": "Static",
4   "bindingTime": "Early" }
```

Listing 4: A single FeatX configuration entry

FeatX works as follows. A user creates a configuration, which needs to be consistent with the feature model (i.e., its semantics, cf. Sect. IV-B). FeatX extends ***ros2cli*** with a

new command `featx` that can be combined with either a `start_config`, `load` or `unload` sub-command [33], [13], [34], [35]. The sub-command `start_config` validates the configuration and with the component FeatX binder starts the plugin container and instantiates and executes all features that are configured as selected and ***early***. As such, it can be manually run before startup or within the startup of ROS. The reconfiguration sub-commands `load` and `unload` validate the new configuration and when valid, perform the loading or unloading of features against a set of reconfiguration rules. Also, before a feature is loaded, if dependencies exist, they are checked. The `load` sub-command activates a feature by reading it into memory and the `unload` sub-command deactivates the feature by removing the executing instance from memory. If invalid, an error message is displayed.

IV. The FeatX DSL

We now describe and justify the abstract syntax and the semantics of our DSL. The latter are only shown partially, for full details we refer to the appendix [18].

A. Abstract Syntax

We rely on feature model syntax (cf. Sect. II-C) we extend with the notion of allowed binding time and mode. The meta-model in Fig. 4 illustrates our formal definitions.

We define the attributes of a feature as follows. Let ID be the set of feature names in the form of string identifiers, $\mathsf{Group} = \{\mathsf{OR}, \mathsf{XOR}\}$ be the type of feature groups supported by a feature model, $\mathbb{B} = \{0, 1\}$ is a Boolean value, $\mathsf{BTime} = \{$***early***, ***late***$\}$ be the supported binding times, and $\mathsf{BMode} = \{$***static***, ***dynamic***$\}$ be the supported binding modes. FM is the set of all possible feature models. Each feature model $fm \in \mathsf{FM}$ is a set of features, so $\mathsf{FM} = \mathcal{P}(\mathcal{F})$, where:[1]

$$\mathcal{F} = \mathsf{ID} \times \lceil \mathsf{ID} \rceil \times \lfloor \mathsf{Group} \rfloor \times \mathbb{B} \times \mathcal{P}(\mathsf{BTime}) \backslash \{\emptyset\} \times \mathcal{P}(\mathsf{BMode}) \backslash \{\emptyset\} \times \lfloor P(\mathsf{ID}) \rfloor \times \lfloor P(\mathsf{ID}) \rfloor$$

If $(n, p, g, o, t, m, i, e) \in \mathcal{F}$, then n is the feature name, p is the parent of the feature with $\top$ for the root element, g indicates the group i.e., OR or XOR (alternative), o determines whether a feature is optional (True) or mandatory (False), t and m are the allowed binding times and the allowed binding modes (consistent with the feature implementation), i and e are cross-tree inclusion and exclusion constraints, respectively.

B. Semantics

We map the abstract syntax to a semantic domain Confs with the following semantic function $[\![\cdot]\!]_{FM} : \mathsf{FM} \longrightarrow \mathcal{P}(\mathsf{Confs})$, which maps a feature model into the set of all possible configurations, where $\mathsf{Confs} : \mathsf{ID} \longrightarrow (\mathbb{B} \times \mathsf{BTime} \times \mathsf{BMode})$. So, a configuration triple consists of a selection status ($\mathbb{B}$), binding time (BTime), and binding mode (BMode). We refer to the appendix [18] for the definition of the semantic function in denotational style. It restricts the configuration space with denotations; for instance, that the binding time and mode in a configuration need to be consistent with allowed binding times and modes defined in the model (recall the definition

[1] $\lceil X \rceil = X \cup \{\top\}$ and $\lfloor X \rfloor = X \cup \{\bot\}$

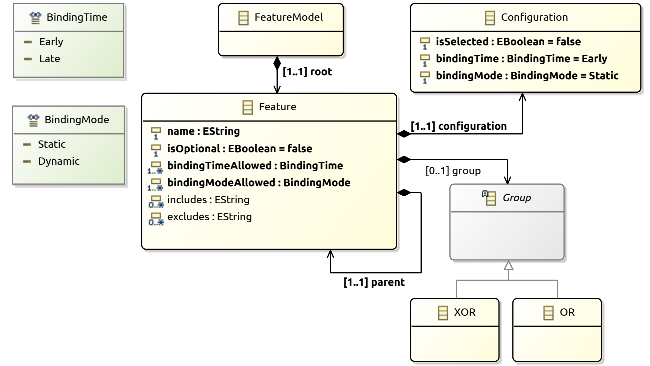

Fig. 4: FeatX's abstract syntax shown as a meta-model

of $\mathcal{F}$) above. The denotations use helper functions, which we provide here for illustrating our main contribution, a formalization of the semantic interactions of the binding time and mode. As seen in our appendix, we investigate the configurator behavior by checking all possible combinations of configuration values for features that depend on each other or exclude, which we used to derive a rule set, specifically five design rationales, see Tables II and III:

- DR_{R1}, DR_{E1}: If a feature is not selected, the features on which it depends may have any configuration.
- DR_{R2}: If a feature is selected, the feature it depends on must be static, provided that DR_{R3} also holds. For example, in Figure 1, if the feature *RaspberryPiCamera* is selected, the feature *Camera* must be both ***static*** and selected to prevent deselection at runtime.
- DR_{R3}: If a feature is both selected and bound ***early***, then any feature it depends on must also be bound ***early***, and DR_{R2} must hold. For example, if a feature like *RaspberryPiCamera* is selected and bound ***early***, the feature *Camera* is likewise bound ***early***, thereby preventing undefined behavior that could arise when *Camera* were unavailable.
- DR_{E2}: If a feature X is selected and excludes feature Y, then feature Y cannot be selected. However, this exclusion does not constrain binding time or binding mode. For example, if *LidarObjDetection* is bound ***dynamic early*** and *CameraObjDetection* is bound ***dynamic late***, both features could still become active simultaneously at runtime. To prevent this, we restrict feature selection. In the case of the feature *ImageProcessing*, this implies that *OpenCV* can be selected only if *ScikitImage* is deselected. While this enforces constraint correctness, it may result in configurations where certain features become locked (i.e., cannot be selected or deselected).

We use the above design rationales to define the helper functions *req* and *excl*, where *req* specifies valid configurations for feature pairs (n, m) in which n requires m.

If $\delta \in \mathsf{Confs}$ and $n \in \mathsf{ID}$, we write $\delta(n)_{\mathsf{sel}}$ for the first component of the valuation (the selection state), $\delta(n)_{\mathsf{tim}}$ for the second one (the binding time), and $\delta(n)_{\mathsf{mod}}$ for the third one (the binding mode). A valid configuration of the modeled feature *Yolov5* could be $\delta(\mathit{Yolov5}) = (\mathsf{True}, \mathsf{Early}, \mathsf{Static})$.

The function *req* (n requires m) is defined as:

TABLE II: Valid feature configurations after resolving the requires dependency ($X \Longrightarrow Y$) in the feature model.

No	Early[1]	Late[2]	X			Y			Description
			$\delta(X)_{\text{mod}}$	$\delta(X)_{\text{tim}}$	$\delta(X)_{\text{sel}}$	$\delta(Y)_{\text{mod}}$	$\delta(Y)_{\text{tim}}$	$\delta(Y)_{\text{sel}}$	
DR_{R1}	✓	✓	_	_	False	_	_	_	If X is not selected, then Y can have any configuration.
DR_{R2}	✓	✓	_	_	True	Static	_	True	If X is selected, Y needs to set to Static and be selected and DR_{R3} needs to hold.
DR_{R3}	✓	✓	_	Early	True	_	Early	True	If X is bound early, Y needs to be bound early and DR_{R2} needs to hold.

[1] Configuration is allowed (✓) / disallowed (✗) early, i.e., before runtime.
[2] Configuration is allowed (✓) / disallowed (✗) late, i.e., at runtime.

TABLE III: Valid feature configurations after resolving the excludes dependency in the feature model.

No	Early[1]	Late[2]	X			Y			Description
			$\delta(X)_{\text{mod}}$	$\delta(X)_{\text{tim}}$	$\delta(X)_{\text{sel}}$	$\delta(Y)_{\text{mod}}$	$\delta(Y)_{\text{tim}}$	$\delta(Y)_{\text{sel}}$	
DR_{E1}	✓	✓	_	_	False	_	_	_	If X is not selected, there are no constraints on Y.
DR_{E2}	✓	✓	_	_	True	_	_	False	If X is selected, Y cannot be selected.

[1] Configuration is allowed (✓) / disallowed (✗) early, i.e., before runtime.
[2] Configuration is allowed (✓) / disallowed (✗) late, i.e., at runtime.

TABLE IV: Configured RB-1-BASE feature bindings

Feature name	bindingTime	bindingMode	isSelected
navigation	early	static	true
to_loading	late	dynamic	false
to_delivery	late	dynamic	false
to_base	late	dynamic	false
elevator	early	static	true
lift	late	dynamic	false
drop	late	dynamic	false
footprint	early	static	true
circular	late	dynamic	false
square	late	dynamic	false
shelf	early	static	true
attach	late	dynamic	false
localization	late	static	false
init_checks	late	static	false
boot	late	static	false
pose	late	static	false

$$\mathsf{req}(\delta(n),\delta(m)) = \begin{cases} \text{true if } \delta(n)_{\mathsf{sel}} = \mathsf{False} \\ \text{true if } \delta(n)_{\mathsf{sel}} = \mathsf{True} \text{ and } \delta(m)_{\mathsf{mod}} = \mathsf{Static} \\ \quad \text{and } \delta(m)_{\mathsf{sel}} = \mathsf{True} \text{ and if } \delta(n)_{\mathsf{tim}} = \mathsf{Late} \\ \text{true if } \delta(n)_{\mathsf{sel}} = \mathsf{True} \text{ and } \delta_{\mathsf{mod}}(m) = \mathsf{Static} \\ \quad \text{and } \delta(m)_{\mathsf{sel}} = \mathsf{True} \text{ and if } \delta(n)_{\mathsf{tim}} = \mathsf{Early} \\ \quad \text{and } \delta(m)_{\mathsf{tim}} = \mathsf{Early} \\ \text{false otherwise} \end{cases}$$

The function *excl* describes the valid feature configurations for two features n and m, where n excludes m:

$$\mathsf{excl}(\delta(n),\delta(m)) = \begin{cases} \text{true if } \delta(n)_{\mathsf{sel}} = \mathsf{False} \\ \text{true if } \delta(n)_{\mathsf{sel}} = \mathsf{True} \text{ and } \delta(m)_{\mathsf{sel}} = \mathsf{False} \\ \text{false otherwise} \end{cases}$$

V. Evaluation

We evaluated FeatX from three main perspectives: (i) consistency, via case studies, (ii) practicality, via a Gazebo warehouse simulation, and (iii) functional correctness, via a testing framework. Figure 5 models an RB-1-BASE robot, in a logistical context. It has 16 features, which we modeled in Fig. 5. Static features were implemented as composed ROS2 nodes enabled at build time (via C++ preprocessor flags), whereas dynamic features were implemented as `pluginlib` plugins. Our configuration is shown in Table IV. We then evaluated our prototype through case studies and a Gazebo-based warehouse simulation.

TABLE V: FeatX configuration correctness results

Metric	Value
Number of generated misconfigurations (cases)	20
Issues count per misconfiguration	12–21
Median issues count	16
Total number of generated issues	324
Total number of detected issues	324 (100 %)

Case Studies. We smoke-tested our configurator to verify consistency and showcase our binding mechanisms in use. The goal was to test the correctness and consistency of our configurator on limited-functionality packages, to validate core configurator behavior while simultaneously monitoring system events and logs. First, we randomly generated 20 FeatX configurations (i.e., cases), with each representing a unique variant of our RB-1-BASE robot in Fig. 5. For each case, the *bindingTime*, *bindingMode*, *isSelected*, *bindingTimeAllowed* and *bindingModeAllowed* attributes were randomly misconfigured to introduce configuration issues. Each case was then parsed, processed and validated by our configurator. The cumulative results of issues detected are shown in Fig. 6. The statistics show that, of the 324 existing configuration issues introduced, our configurator detected all of them (100 %). The detected issues comprised 239 binding inconsistencies, 40 dependency mismatches, and 45 selection inconsistencies, with a median of 16 issues generated per case (see Table V). Furthermore, detected issues were logged with structure and intent such that they were actionable and easy to fix. All data (i.e., scripts, cases, data-sheet, logs and packages) together with replication instructions are available in our replication package. In addition, our appendix and technical documentation [18] provide detailed implementation and usage support.

Simulation Test. Next, we tested FeatX in a simulated warehouse environment, to validate our control logic in a precise yet resource efficient domain. Here, the robot is

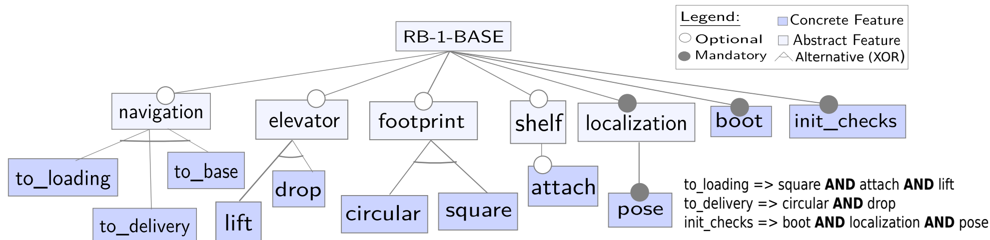

Fig. 5: Modeled RB-1-BASE features

expected to navigate to a loading position, attach to a cart, move the cart to a shipping position, offload the cart, and finally navigate back to the initial position (see Fig. 7). Using RViz, Gazebo and the ***ros2cli***, we visualized and executed FeatX commands to follow a warehouse operator's control sequence, observing the resulting motion and dynamics under live system functionality. We triggered load and unload events via the console to select and deselect modeled RB-1-BASE features and complete the mission. This demonstrates our flexible binding technique in practice and shows that the FeatX configurator can consistently control runtime reconfigurations by approving valid changes and gracefully rejecting invalid ones in a realistic simulated scenario (see demo) [18]. Our warehouse simulation setup is publicly available here.

Test Framework. Using the `pytest` framework, we verified functional correctness in our configurator logic. The authored test suite (i.e., `test_configurator.py`) mainly covers tests surrounding file I/O operations, such as reading, parsing, searching and updating model and configuration schemas with safeguards. These were implemented mostly in anticipation of missing resources, service failures and invalid reconfiguration operations. They provide internal verification meant to establish functionalities such as ensuring binding consistency, enforcing feature dependency rules, constraining and validating feature selections and graceful handling of errors.

VI. Related Work

Feature models can be seen as the most popular notation to model configuration spaces in software [36]. Upon the original FODA notation [27], many concrete languages have been created [30], such as Clafer, Linux Kconfig, CVL, UVL, and CDL. Open-source and commercial tools support feature modeling, such as pure::variants, Gears, and FeatureIDE. Several extensions address runtime configurability; for example, Lotz et al. [5] model dynamic functional and non-functional variability in robotic systems using SmartTCL and VML [31], [32]. Most closely related to us is Rosenmueller et al. [11], who realize flexible binding modes by combining static and dynamic binding for a specific mechanism. In robotics, while traditional feature models have been used [20], [21], [22], none support binding times and modes per feature. The ability to configure features either as static or dynamic, and parts of the underlying semantics are inspired by the three-state-logic-based semantics of the Linux Kconfig language [17], [37], which allows binding kernel features either directly into a monolithic kernel or as loadable modules.

HyperFlex [29] is a feature-model-oriented tool for autonomous robotics, capable of modeling architectures with components, interfaces, connectors, and component wiring. Feature model formalisms are used to represent system variability and constraints. FeatX also targets robotics, but the language and tooling differs substantially.

Frameworks and middlewares such as ROS, Orocos, RT-Component, Rock, and SmartSoft try to simplify robotics development. They generally provide component models which define a set of architectural elements (e.g. interfaces and connectors). They also provide runtime infrastructure, for instantiating, connecting, configuring and activating components. While ROS became a de facto standard, its limitations lie in selecting, integrating, and configuring components with dependencies [33]—a problem that can be solved with powerful configuration techniques known from software-engineering practice and research. Peldszus et al. [9] survey reconfiguration techniques in robotics, especially ROS, revealing the gaps in expressiveness, which motivated FeatX.

Finally, like FeatX, robot-specific feature modeling tech-

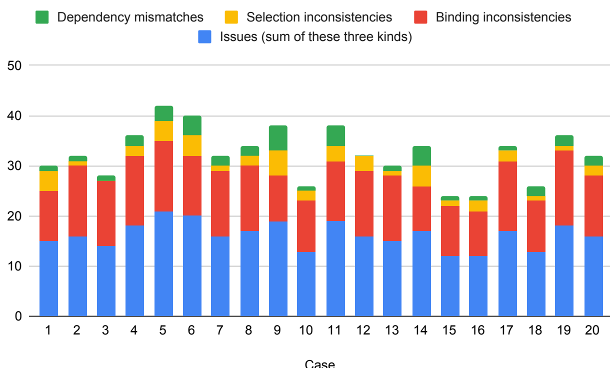

Fig. 6: Statistics on the detection of misconfigurations

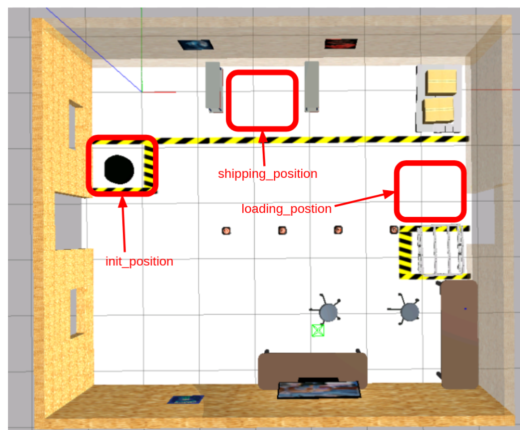

Fig. 7: Mapped mission sequence of simulation

niques exist. Cabello et al. [38] create a feature-modeling language and use it to model mobile robots; which allows them to automatically derive concrete variants through feature configuration. Buchmann et al. [39] propose a model-based technique inspired by feature models to derive robot variants.

VII. Conclusion

We presented FeatX, a configuration technique for ROS comprising a configuration DSL inspired by and extending feature models, together with a configurator tool. The main benefit is the central representation of features (i.e., configuration options), dependencies, and binding modalities. It can be used together with existing configuration mechanisms (e.g., specialized configuration files could be created), or it could potentially replace the decentralized mechanisms. We systematically designed and formalized the DSL semantics and implemented them in the configurator, which we integrated with ROS in our prototype. We validated FeatX based on correctness, consistency, and practicality through test cases, case studies, and a Gazebo-based simulation.

In future work we plan to extend the expressiveness of FeatX, with the ability to model numeric (integer, float) or string features, together with arithmetic or string constraints. We plan experiments that demonstrate the real-world benefits of our underlying feature binding technique. We will measure configuration latency, while performing build time and runtime overhead checks. We will also investigate scalability and performance via large-scale systems with complex feature dependencies and constraints. Finally, we intend to improve the concrete syntax and tooling, including for instance configuration auto-completion and diagnostics, while providing better editing support for the model itself (e.g., via a language server protocol).

References

[1] ISO, Robotics — Vocabulary, 2021, https://www.iso.org/standard/75539.html.

[2] D. Brugali, “Modeling variability in self-adapting robotic systems,” Robotics and Autonomous Systems, vol. 167, p. 104470, 2023.

[3] S. García, D. Strüber, P. Pelliccione, T. Berger, and D. Brugali, “Robotics software engineering: A perspective from the service robotics domain,” in ESEC/FSE, 2020.

[4] M. Ceccarelli, Service Robots and Robotics: Design and Application. IGI global, 2012.

[5] A. Lotz, J. F. Inglés-Romero, C. Vicente-Chicote, and C. Schlegel, “Managing run-time variability in robotics software by modeling functional and non-functional behavior,” in BPMDS, 2013.

[6] S. García, D. Strüber, D. Brugali, A. Di Fava, P. Pelliccione, and T. Berger, “Software variability in service robotics,” Empirical Software Engineering, vol. 28, no. 2, p. 24, 2023.

[7] 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, 2019.

[8] M. Rosenmüller, N. Siegmund, M. Pukall, and S. Apel, “Tailoring dynamic software product lines,” in GPCE, 2011.

[9] S. Peldszus, D. Brugali, D. Strueber, P. Pelliccione, and T. Berger, “Software reconfiguration in robotics,” Empirical Software Engineering (EMSE), vol. 30, no. 3, pp. 1–51, 2025.

[10] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, “What is a feature? a qualitative study of features in industrial software product lines,” in SPLC, 2015.

[11] M. Rosenmüller, N. Siegmund, S. Apel, and G. Saake, “Flexible feature binding in software product lines,” Automated Software Engineering, vol. 18, pp. 163–197, 2011.

[12] V. Chakravarthy, J. Regehr, and E. Eide, “Edicts: implementing features with flexible binding times,” 2008, pp. 108–119.

[13] O. Robotics. (2024) Ros 2 documentation. [Online]. Available: https://docs.ros.org/en/iron/The-ROS2-Project.html

[14] Z. Yu, I. Warren, and B. Macdonald, “Dynamic reconfiguration for robot software,” in CASE, 2006.

[15] D. Sanjay, M. C. Chinnaiah, T. S. Savithri, and P. R. Kumar, “A survey of reconfigurable service robots,” in RAINS, 2016.

[16] A. Kumar and D. S. Kumar, “Dynamic reconfiguration of robot software component in real time distributed system using clustering techniques,” Procedia Computer Science, vol. 125, pp. 754–761, 2018.

[17] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, “A study of variability models and languages in the systems software domain,” IEEE Transactions on Software Engineering, vol. 39, no. 12, pp. 1611–1640, 2013.

[18] “Featx project website: Tool, appendix, and documentation.” [Online]. Available: https://jgyimah.github.io/featx/

[19] A. Wąsowski and T. Berger, Domain-Specific Languages: Effective Modeling, Automation, and Reuse. Springer, 2023.

[20] D. Brugali and N. Hochgeschwender, “Software product line engineering for robotic perception systems,” International Journal of Semantic Computing, vol. 12, no. 01, pp. 89–107, 2018. [Online]. Available: https://doi.org/10.1142/S1793351X18400056

[21] ——, “Managing the functional variability of robotic perception systems,” in IRC, 2017.

[22] L. Gherardi and N. Hochgeschwender, “Rra: Models and tools for robotics run-time adaptation,” in IROS, 2015.

[23] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wasowski, “Feature-to-code mapping in two large product lines,” in SPLC, 2010, extended Abstract.

[24] T. Berger, S. She, K. Czarnecki, and A. Wasowski, “Feature-to-code mapping in two large product lines,” Department of Computer Science, University of Leipzig, Germany, Tech. Rep., 2010.

[25] T. Berger, R.-H. Pfeiffer, R. Tartler, S. Dienst, K. Czarnecki, A. Wąsowski, and S. She, “Variability mechanisms in software ecosystems,” Information and Software Technology, vol. 56, no. 11, pp. 1520–1535, 2014.

[26] L. Passos, J. Padilla, T. Berger, S. Apel, K. Czarnecki, and M. T. Valente, “Feature scattering in the large: A longitudinal study of linux kernel device drivers,” in MODULARITY, 2015.

[27] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, “Feature-oriented domain analysis (foda) feasibility study,” Carnegie Mellon University, Software Engineering Institute's Digital Library, Tech. Rep. CMU/SEI-90-TR-021, Nov 1990.

[28] J. Martinson, W. Mahmood, J. Gyimah, and T. Berger, “Fm-pro: A feature modeling process,” IEEE Transactions on Software Engineering, vol. 51, no. 1, pp. 262–282, 2025.

[29] L. Gherardi and D. Brugali, “Modeling and reusing robotic software architectures: The hyperflex toolchain,” in ICRA, 2014.

[30] T. Berger and P. Collet, “Usage scenarios for a common feature modeling language,” in SPLC, 2019.

[31] S. Zschaler, P. Sánchez, J. Santos, M. Alférez, A. Rashid, L. Fuentes, A. Moreira, J. Araújo, and U. Kulesza, “Vml*–a family of languages for variability management in software product lines,” in SLE, 2009.

[32] A. Steck and C. Schlegel, “Smarttcl: An execution language for conditional reactive task execution in a three layer architecture for service robots,” in DYROS, 2010.

[33] A. Koubaa, Robot Operating System (ROS), The Complete Reference (Volume 1), 2016.

[34] S. Macenski, A. Soragna, M. Carroll, and Z. Ge, “Impact of ros 2 node composition in robotic systems,” IEEE Robotics and Autonomous Letters, vol. 8, no. 7, pp. 3996–4003, 2023.

[35] F. M. Rico, A Concise Introduction to Robot Programming with ROS2. Chapman and Hall/CRC, 2022.

[36] 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, 2013.

[37] S. She and T. Berger, “Formal semantics of the kconfig language,” arXiv preprint arXiv:2209.04916, 2022.

[38] M. E. Cabello, R. Aquino, G. Ramírez, and A. Edwards, “A methodology to develop mobile robots based on a feature model,” in ISIEA, 2014.

[39] T. Buchmann, J. Baumgartl, D. Henrich, and B. Westfechtel, “Robots and their variability: a societal challenge and a potential solution,” in PLEASE, 2015.