MoTiVML: A Variability Modelling Language for Flexible and Customizable Robotic System Design

Jude Gyimah
Department of Computer Science and Engineering
Chalmers | University of Gothenburg
Gothenburg, Sweden
gusgyiju@student.gu.se

Sergio García
Department of Computer Science and Engineering
Chalmers | University of Gothenburg
Gothenburg, Sweden
sergio.garcia@gu.se

Thorsten Berger
Department of Computer Science and Engineering
Chalmers | University of Gothenburg
Gothenburg, Sweden
thorsten.berger@chalmers.se

Patrizio Pelliccione
Department of Computer Science and Engineering
Chalmers | University of Gothenburg
Gothenburg, Sweden
patrizio.pelliccione@gu.se

Abstract—Modern technological advancements have led to a growing demand for customized solutions, that can operate under non-deterministic conditions, with a high level of adaptability. In the wake of the current pandemic, industries have had to deal with unforeseen challenges that have risen in the form of labour, production and logistical deficits. These incidental challenges have negatively affected human-driven supply systems, through enforced safety regulations and human resource depletion. Undeniably, these challenges have had a resounding effect across the global service industry; and as such, these industrial challenges have also compelled organisations to devise alternative methods to alleviate them. One of such alternatives which has emerged and proven to be useful and feasible over time, is the implementation and deployment of service robots.

As a category of robots that render specialised services to humans, service robots are often designed to operate in highly heterogeneous environments, in collaboration with humans, or other robots. During operation, the effective completion of tasks by a given service robot may require a combination of specific sets of robotic capabilities. These inbuilt capabilities are more often than not driven by implementations known as robotic features.

Valid combinations of robotic features to fit varying contexts, have the propensity to give rise to some level of variability—i.e., the ability of a software artifact to be changed to match miscellaneous contexts, environments, or purposes, such as dynamic obstacles, noisy perceptions and random execution failures. This adaptation demand, presents the need for a possible strategy that enables roboticists to customize or configure robotic applications in correspondence to different operational scenarios. This has also provided the much needed incentive to have an effective mechanism that allows domain experts to plan, design, and implement variability flexibly.

As a possible solution to this need, we present a technique that implements variability via a feature's binding time and binding mode. The implementation of this novel technique is offered as an extension to the Self-adaptive dEcentralized Robotic Architecture known as SERA. SERA, which is a decentralized reference architecture that supports autonomous, heterogeneous, and collaborative robotic application development, lacks the innate capability to manage variability dynamically. This implies that, SERA and its ilk typically do not provide domain experts with the means and techniques required to manage variability effectively.

To solve this problem, we applied design science methods in studying the existing variabilities present in example systems, that can be categorised as service robots. This was necessary to implement a variability modelling framework, that provides a language together with mechanisms, capable of managing variability, based on a feature's binding time and mode.

In a domain where variability is typically performed in an ad-hoc manner, this open source solution will provide basic support for binding features together with verifiable evidence to prove the extensibility of reference architectures to support variability. That being said, this study is expected to ease extension complexities, increase binding flexibility, and provide mechanisms for managing variability in robotic systems.

Furthermore, this research provides evidence to back the claim that our proposed variability management technique is novel, realizable, useful in practice and has the capability of assessing valid configurations of robotic systems.

Index Terms—variability modelling, service robots, robotics software engineering, domain specific languages, feature reconfiguration, flexible feature binding

I. Introduction

Service robots—"*a type of robot mainly designed for personal or professional use that performs useful tasks for humans or equipment*" [1]—are gradually becoming an integral part of human existence. It is estimated that the global service robot industry will be valued at about 153 billion US dollars by the year 2030.[2] According to the International Service Robot Association, service robots can be classified as machines that sense, think, and act to extend human capabilities or increase human productivity [11]. This implies that service robots are often conceived, built, and used as intermediary solutions to assist humans in performing daunting and repetitive tasks.

As cyber-physical systems, service robots are often designed to operate in highly heterogeneous environments in collaboration with humans, other robots, or both. Practical

---

[1]https://www.iso.org/standard/75539.html
[2]https://www.alliedmarketresearch.com/service-robotics-market

implementations of such service robots include UVD's Model C [3] and PAL Robotics's TIAGo Base[4] disinfection range of robots, which were deployed in the fight against the spread of COVID-19, in areas such as shopping centres, airports, and hospitals across the globe.

In any given context of operation, a robot only needs a defined subset of core assets, otherwise referred to as features, that define the robot's capabilities. A feature can be classified as a logical unit of behaviour defined by a set of functional and non-functional requirements [5]. Depending on the usage context, features may be selected or deselected as part of a configuration to drive a given set of related capabilities.

For example UVD's disinfection range of robots, operate by moving at a sufficient speed in a 360-degree fashion, while emitting enough UV-C ultraviolet light onto relevant surfaces to eliminate viruses and bacteria. In so doing, a typical disinfection robot might make use of core assets such as navigation, obstacle detection, collision avoidance, disinfection, irradiation, and teleoperation when necessary.

Valid combinations of such core assets to match a robot's operational context gives rise to some level of variability within the robotic system. By definition, variability can be described as the ability of a core asset to adapt to usage in different product contexts that are within a product line scope [8]. One of the goals of managing variability is to create flexible, cost-effective and customisable core assets that are easy to build, extend and maintain.

Within a variability management context, all decisions on variability design are required to be communicated and documented for future use. As an important consequence, it has become necessary to have clear representations of variants, variation points, and mechanisms when realising variability. These said representations that manifest themselves in product lines are usually depicted with the aid of a reference architecture.

A reference architecture which is essentially a predefined architectural pattern, or set of patterns, is designed to capture high-level designs of product line applications [7]. These patterns that define reference architectures, could be partially or completely instantiated, designed and proven for use in specific business and technical contexts, together with supporting artefacts [17].

In this study, we focus on extending the reference architecture known as the Self-adaptive dEcentralized Robotic Architecture (SERA) [1]. As an architecture for decentralized, collaborative, and autonomous robots, SERA was conceptualized and designed by Sergio García et al to support human-robot collaborations, as well as the adaptation and coordination of single and multi-robot systems in a decentralized fashion.

SERA and others similar to it, typically do not support any standardised form of variability management. This is however not desirable in a complex, innovative and fast-paced domain such as robotics. Reason being that, robots are usually mandated to operate in a variety of environments, which are often unpredictable and human-populated. To navigate their way through such heterogeneous environments with ease, robots usually possess a selection of unique characteristics in the form of functional and non-functional capabilities, which are reusable in multiple operational scenarios.

Thus, we deemed it necessary to extend the capabilities of SERA's pre-existing architecture to include variability management, with a high level of flexibility and customizability of robotic product line entities, to improve practices currently present in robotics applications design and development.

This is particularly useful because, just like SERA, most known reference architectures, manage variability via ad-hoc mechanisms. Sergio García [41] together with a team of researchers in 2019, gathered EU, academic and industrial project experiences, that indicated that robotics industry stakeholders often relied on ad-hoc strategies like clone & own, configuration facilities of robotic frameworks and home grown configurators to manage variability.

Reference architectures commonly rely on such ad-hoc strategies because they do not inherently provide any means of controlling product entity bindings in a systematic way. In addition to this, they are also incapable of tracking and providing an overview of product entity commonalities and variabilities together with their dependencies consistently [23].

One of the fundamental concepts of adaptable software architectures is their ability to establish an overview of understanding, by systematically modelling the adaptation space using representations such as feature models, while having dedicated techniques to match such adaptations.

Our overall goal in this study is to provide roboticists with a variability modelling framework that has the means and techniques for planning, designing, and implementing variability. We achieve this by introducing techniques that realize variability. These techniques, typically referred to as variability mechanisms, comprise of a means for modelling variability such as feature models, and techniques that implement variation points within modelled features [23].

The solution we are offering in this study, contributes a novel technique for implementing variation points in robotic systems via a feature's binding time and binding mode. Time and mode bindings are implemented as an extension to feature models [2], where binding time is defined as either compile time or runtime, while binding mode may be static or dynamic. Usually, the semantics of such an implementation can be potentially complex, since valid feature reconfigurations are not only constrained by dependencies, but also rely on valid binding combinations that exist amongst dependent features.

In summary, our research work provides a variability modelling framework, in response to the following domain challenges:

1) *The lack of a standard variability modelling framework for robotic system design and development:* The lack of a variability modelling framework, integrated with standard contracts that determine the legal attributes of a feature together with a well defined internal order of

modelled features in an architectural landscape that can model different versions of such features and validate them. In addition to this, from a standard implementation standpoint, various ad-hoc mechanisms exist in different implementations where implementers have sought to enforce mechanisms in anticipation of possible context changes in a robot's operating environment. However, none of these solutions offer the required capabilities for flexible and customizable modelling, implementation and validation. Rather, such ad-hoc mechanisms tend to complicate the entire implementation due to their low level of reusability as well as their complexity of extension and maintenance.

2) *The lack of standard mechanisms and guidelines for implementing variability management techniques in robotic systems:* Aside the existence of a plethora of ad-hoc strategies, there is also a lack of useful guidelines that detail how to integrate and use a set of standard mechanisms that have been provided to design and implement variability in robotic configurations. The lack of a standardised set of mechanisms, can be attributed to the fact that, usually, implementations of variability mechanisms are included in robotic applications in the form of preemptive contingencies. For that matter, due to the non-deterministic nature of the contexts in which service robots are designed to operate in, these mechanisms fail to cater for all possible scenarios. And even in cases where they do, there are little to no guidelines to describe how they are implemented under the hood.

Additionally, to the best of our knowledge, ROS-based open-sourced applications in general, lack guidelines that (i) verify the specifications that define their capabilities and (ii) provide comprehensive coverage of the application and its platform, along with practical step-by-step usage of its functionalities.

In a domain like robotics which is relatively new and rapidly advancing, when it comes to implementing variability mechanisms, there seems to be no streamlined or platform-specific approach for doing so. In many cases, robotic application developers resort to undocumented methods of implementing and managing variability. Thus, these methods tend to be ad-hoc in nature, and as such, they are neither reusable nor maintainable.

3) *The lack of a variability management solution that addresses the complexity of engineering robotic systems:* A full-fledged robotic system is a complex system that consists of many modules. Whether single or multi-purpose, the versatility that a robotic system offers in different scenarios, requires the integration of heterogeneous modules. Many of these modules that need to be integrated are often developed by a diverse group of industry professionals (e.g., electrical engineers, perception experts, control theorists, software engineers). This perceivable diversity complicates the integration, customisation, and maintainability of packages in the robotics sphere. These complications translate into complexities when it comes

to collaboratively engineering robotic applications. Furthermore, due to the low-level of code reuse when it comes to implementing mechanisms, there is always the likelihood of having dispersed implementations of mechanisms which fulfil the same purpose, but are costly to maintain.

In realising a solution such as ours, a key requirement we sought to fulfil was to keep our solution as lightweight as possible without requiring new tooling; all the while maintaining a decent level of abstraction. Thus, we relied on design science approaches which were empirically guided by the following research questions.

- **RQ 1:** *What are the possible example instances of feature realizations with different binding times and modes in a ROS-based robotic system?*
  Example instances of feature realisations can be derived from example systems of service robots. Depending on their operational domain, these example systems of robots may possess different sets of features that can be bound at different times and in different modes.

- **RQ 2:** *How can a variability modelling language that allows features to be modelled together with their binding times and binding modes be designed?*
  A variability modelling language that allows features to be modelled together with their binding times and modes can be designed by initially clarifying the relevant aspects of the language domain, through domain analysis. Next, based on the key requirements gathered, the abstract syntax, concrete syntax and static semantics of the language can be derived and established.

- **RQ 3:** *Which mechanisms and guidelines can be used to implement features with different binding times and modes in a ROS-based robotic system?*
  Variability mechanisms are implementation techniques with the capability of realizing variability. In language designs of this nature, formalisms for modelling features can be provided based on the variant derivation concepts of feature models; along with implementations that demonstrate a time-mode variation mechanism. Supporting artefacts in the form of guidelines can also be provided along with implemented variability mechanisms to assist domain stakeholders in implementing and validating their modelled systems.

II. LITERATURE REVIEW

Assessing valid feature configurations based on feature binding times and modes can be tricky and somewhat cumbersome. For this reason, there is the need for a common method or approach to deriving configurations, that eliminates some of such complexities. As an antecedent of such a standardised configuration generation method, provision must also be made for established dependencies and constraints that exist between features.

A vast variety of key studies in the domain of robotics software engineering, software architectural design and variability management served as inspiration to this study. The most

prominent of these being SERA. SERA as a layered architecture that contains components that manage robotic system adaptations at different levels of abstraction was conceived by Sergio García et al. [1] to solve three distinct problems, namely: (i) the lack of architectural models and methods in the production of software for robotic systems, (ii) the absence of a common approach or strategy that might allow vendors to produce their own robots and deploy them within a team, and (iii) the lack of systematic support for adaptations of robot teams.

By communicating through well-defined interfaces, SERA has the ability to be implemented within a wide variety of projects. SERA's architecture can be realized using different middlewares and component frameworks related to robotics. This further emphasises the extensibility of its framework architecture in anticipation of future changes. That notwithstanding, SERA and architectures similar to it, seem to lack adaptation capabilities. More so, as an architecture for collaborative and autonomous systems, runtime adaptation in particular remains a top priority in SERA's operation due to the level of adaptation complexity that exists within the domain of collaborative autonomous systems. To cater for this capability gap, we have implemented an extension to SERA's architecture, in the form of an open source framework, that offers a domain specific language(DSL) implemented in Python[5] together with robotics driven variability management mechanisms delivered in C++[6].

A. Domain-Specific Languages

"A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain" [25].

Domain-Specific Languages (DSLs) are usually declarative. One of the key characteristic of DSLs is their focus on expressive power. They allow solutions to be expressed at the level of abstraction of the problem domain. For this reason, domain experts can understand, validate, modify, and often even develop DSL programs [25].

Developing advanced robotic systems can be challenging as expertise from multiple domains need to be integrated conceptually and technically. Through domain-specific modelling, robotic concepts and notations can be modelled descriptively. This raises the level of abstraction and results in models that are easier to understand and validate. Furthermore, DSLs increase the level of automation, e.g., through code generation, while bridging the gap between modelling and implementation [29]. Besides automation of software development through code generation, there are several added benefits of DSLs. These include analysis, optimization, and most importantly the inclusion of non-developer stakeholders into the process of software creation.

Literally hundreds of DSLs are in existence. However, only a subset of them are described in software engineering or programming language literature. Inspirational concepts that influenced our language's architectural design and implementation were sparsely drawn from existing DSLs, in our attempt to offer a user friendly yet intuitive language that possesses an optimum level of abstraction that fits the needs of multiple end users with varying skill sets. With dsl zoo [29] as our main reference point, annotated bibliographies of domain-specific languages in the area of robotics and automation technology were carefully reviewed. Examples of such DSLs whose bibliographies we considered, include Mauve[7], Robotml[8], LE[9], and eTaSL/eTC [10].

B. Runtime Adaptation

Runtime adaptation in modern day systems often involves the anticipation of some level of uncertainty. In most software systems today, handling uncertainty in advance is often infeasible and resource intensive. This signifies that there may be the need to deal with uncertainty, as and when the knowledge required in that particular scenario becomes available. At runtime for instance, robots need to manage large amounts of different execution variants that can neither be foreseen nor completely pre-programmed. Thus, these execution variants cannot be analysed and checked entirely at compile time.

For this reason, Hochgeschwender et al. [20] employed a model-driven approach which involves capturing domain knowledge explicitly in the form of domain models. Such models which are described by domain-specific languages are accessed by robots at runtime to take decisions for adaptation purposes. Granting robots access to software-related models at runtime implies that different notations and DSL formats are persistently stored; by composing various domain models; and queried over multiple domains at run time. Portions of the methods employed in Hochgeschwender's study to achieve adaptation, align with ours, in that, resources relevant to the operation of a given robot are modelled using DSLs and stored. Stored resource models are then accessed on demand at runtime, when binding information becomes available, for the sake of adapting the system to match changing environmental conditions.

As mentioned previously, models are fundamental to the functioning of robots. Steck et al. [21] provided an in-depth argument for why models and a model-centric approach to robotics software design and implementation is important. In their study, they reaffirmed the role which models play in providing a means to check the validity of desired configurations and parametrizations of robotic system components. A model-driven approach also feeds into the narrative of making an implementation generic enough to be used by multiple platforms. This supports our choice to utilise feature models

---

[5]https://www.python.org/
[6]https://isocpp.org/

[7]https://corlab.github.io/dslzoo/architectures-and-programming-subdomain.html#lesire2012mauve
[8]*#dhouib2012robotml
[9]*#gordillo1991high
[10]*#aertbelien2014etasl

as a graphical means of abstraction over robotic components, where modelled features mapped to such components, can be bound with respect to time and mode, and studied to realise runtime adaptation through the runtime reconfiguration of binding units extracted from valid combinations of features.

In our implementation, valid combinations are determined based on binding time and mode pairs attached to a given set of user selected features. Similarly, Pinto et al. [22] in implementing their middleware framework for context-aware applications, made provision to keep a record of policies that are fed with contextual information obtained by an agent (e.g., a robot) that interacts with the environment. Consequently, actions tied to functions are triggered when conditions surrounding a policy are satisfied. A policy set is dynamic in that it is updated in real time according the environmental changes experienced by the navigating agent.

This train of thought aligns well with design decisions we have made in our framework implementation where robotic features are assigned a specific set of bindings, to collectively serve as the underlying policy upon which modelled robots will base their adaptations.

C. Managing Adaptation Complexity

Adaptive systems may be defined as systems that exercise variability to cope with changing system requirements. Adaptive systems that support feature binding at runtime are sometimes referred to as dynamic Software Product Lines (DSPLs). DSPLs are usually built from coarse-grained components, which reduces the number of scenarios in which they can be possibly applied.

In a related study by Rosenmuller et al, [13], it was established that software product lines tend to increase in complexity when static and dynamic feature implementations are blended together. In such scenarios, it was discovered first and foremost that implementation complexities existed in software product lines. And as such, these internal complexities are often birthed, due to the effect that cross-cutting features have on each other.

With regards to DSPLs, Rosenmuller et al, as an extension of their previous studies on dynamic feature binding, [32] conceptualised a feature-based adaptation mechanism that reduces the effort of computing optimal configurations at runtime.

This was done by generating a DSPL from an SPL, through static selection of features required for dynamic binding, and then generating a set of dynamic binding units from the select features. In order to support runtime adaptation of programs effectively, Rosenmuller, Siegmund, Apel and Saake integrated a customizable framework, called FeatureAce[11]. By including FeatureAce into a generated DSPL, they were able to harness the capability of composing features and modifying configurations at runtime.

Learning points from these studies bear semblance to our own study in many ways. First off, Rosenmuller et al [32] acknowledged the complexity that exists within software product lines and as such understood that variability can only

be managed with an effective mechanism that can keep up with changing requirements, without having to deal with an overwhelming amount of overhead that comes with it; be it functional or compositional.

To remedy this problem they chose to use DSPLs composed out of SPLs that are combined into dynamic binding units. In line with that, we on the other hand, decided to take the approach of leveraging robotics-based variability mechanisms that implement binding time and binding mode as a means of adapting our configurations. The difference in approach here lies in the fact that our methods consist of robust platform and language specific implementations for static, dynamic, early and late feature binding. In the context of this study, our approach does not focus so much on the quality of dynamic binding units. Rather we place a significant amount of emphasis on the feasibility and usability of our chosen mechanisms in conjunction with our variability modelling language.

D. Feature Modelling Languages

Feature modelling; which is—"*a concept within Feature-Oriented Domain Analysis (FODA) [38]; which presents an abstract means of representing commonalities and variabilities that exist between product line variants*", has given rise to the design and implementation of multiple feature modelling languages both in the software engineering and robotics software engineering domain. With the current set of feature modelling languages that are available, Clafer,[12] Kconfig,[13] and CDL[14] appear to be quite popular amongst industry professionals. Clafer is one of the most expressive feature modeling languages that unifies both feature and class modelling. The notion of features and classes are unified within the Clafer architecture. Clafer fundamentally offers types, constraints, and attributes; a set of properties that can be identified in many languages as well. It also supports multi-level modelling and has well-specified semantics, coupled with rich tooling that supports instance generation, configuration, and visualization. As a textual language, Clafer has one of the simplest, and by far the most intuitive syntax in industry today [13]. The language is predominantly built on first order logic with quantifiers over basic entities combined with linear temporal logic [14].

Kconfig and *CDL* are also examples of popular languages that have the capability to describe the variability of systems expressively. Even though *Kconfig* and *CDL* share similar design concepts, it is important to note that they were developed independently from each other, as well as other feature modelling languages with research origins [37]. Kconfig and CDL are two of the most successful languages, primarily used in systems software engineering [13].

In a 2019 study pertaining to usage scenarios for a common feature modelling language conducted by Berger et al. [13], it was established that a feature modelling language that is

---

[11]https://sourceforge.net/projects/featurecpp/

[12]https://www.clafer.org/
[13]https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html
[14]http://www.cse.chalmers.se/~bergert/paper/cdl_semantics.pdf

intuitive, simple, and also expressive enough to cover a fair range of usage scenarios, must make provision for a base set of usage scenarios that include Exchange, Storage, Domain Modelling, Teaching and Learning, Mapping to implementation, Model generation, Benchmarking, and Analyses. This set of scenarios were derived from a larger set of general usage scenarios elicited from researchers at a meeting during the Software Product Line Conference (SPLC) in September 2018, held in Gothenburg.

Despite the fact that most of these languages share similarities in some of their fundamental concepts, they also come with several limitations. These perceivable pitfalls have influenced our decision to propose yet another feature modelling language to address them. In comparison to the others, our proposed language has been evaluated based on eight characteristic properties. These characteristics confirm the relevance and novelty of our feature-modelling language with respect to others in the robotics domain. The set of characteristic properties which we considered in our language design include: (i) C1: Expressiveness (ii) C2: Coverage of a wide range of usage scenarios (iii) C3: ROS compatibility (iv) C4: Ability to describe variability (v) C5: Abstract syntax support for variability modelling based on binding time and binding mode (vi) C6: Ability to enforce and evaluate modelling constraints. (vii) C7: Ability to implement time and mode binding as a variation mechanism. (viii) C8: Provision of guidelines and mechanisms that implement variability.

Table I shows how our proposed feature modelling language compares to other existing languages with respect to the chosen characteristics stated above.

TABLE I
CHARACTERISTIC COMPARISON OF FEATURE-MODELLING LANGUAGES

Language	C1	C2	C3	C4	C5	C6	C7	C8
PyFML	•	•				•		
HyperFlex	•	•	•	•		•		
TVL	•	•		•		•		
Clafer	•	•		•		•		
KConfig	•	•		•		•		•
CDL	•	•		•				
MoTiVML	•	•	•	•	•	•	•	•

From Table I we can deduce that the ambition of our proposed modelling language differs from other existing languages significantly. In that, we are providing a solution that is not only expressive and capable of describing variability, but also a solution that covers a wide range of usage scenarios by providing implementation mechanisms and guidelines. We are also offering a solution that is compatible with the robot operating system and possesses an abstract syntax that supports variability management based on feature binding time and binding mode. Finally our variability modelling language enforces binding time and binding mode level constraints that can be evaluated for well-formedness and type correctness.

E. Variability-modelling Frameworks, Middleware, and Toolchains

Over the years many component-based frameworks and toolkits that are robotics inclined have emerged. These frameworks possess mechanisms for real-time execution, synchronous and asynchronous communication, control and data-flow management and system configuration.

When it comes to variability modelling frameworks, ROS, Orocos and SCA stand out in terms of popularity, scale and ecosystem support. Fundamentally, these frameworks have the following features in common i.e., (i) They provide a component model, which defines a set of architectural elements (e.g. components, interfaces, connection) and the rules for composing them in order to build a component based system [33]. (ii) They provide a runtime infrastructure, which is in charge of instantiating, connecting, configuring and activating system components. ROS and Orocos cater more to the robotics domain, while SCA leans more towards the idea of a service oriented architecture development. However, as the years have rolled by, ROS has emerged as the de facto standard when it comes to robotic application development for several reasons.

By design, ROS favours a software development approach that lays emphasis on designing components which implement common robotic functionalities. The strength of this approach lies in its capability to develop a large variety of control systems by composing reusable software building blocks in different ways. The drawback of this approach is the lack of support for the reuse of effective solutions to recurrent architectural design problems. Consequently, application developers and system integrators have always had to solve such difficult architectural design problems from scratch. One of ROS's standout challenges can be observed when it comes to selecting, integrating, and configuring coherent sets of components that provide required functionalities, taking into account their mutual dependencies and architectural mismatches [34]. As a potential remedy for this, *Hyperflex* was developed.

HyperFlex is a toolchain designed for developing software product lines for autonomous robots based on robotic component frameworks, such as ROS and Orocos. As a collection of Eclipse plugins implemented by means of the Eclipse Modeling Project, HyperFlex allows users to explicitly model the architecture of functional systems in terms of components, interfaces, connectors, and components wiring. Models of functional systems can be reused as building blocks and then hierarchically composed in order to build more complex systems and applications [34]. HyperFlex uses feature model formalisms to symbolically represent the variability of a system together with the constraints that exist between variation points and variants that limit the set of valid configurations [34].

Comparatively, the *Hyperflex* solution in some ways, has similarities to our proposed solution, in terms of its variability management capabilities and ROS base. The difference however lies in the variation mechanism being implemented. While

Hyperflex is only concerned with the general attributes of a feature i.e. optional and mandatory, we go a step further to include and decouple binding time and binding mode attributes as our variation mechanism. Another standout difference that is noteworthy is the fact that *Hyperflex* is presented as an eclipse plugin as opposed to our solution that is offered as a platform independent open-source framework.

In a profound study conducted by Lotz et al [39], functional and non-functional behaviour were modelled in an attempt to manage runtime variability in robotic systems. With the aid of two DSLs; namely SmartTCL[15] and VML [40], their goal was to separate concerns when modelling variability both in operation and quality by having a dedicated DSL that handles action coordination modelling and another that deals with task fulfilment quality modelling. Similar to ours, Lotz's approach focuses on improving the general ad-hoc method of managing variability in robotic systems due to its many pitfalls. However, the distinct divergence between the two, lies in their focus on flexibility and quality as opposed to our flexibility and customizability based approach.

Comprehensively, our variability management solution can be considered to be a framework that amalgamates a variability modelling language and variability mechanisms. Both of these implemented components ensure feature model instantiation, configuration, validation and feature source-code integration.

F. Flexible Feature Binding

A broad spectrum of binding techniques exist within Software Product Line Engineering. These binding techniques possess unique characteristics that set them apart from each other. Two of such techniques are time and mode binding. Time in the context of compile time or runtime and mode in terms of how static or dynamic a feature is allowed to be.

By strategically combining both techniques, feature flexibility and customizability can be harnessed to enhance the design of product line variants tremendously.

1) Time - Mode Binding Technique: In a time - mode binding scenario, product line variants can be generated by composing configurations of implemented features based on static, dynamic, early and late feature attributes. A static feature is one that is bound into a program before load time [2]. This implies that once a static feature is bound within a program, it cannot be rebound without rebuilding the entire program. That is, an activated feature cannot be deactivated any more and vice versa until the program containing that bound feature has stopped running. Examples of mechanisms that realize static binding include: the C/C++ preprocessor[16] and antenna[17] [2]. Implementing this exclusively introduces a functional overhead where some features may have the tendency of being loaded but never used.

A dynamic binding mode on the other hand implies that variation points bound dynamically in a software product line

can be altered on demand. However, this usually generates some overhead with regards to performance and resource consumption, through an increase in execution time and binary size respectively. Examples of mechanisms that realize dynamic binding include: FeatureIDE's runtime parameters[18] and ROS pluginlib[19] from ROS's ecosystem [2].

Static and dynamic binding concepts are not new when it comes to software product lines. These binding forms provide a number of unique benefits in software product line design and implementation.

According to a study conducted by Marko Rosenmüller et al, utilising static or dynamic binding exclusively, often restricts the applicability of product line variants generated in a given instance [2]. For example, static binding cannot be used when required features in a variant are not available or known at deployment time; as is the case for third-party extensions.

To decrease some of such restrictions, different approaches for combining static and dynamic binding have been proposed in multiple studies with the goal of retaining customizability and flexibility with a fair amount of runtime overhead. Additionally, the practice of supporting different binding times based on the same implementation mechanism simplifies Software Product Line development and maintenance considerably [2]. For instance, in our proposed technique, at binding time, an early binding classification implies that a feature is programmed to be loaded or unloaded at compile time while a late binding classification implies that a feature only allows runtime loading or unloading.

For both time and mode binding, our proposed variability modelling technique supports an implicit "Any" binding type. In that, the binding time attribute of a feature can reference both an early and a late binding time state. Likewise in the case of binding mode attributes, an "Any" binding type implies that a feature can be both static and dynamic.

Within the feature modelling capability of our language, the variability of any given robotic feature can be modelled and configured within the following binding scope:

- **Static Early:** Static Early binding defines a feature that cannot be unloaded when loaded and vice versa, is loaded (or unloaded) at compile time.
- **Static Late:** Static Late implies that a feature that cannot be unloaded when loaded and vice versa, is loaded (or unloaded) at runtime.
- **Static Any:** This indicates that based on feature's mode, the feature cannot be unloaded when loaded and vice versa.
- **Dynamic Early:** Dynamic Early implies that a feature that can be loaded and unloaded several times is bound at compile time.
- **Dynamic Late:** Dynamic Late features can be loaded and unloaded several times is bound at runtime time.

---

[15]https://wiki.servicerobotik-ulm.de/about-smartsoft:robotic-behavior: smarttcl

[16]https://mcpp.sourceforge.net/

[17]http://antenna.sourceforge.net

[18]https://featureide.github.io

[19]http://wiki.ros.org/pluginlib

- **Dynamic Any:** This implies that a feature can be loaded and unloaded multiple times but has the potential to be bound either at compile time or runtime.
- **Any Early:** Dynamic Early binding refers to a feature that can be loaded and unloaded several times is bound at compile time.
- **Any Late:** Dynamic Late implies that a given feature that can be loaded and unloaded several times is bound at runtime time.
- **Any Any:** This refers to a situation where a feature can be bound either statically or dynamically at compile time or runtime.

The set of binding pairs indicated above can be generated through an expansion of all possible binding time values combined with all possible binding mode values assigned to a feature. The resultant attribute pairs made up of nine combinations of both implicit and explicit binding types, provide a high-level overview of all possible binding states that are likely to exist within the configuration space of a given mobile service robot example.

III. RESEARCH METHOD

Our research method of choice for this study is design science. We chose the design science paradigm mainly due to its deep roots in engineering science. In that, it generally offers a problem-solving approach that seeks to create innovations that define the ideas, practices, technical capabilities, and products through which analysis, design, implementation, management, and information system usage can be effectively and efficiently accomplished [10]. Figure 1 provides a layered and well defined overview of the empirical steps we took to create artefacts that rely on existing theories that are applied, tested, modified, and extended through experience, creativity, intuition, and problem solving, as part of our study.

Our language design and its implementation methods, can be broken down into a number of distinct phases. These include domain analysis, abstract syntax definition, concrete syntax derivation, static semantics implementation, system development and evaluation. These phases were performed iteratively in a manner that ensured that outputs obtained from each phase, fulfilled a specific requirement that can be directly traced back to our research questions. In our domain analysis phase, we clarified the core ideas of our language from sources such as domain experts and related documentation. Based on our findings, we conceived our abstract syntax and expressed it with meta-models. Subsequently, we established the concrete syntax of our language in both graphical and textual formats. Our resultant concrete syntax, together with the static semantics of our configuration logic were transformed into a ROS compatible implementation. Finally we evaluated our implementation based on its novelty, correctness and realizability.

A. Performing Domain Analysis

Prior to developing formalisms for our language, we identified key concepts that are relevant to the domain in which our language would be used i.e. variability management in the field of robotics. We also identified and clarified details such as the purpose of our language, target stakeholders, concepts and relations. We identified all such relations between our gathered domain concepts by studying the properties of example systems that embody our domain concepts and relationships.

B. Implementation

In our language implementation step, we defined the capabilities of our language based on prior knowledge from our domain analysis phase and then conceived both the graphical and textual forms of selected example systems, to emphasise the syntactic structure and tokens that make up our language. Graphically, we used feature models to define and model features of the same robot examples from previous activities. And finally, on an abstract level, we used class diagrams to describe the capabilities present within our language.

1) Defining the Abstract Syntax: In creating the abstract syntax of our language, we formalised prior knowledge gathered in our domain analysis step into models. These models, otherwise known as meta-models were then refined in multiple iterations to reflect the requirements that form the capabilities of our variability-modelling language with respect to the problem space. We followed the following concrete steps to realise the abstract syntax of our language: (i) Create a single decomposition of a class diagram together with compositional relationships. (ii) Verify the manner in which concepts are classified, and classes are organized in a hierarchy. (iii) Concretize concept relationships if need be. If these relationships between concepts posses properties, then they need to be transformed into classes. (iv) Remove redundancies by finding and removing multiple classes that have the same property. (v) Evaluate resultant abstract syntax based on how accurately it reflects the problem and how well the concepts of design were kept within scope.

2) Defining the Concrete Syntax: The concrete syntax of our language is expressed both graphically and textually through feature models and JSON notation respectively. A major requirement of our implementation is to give potential users the expressive capability to create their own custom programs that will be in the form of a collection of modelled robotic features. In that, end users of our domain specific language have the ability to implement their own feature models of example robots that are adaptable in varying contexts. These feature models were conceptualised and implemented with FeatureIDE[20] and then further translated into collections of JSON objects.

C. Establishing the Static Semantics

After defining both the concrete and abstract syntax of our language, we proceeded to establish the static semantics of our language. Static semantics in this context, refers to the underlying rules that determine the validity of model variants

[20]https://featureide.github.io/

Fig. 1. Methodology Overview
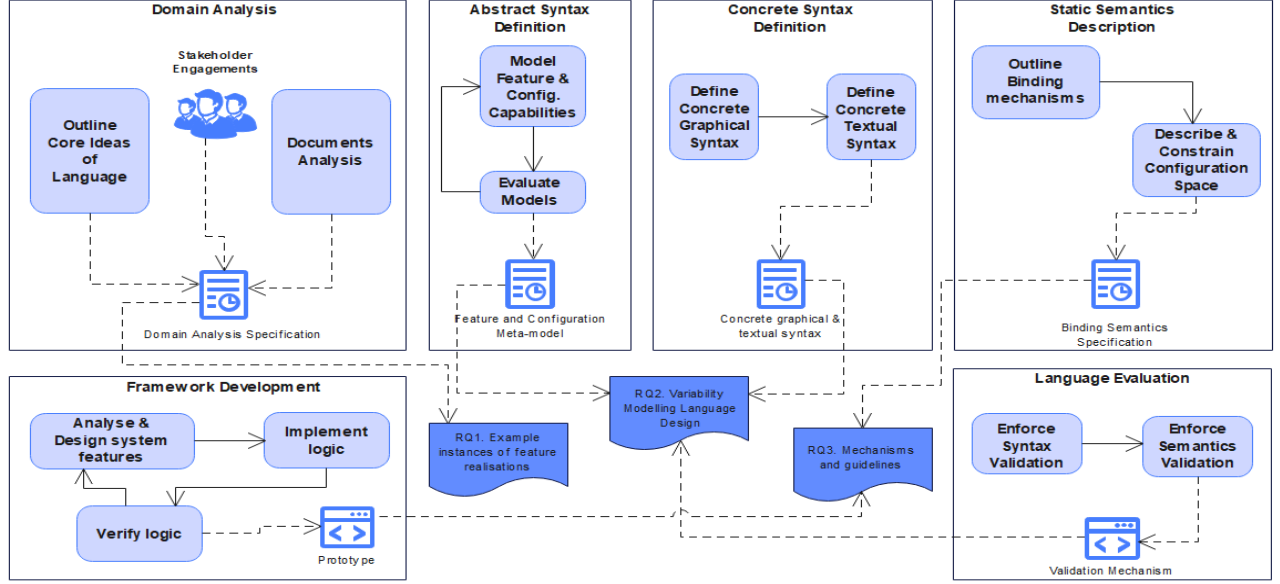
that can be created with our language. Key aspects of the static semantics which we clarified and established include:

1) Outlining mechanisms that realise time and mode binding.
2) Describing the configuration space that defines the configurability of our model instances.
3) Constraining the configuration space to limit the scope of adaptability.

D. Development

In our development phase, we utilised all the knowledge gathered from our domain analysis, concrete and abstract syntax definition and static semantics description, to implement our domain specific language as part of our framework. Our development approach can be summarized into the following steps:

1) Gather system requirements by establishing the purpose of the language with regards to its functionality.
2) Analyse requirements by transforming its specified textual functionalities into concrete functional and non-functional requirements.
3) Establish design decisions such as the general architecture of the system, dependency management, choice of technologies and their trade-offs, as well as implementation algorithms.
4) Realise language designs with Python based algorithms and data structures. Likewise interfaces for integrating encapsulated feature implementations need to be realised in C++[21].
5) Test implementation by evaluating the interactions of all the various integrated system components.

[21]https://isocpp.org/

E. Language Evaluation Overview

Evaluating our language means to assess its implementation from three main perspectives. These perspectives include (i) Correctness, (ii) Realizability and (iii) Novelty.

- **Correctness:** The validity of models and configurations created with our language along with the implied effect of a model's feature constraints are closely evaluated by both a schema and a constraint checker. These operate by detecting syntax, semantic and constraint violations, and report them to the user accordingly.
- **Realizability:** Proof of realizability indicates that our binding technique which forms the core of our implemented variability mechanisms is feasible. This is quite relevant, given that modelling variability with feature binding times and modes is yet to be applied in the robotics domain.
- **Novelty:** The novelty of our work is assessed along the lines of the innovative and conceptual uniqueness that defines the binding technique of our variability mechanism. By comparatively positioning our study with other studies from the robotics domain, we would be in a position to back our claim of novelty through research.

By realising our variability modelling language to implement binding time and binding mode mechanisms in a manner that fulfils our research objectives, we were able to automatically provide proof of the realizability and novelty of our binding technique. In addition to this, a comprehensive evaluation plan was applied to test for type and syntax correctness.

F. Correctness Evaluation

As a functional requirement of our language, end users have the means to build programs by modelling robotic

features, defining configuration bindings for all modelled features, and then applying those configurations to multiple scenarios through implemented mechanisms. Configurations are typically an amalgamation of features, relationships and constraints. To be able to assess the validity of user-defined configurations described with our language, we have implemented mechanisms that can be used to evaluate the well-formedness of both models and configurations. A high level description of our framework evaluation plan can be summarised as follows:

1) Realise mock-ups of example robotic systems.
2) Define configuration bindings for each mock-up example.
3) Encapsulate and plug in source code implementations of defined mock-up features.
4) Compile and validate the syntax and semantics of models and configurations.
5) Execute and demonstrate model variability.

For stakeholders to be able to verify the correctness of their models, all models and configurations defined with our modelling language must be validated on multiple levels of correctness i.e. syntax and semantic accuracy.

Syntax Validation: To effectively validate our language syntax, we evaluate the schemas of both model and configuration objects. Schema validation in this sense refers to token and token state validity.

Semantics Validation: In our attempt to validate the semantic implications of our language, we assess the correctness of configuration bindings in relation to the corresponding binding constraints specified. We also judge the correctness of each configuration binding pair with respect to its implied effect on other features in a given configuration space.

IV. RESULTS

A. Architectural Overview

Our language, otherwise known as MoTiVML, is a lightweight variability modelling language which is structurally based on the Javascript Object Notation (JSON)[22], and implemented as a domain specific language of the Python programming language. Architecturally, it consists of three modular components. These include a syntax manager, a configuration manager and a source code implementation manager. As shown in Figure 2, all these components are collectively offered as an open-sourced ROS-compatible framework.

In there, the syntax manager can be seen to contain all implementations related to language tokens, as well as the language's lexical and syntactical schemas. Tokens in this sense, represent the legal keywords or lexemes that are permissible in our language. There are two main objects that define MoTiVML i.e. that of a feature and that of a configuration. Feature objects modelled according to our unique feature schema design are nested within a model collection. All such features possess a set of constraints embedded within their schema definitions. Likewise, uniquely identifiable feature configurations,

---

[22]https://www.json.org/json-en.html

modelled per our configuration schema specification, reside within a *properties* collection as *config* documents. For each product line design, a decoupled set of tokenized model and configuration collections can be associated with it. Listings 1 and 2 show examples of a feature and configuration object respectively.

```json
{
    "id": "F0",
    "name": "ComponentControl",
        "constraints": {
            "featuresIncluded": [],
            "featuresExcluded": [],
            "bindingTimeAllowed":"Early",
            "bindingModeAllowed":"Static"},
    "group": "OR",
    "isMandatory": true
}
```

Listing 1: Feature Object Schema

```json
{
    "id": "F0",
    "props": {
        "mode": "Static",
        "time": "Early"
    }
}
```

Listing 2: Configuration Object Schema

In general, variability management provides two main capabilities i.e. feature modelling and variation points definition. The configuration manager component of our language library is made up of two parts. Namely, a binding semantics interpreter and a constraint checker. The binding semantics interpreter assimilates and displays the result of the interaction found between the assigned binding pairs of any collection of features. The constraint checker on the other hand parses the entire model definition of a given example system, to evaluate it for constraint violations in an attempt to validate user-defined configurations with respect to the configuration space. Thus, the constraint checker enforces configuration rules that define the adaptability of all configurations based on time-mode bindings. The source code implementation manager consists of a plug-in management interface which provides an integration point where end users can include feature source code into their modelled configurations by wrapping them as self-contained plug-ins.

B. Domain Analysis

In our language design, we clarified core concepts that are key to the domain context of our language. Furthermore, we identified design concepts and relationships that determine the type of programs which domain stakeholders would be capable of building with our language. All these concepts and relationships were then formalized into a capability model, that was iteratively refined to develop our language's abstract syntax.

During our domain analysis phase, we identified five standard questions which make up the core ideas and knowledge

Fig. 2. Architectural Overview of Framework Components
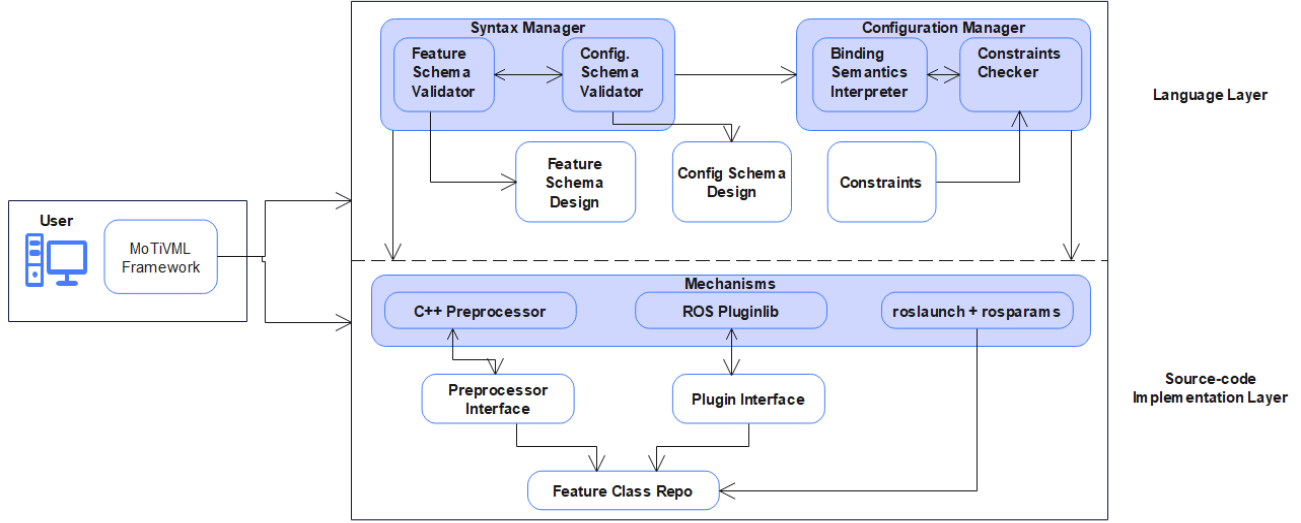
base that represents our language and its use. These questions can be categorised as follows:

- **Purpose:** *What is the purpose of this language?*
  To provide roboticists with the means and techniques for planning, designing, and implementing variability through mechanisms, that implement binding time and binding mode techniques.

- **Stakeholders:** *Who are the key stakeholders and the intended users of the language?*
  Our language targets a very diverse group of skilled professionals in the robotics domain who can be classified as end users. These stakeholders/end users include (i) **Operators:** End users with minimal training on the usage of our framework, but have the ability to operate robotic applications with pre-configured configurations. (ii) **Developers:** End users in charge of developing, implementing and integrating new features into our framework. (iii) **System Engineers:** End users in charge of adapting the framework to match the specific requirements of a customer.

- **Concepts:** *What are the key domain concepts that targeted stakeholders care about?*
  From an end user perspective, concepts such as simplified feature modelling techniques, standardized configuration management techniques and flexible yet customizable feature binding based on time and mode are key.

- **Relations:** *How are domain concepts related, and what are their relevant properties?*
  Properties that define the core user concepts of our language include:
  – Every instance of a feature class is selected by default.
  – All features in a model are defined as mandatory by default upon instantiation.
  – A parent feature may have zero or more child features.
  – A modelled feature may have zero or more groups but

a group must have two or more features to exist.
  – A grouped set of features may belong to an OR or XOR group.
  – A feature must have a binding time property which is set to *Early* by default.
  – A feature must have a binding mode property which is set to *Static* by default.
  – A modelled feature's binding time property can only exist in three states. i.e. *Early, Late, Any*.
  – A modelled feature's binding mode property can only exist in three states. i.e. *Static, Dynamic, Any*.

- **Examples:** *What examples of language instances are available?*
  To the best of our knowledge, no language designed to model variability in robotic systems, using binding time and binding mode exists. Our language is built on novel principles crafted to effectively manage variability in ROS-based robotic systems.

C. Implementation

Configuration capabilities within our architectural implementation pattern are decoupled from that of a feature. This is to primarily make features easier to manage, reduce duplication of non-configurable attributes i.e. eliminate redundancy, and to further separate concerns. This separation is required to show that both components have clearly defined responsibilities. By detaching features from configurations, end users can model and configure their robots without having too deal with extraneous information. Furthermore, each set of decoupled configuration attributes references an existing feature by a unique *ID* attribute.

1) Developing Mock-up Examples: An important use case of our language is that, it can be aid in developing mock-up examples that demonstrate our language's expressiveness. Within the context of this study, examples refer to modelled

features of robot product lines. Figure 3 and 4 show graphical representations in the form of feature models, of selected example robots. Each feature present in these models, can be expressed textually with tokens from our language as shown in Listings 3 and 4. Inspiration for these examples were drawn from general utility and disinfection robots such as TIAGo and UVD Model C.

Furthermore, in our model refinement step which aims to extend our mock-up examples against our requirements, we analysed the requirements associated with our language, justified them, and then proceeded to provide evidence of said requirements that have been fulfilled.

```json
{
    "id": "root_feature",
    "name": "simpleBot",
    "group": "",
    "isMandatory": true,
    "sub": [
            {"id": "F0",
            "name": "ComponentControl",
            "constraints": {
              "featuresIncluded": [],
              "featuresExcluded": [],
              "bindingTimeAllowed":"Early",
              "bindingModeAllowed":"Static"},
            "group": "OR",
            "isMandatory": true
            "sub":[
                {
                "id": "F1",
                "name": "..."
                }
            ]
        }
    ]
}
```

Listing 3: Textual Sample of a Set of Features

All features present in our feature model specification posses binding time and binding mode configurations which can be used by our language to manage variability within a modelled instance of a robotic system. As indicated in our architectural design, configuration collections are separated from feature collections for the sole purpose of decluttering model documents and abstracting attributes that have distinct responsibilities from each other. Listing 4 captures the format of a sample representation of the concrete textual syntax of feature configurations present in a given model. This consists of a collection of feature *ids* together with their *time* and *mode* binding properties, derived from a separate collection of feature documents nested in the same order as their corresponding graphical feature model.

2) Extending Mock-up Examples Against Requirements:
Based on our understanding of the domain from previous steps, we extended our mock-up examples in Figure 3 and 4 against our set of already established language requirements. These requirements are as follows:

1) R1: End users can instantiate robotic models.
2) R2: End users can create instances of features within models.
3) R3: End users can create decoupled model configurations.

```json
{
    "properties": [
            {
            "id": "F0",
            "props": {
                "mode": "Static",
                "time": "Early"
            }
        },
            {
            "id": "F1",
            "props": {
                "mode": "Static",
                "time": "Early"
            }
        }
    ]
}
```

Listing 4: Textual Sample of a Set of Feature Configurations

4) R4: End users can define and link configurations to feature instances.
5) R5: End users can evaluate models both syntactically and semantically.

Using the concrete textual syntax of our modelling language highlighted in Listing 3 and 4 as a reference point, we identified and generated example syntax representations that fulfil all the requirements listed above. Table II captures a mapping of each stated requirement along with its corresponding example syntax representation, that justifies and fulfils that same requirement.

TABLE II
EXTENDED MOCK-UP EXAMPLES AGAINST REQUIREMENTS

Requirement	Example Syntax
R1	$\{"id":"root\_feature","name":"simpleBot",$ $"group":"","isMandatory":true,"sub":[]\}$
R2	$\{"id":"F1","name":"Localization",$ $"constraints":\{"featuresIncluded":[],$ $"featuresExcluded":[],$ $"bindingTimeAllowed":"Early",$ $"bindingModeAllowed":"Static"\},$ $"group":"","isMandatory":true\}$
R3	$\{"id":"F1","props":\{$ $"mode":"Static","time":"Early"\}\}$
R4	$"props":\{"mode":"Static","time":"Early"\}\}$
R5	Using the schema and constraint checker, syntax, grammar and semantics of a model can be evaluated

3) Identifying Language Tokens: The basic compositional unit of our language is an attribute. From a lexical standpoint, these attributes are derived from a defined set of legal or permissible keywords referred to as tokens. These tokens form part of a key-value pair production. Tokens can be identified and extracted from the key-value pair attributes that make up feature and configuration collections in our model. Attribute keys serve as meta-data for the data held by their
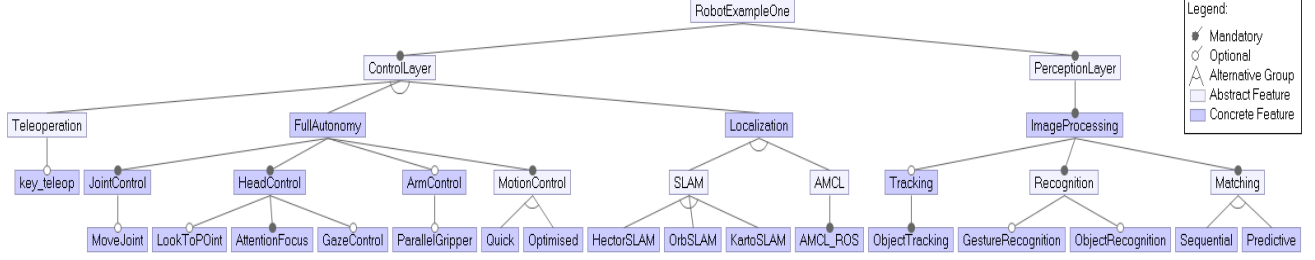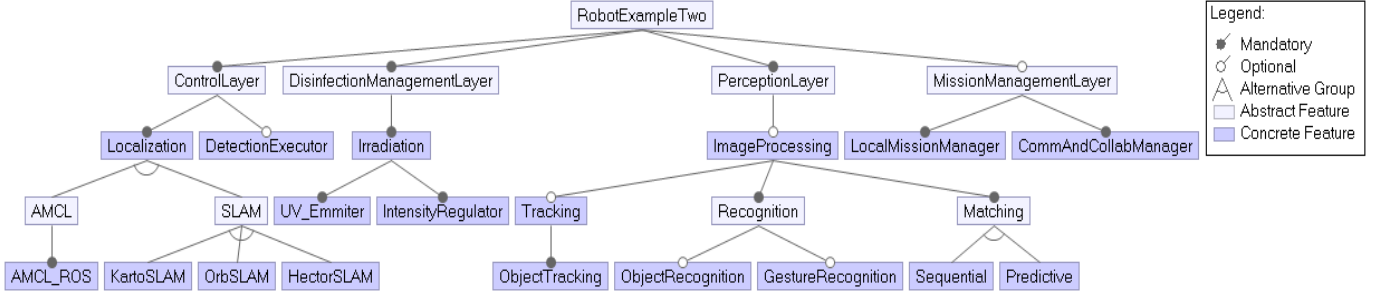
Fig. 3. Modelled Example One

Fig. 4. Modelled Example Two

corresponding values. Table III shows all identified tokens together with the regular expressions that define their legal or valid lexical structures.

In Table III, for each identified token in our language, we translated a valid representation of it into a regular expression to demonstrate which valid forms of these token can exist in our language. Syntactically speaking, modelled features and their configurations, encapsulate key-value paired attributes as self-contained objects. Lastly, Table III also captures a list of generated terminal symbols that can be used in grammar production.

4) Specifying Language Terminals: Our language's syntax can be described as a production consisting of left-hand-side and right-hand-side attributes. For each attribute, the left-hand-side consists of non-terminals that serve as language meta-data. On the right-hand-side however, we can identify singular or grouped forms of terminal and non-terminal symbols in the form of data. Left-hand-sided and the right-hand-sided attributes are separated by a colon (:) and each model attribute in turn is separated from the next by a comma(,). Table III captures all meta-data and data tokens that define our language.

5) Identifying Syntactic Categories Within Our Language: The modelling capability of our language provides domain stakeholders with a means of instantiating a feature model, as well as the possibility of defining and configuring features. Within our language's concrete syntax, one implicit and two explicit syntactic categories can be identified. These categorizations include:

1) *Feature Description Block:* This is made up of all feature attributes that are concerned with describing the general purpose of a feature. Although this category is not explicitly expressed in the textual formalism of a feature's

TABLE III
LANGUAGE TOKENS

Token	Regular Expression	Terminal Symbol
id	'id'	Id
name	'name'	Name
featuresIncluded	'featuresIncluded'	FeaturesIncluded
featuresExcluded	'featuresExcluded'	FeaturesExcluded
bindingTimeAllowed	'bindingTimeAllowed'	BindingTimeAllowed
bindingModeAllowed	'bindingModeAllowed'	BindingModeAllowed
group	'group'	Group
isMandatory	'isMandatory'	IsMandatory
time	'time'	Time
mode	'mode'	Mode
"Early"	'Early'	Early
"Late"	'Late'	Late
"Static"	'Static'	Static
"Dynamic"	'Dynamic'	Dynamic
"Any"	'Any'	Any
"true"	'true'	True
"false"	'false'	False
"OR"	'OR'	OR
"XOR"	'XOR'	XOR
alphaValue	$(a-zA-Z)^{+}$	alphaVal
boolValue	$[true|false]$	boolVal

schema, it can still be defined as a set of individual feature attributes which intuitively provide descriptive contexts to the functionality and purpose of a feature.

2) *Constraints Block:* This is represented as a nested block of feature attributes that define the type constraints applicable to a feature as well as the set of possible constraint values that are valid for that feature.

3) *Binding Properties (props) Block:* For every feature, there exists a corresponding configuration. Within the

configuration of a feature exists a binding property block definition. The purpose of this binding property block is to allow domain stakeholders to configure binding time and binding mode attributes for each feature they define. These binding time and binding mode attribute definitions serve as the fundamental variation mechanism for each feature model.

6) Specifying Grammar Rules: By combining the terminals defined in Table III with the syntactic categories identified in the previous section, we were able derive grammar productions for our language. The grammar rules present in our language for all three syntactic categories can be expressed as follows:

featureDescriptionBlock $\longrightarrow$ '{' 'id' ':' alphaVal, 'name' ':' alphaVal, 'group' ':' alphaVal, 'isMandatory' ':' boolVal '}'

constraintsDefinitionBlock $\longrightarrow$ constraints: '{' 'featuresIncluded' ':' '[' Id, Id, ... ']', 'featuresExcluded' ':' '[' Id, Id, ... ']', 'bindingTimeAllowed' ':' alphaVal, 'bindingModeAllowed' ':' alphaVal '}'

bindingPropertiesBlock $\longrightarrow$ '{' 'time' ':' alphaVal, 'mode' ':' alphaVal '}'

7) Abstract Syntax Definition: The abstract syntax of any language refers to the logical representation of that language in memory. In Figure 5 and 6 we provide abstractions to the user interfaces of both feature and configuration objects in the form of class diagrams. In both meta-models, sets of capability related data pertaining to the same concerns are modelled and encapsulated together. These related user-defined types can be instantiated with variable data values that conform to the requisite type assigned to the attribute in question. For instance, upon instantiation, a modelled robotic feature is selected by default. By design, every modelled feature must bear a unique string *id* attribute which serves as a reference point in key aspects of our framework. Every feature must also bear a *name* attribute that describes its intended purpose and capability.

Furthermore, every modelled feature has a set of constraints which define the scope of the feature in the model configuration space. Feature constraints are categorised into two main groups. i.e. cross functional constraints and binding property constraints. Cross functional constraints refer to *inclusion* and *exclusion* constraints while binding property constraints define the possible state values of binding time and binding mode attributes within configurations. A feature may also contain an aggregation of sub-features. Lastly, every feature is composed of a group, which may comprise of strictly two or more features.

Fig. 5. Feature Meta Model
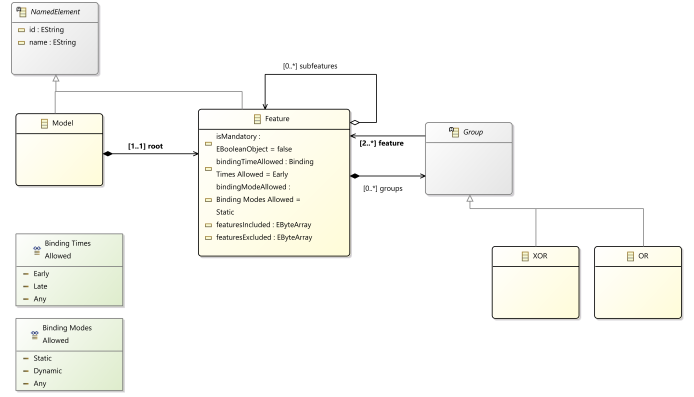
The configuration capabilities of a feature reflects its ability to adapt in accordance to variable contexts. Per our proposed variability management technique, feature adaptations are determined by a binding time and binding mode combination, specified or assigned based on end user preferences. A configuration can only exists as long as a feature exists. Thus, each configuration bears reference to an existing feature by way of an *id* attribute. This explains the existing compositional relationship between a feature and a configuration, as shown in Figure 6. For each feature, its multiplicity can be described as one or more. Thus, there can exist multiple configurations, provided these configurations are valid with respect to all related constraints. Additionally, feature and configuration attributes, together with their associations and functionalities can be inherited through the generalisation of a single named element.

Fig. 6. Configuration Meta Model
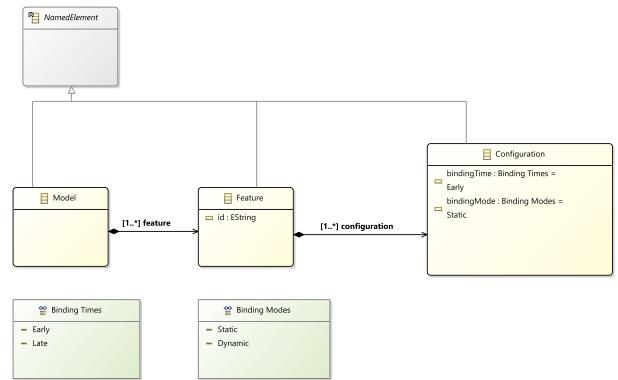
8) Identifying Binding Mechanisms: Realising static, dynamic, early and late bindings in robotic applications can be tricky due to platform and language constraints. For both time and mode based binding, there are a number of optimised implementations and packages found in C++ and ROS, that can be used to realise this.

In Table IV, we have highlighted some of such implementations, otherwise known as mechanisms, which could potentially be used to realise all possible binding pairs found

within our configuration space. These include, the C++ pre-processor or antenna for static, ROS pluginlib for dynamic and a combination of ROS parameters[23] and roslaunch[24] for both early and late binding.

TABLE IV
STATIC AND DYNAMIC BINDING MECHANISMS

Binding Type	Binding Mechanism
Static	C++ preprocessor, antenna
Dynamic	ROS Pluginlib
Early	roslaunch and ROS parameters
Late	Virtual functions, roslaunch and ROS parameters

9) Implementing Binding Mechanisms: The binding mechanisms mentioned in Table IV serve as unique techniques that provide our modelling framework with the capabilities to handle variability on an implementation level. In that, modelled variation points of robot instances that have been described and implemented with our framework, are able to leverage such internal mechanisms within the development environment i.e. ROS, to demonstrate static, dynamic, early and late binding of implemented features within all user defined programs.

These integrated mechanisms are highly optimised, robust and open-sourced implementations that are compatible across a number of platforms. Thus, by utilising their capabilities, we were able to infuse both flexibility and customizability into our implemented feature bindings.

Furthermore, as indicated in Figure 2, features modelled on the language layer can in turn be implemented on the source-code implementation layer as feature classes that form part of an internal feature repository, connected to our implemented mechanisms through a set of interfaces.

- **C++ Preprocessor for Static Binding:**

```
#ifdef AMCL
 #include "../../featx/Amcl.h"
#endif

#ifdef MTNPLNCTRL
 #include "../../featx/MotionPlanningCtrl.h"
#endif

#ifdef COMPCTRL
 #include "../../featx/ComponentControl.h"
#endif
```

Listing 5: Examples of Static Feature Inclusions

Within our application, we have a designated static feature loader module that executes a series of conditional statements in the form of preprocessor directives, to determine which features are eligible to be included at compile time.

As shown in the instance captured in Listing 5 above, implemented feature classes *Amcl.h*, *MotionPlanningCtrl.h* and *ComponentControl.h* will be included in our user

[23]http://wiki.ros.org/rosparam
[24]http://wiki.ros.org/roslaunch

program build if they are explicitly defined as part of the configuration. This mechanism exists strictly for static features that are loaded at compile time.

For static features loaded at runtime, we implemented an immutable plugin mechanism which disallows changing a given feature after it has been loaded at runtime.

- **ROS Pluginlib for Dynamic Binding:**

```
    PLUGINLIB_EXPORT_CLASS(motivml_plugins::Slam,
↪  plugin_base::PluginInterface)

    PLUGINLIB_EXPORT_CLASS(motivml_plugins::Hands,
↪  plugin_base::PluginInterface)

    PLUGINLIB_EXPORT_CLASS(
    motivml_plugins::Pointscloud,
↪  plugin_base::PluginInterface)

    PLUGINLIB_EXPORT_CLASS(
    static_integration::Amclros,
↪  static_base::StaticInterface)
```

Listing 6: Examples of Dynamic Feature Plugin Implementations

To simulate dynamic binding in general, we used the ROS pluginlib package to encapsulate feature classes as plugins. Our choice to use ROS pluginlib was influenced by the fact that it is lightweight, robust, highly optimised and easy to integrate as a third party library. Pluginlib provides our framework with a microkernel-like structure that allows end users to add and remove feature extensions from their implemented models without restarting the core of the applications they have built with our framework. In addition, it also comes with a very intuitive documentation[25] backed by an active community of maintainers and contributors. The code snippet in Listing 6 demonstrates how ROS pluginlib can be used in our framework, to export classes that can later be consumed by programs as plugins.

- **ROS Params and roslaunch for Time Binding:** On the ROS parameter server of our application, categorised lists of bound features of a given robot instance, are stored to be able to categorically load them at specific points in time upon the execution of the *roslaunch* command.

Virtual functions in C++ were also considered as a viable option for late binding. However, due to factors such as seamless ROS compatibility and high-level abstraction, other equally capable mechanisms were favoured ahead of it.

10) Integrating Binding Mechanisms Into Framework Components: To provide roboticists with a SERA integrated solution that has the means and techniques to implement variability, we integrated our implemented binding mechanisms on a component level into out framework solution. Figure 2 shows the details of our application's landscape design together with the component interactions that exists between our variability

[25]http://wiki.ros.org/pluginlib

modelling language tier and our robot builder tier. To elaborate more on this, we have provided a brief description of our chosen mechanisms with respect to their functions below.

- **Language Tier:** The language tier and its components enable domain stakeholders to translate feature models into MoTiVML object models where time and mode bindings can be defined with adequate constraints. At this layer, our language provides both schema and constraint checker tools, purposefully for validating user defined models for syntactic and semantic correctness respectively. Furthermore, the language tier contains a parameter initializer which instantiates and initializes ROS parameters stored in a yaml configuration file used by our framework to enforce time based binding.

- **Robot Builder Tier:** The robot builder tier can be defined as a logical component made up of a group of components, present in the source-code implementation layer. The robot builder is wrapped as a ROS node within our source code implementation layer and subsequently linked to the language layer via a *rostopic*[26]. It provides interfaces through which class implementations of registered features in our user defined models can be bound. Each registered feature is represented by a self containing class that inherits from a base feature class as well as a configuration class.

  At runtime, the robot builder listens for specific configuration commands executed on the language tier from the MoTiVML console interface over a ROS topic. For statically bound features, conditional C++ preprocessor statements are used to include or exclude feature classes to and from robot programs. Static features communicate with our framework through a static class interface. This abstract class interface has access to the functionality of any feature class that implements it, through a pure virtual function that has been overloaded through polymorphism. In the case of dynamic binding, there is also a plugin interface that works in a similar fashion to that of the static class interface. However, the ROS pluginlib library is used to register, encapsulate, export and load feature classes dynamically into prebuilt robotic programs in this instance. These programs are presented as flexible and cleanly separated feature plug-ins supported by the core logic of ROS's pluginlib implementation. Finally, this component also reads application parameters initialised on the ROS parameter server, in order to retrieve and classify program features into time based categorizations.

D. Static Semantics

In the compile time state of our variability modelling language, program syntax is evaluated based on internally established feature relationships and constraints that define the static semantics of our language.

The scope of our static semantics, which can be verified at compile time, includes data type evaluation i.e. whether or

---

---

not all tokens have been declared, which token declaration applies to which instances and so on. When it comes to controlling the semantic scope, constraint type declaration and implementation as well as the model level effect of constraints on each other can be inferred from feature interaction in a configuration.

1) Describing the Configuration Space: As shown in the concrete syntax of our feature modelling language, every instance of a feature has a binding time and binding mode attribute assigned to it. The interaction that occurs between these binding attributes, exits in a scope known as the configuration space. Within this scope, each feature possess dependencies and constraints that validate or invalidate defined feature instances. With the aid of propositional logic, table V provides a description of all likely dependencies and constraints of any given model instance. Per the results displayed in table V, mandatory features exhibit a bidirectional logical relationship between features indicating that both features require or depend on each other in all instances in which they are implemented. Conversely, an optional feature indicates a unidirectional behaviour between a parent and child feature, where a child feature may imply a parent feature but the parent feature on the other hand can exist without that child feature. For cross-functional constraints in the form of inclusions and exclusions, feature inclusion translate to an implication constraint while exclusion could be represented as $A \Rightarrow \neg B$ or $\neg (A \land B)$.

TABLE V
PROPOSITIONAL LOGIC MAPPING FOR FEATURE MODEL ATTRIBUTES

Attribute Name	Feature Model Notation	Propositional Logic
Mandatory	(feature diagram: A—B, filled circle)	$A \Leftrightarrow B$
Optional	(feature diagram: A—B, open circle)	$A \Rightarrow B$
Includes (requires)	A implies B	$A \Rightarrow B$
Excludes	A implies (Not B)	$A \Rightarrow \neg B$

The underlying logic of this language design requires the interaction of a feature's binding attributes, that serves as a variation mechanism for all feature models defined with this language. Table VI captures the semantics of possible binding time and binding mode combinations in any given model. This is relevant because, analysing the combined semantic effect of possible binding pairs in a model, gives insight into the feasibility or validity of combinations in the configuration space. Using propositional logic, Table VI shows the results of every possible interaction between binding pairs with respect to feature attributes such as mandatory, optional, includes and excludes.

TABLE VI
FEATURE MODEL BINDING SEMANTICS: **S = STATIC, E = EARLY, L = LATE, D = DYNAMIC**

A	B	B $\Rightarrow$ A	A $\Rightarrow$ B	A $\Leftrightarrow$ B
SE	SE	1	1	1
SE	SL	1	0	0
SE	DE	1	0	0
SE	DL	1	0	0
SL	SL	1	1	1
SL	DE	0	0	0
SL	DL	1	0	0
DE	DE	0	0	0
DE	DL	0	0	0
DL	DL	0	0	0

2) Constraining the Configuration Space: As part of our static semantics definition to confine our configuration space, we enforced a number of feature level constraints. To implement this, we conceived a number of ground rules or pre-conditions and enforced them as part of our modelled feature capabilities, to serve as our means of managing the scope of robot product lines that can be modelled with our solution. Additionally, we provide a means of validating a feature's constraint settings via an in-built constraint checker mechanism, that parses and interprets configuration attributes, to identify possible constraint violations. The different types of constraints that exist in our language, their purpose, together with their cumulative effect on each other, are described as follows.

- *Inclusion Constraint:* Otherwise referred to as an implies constraint, satisfies the condition that a given robotic feature tends to require others to function. For example, supposing there are two features **A** and **B**, **A** *includes* **B** would imply that, if feature **A** is selected, feature **B** must also be to be selected.
- *Exclusion Constraint:* Some robotic features on the other hand do not require others to function We express this as an exclusion constraint in our implementation. Again, supposing there are two features **A** and **B**, **A** *excludes* **B** would imply that when feature **A** is selected, feature **B** cannot be selected and vice versa.
- *Sub Feature Constraint:* Given the fact that the capabilities specified in our abstract syntax definition support a hierarchical parent-child relationship amongst modelled features, a feature with a *static* mode cannot exist as a child of a *dynamic* parent. Allowing that form of inheritance would imply that an unloaded parent feature would create an orphan child feature, leading to the phenomenon of unused features stuck in runtime.
- *Binding Property Constraints:* As shown in our feature meta-model, the binding time and binding mode attributes of a feature are constrained to a set of valid inputs. By defining a set of valid binding time and mode input values, we can constrain all user-defined features that exist in any model instance.
  - *Binding Time:* The binding time attribute of a feature can assume three distinct values. Binding time can be

TABLE VII
VALID BINDING PROPERTY VALUES

Binding Property	Allowed State Values
Binding Time	Early, Late, Any
Binding Mode	Static, Dynamic, Any

initialised to an *early*, *late*, or *any* state. This means that features can either be strictly bound at compile time, at runtime, or alternate between the two.
  - *Binding Mode:* Presumably, binding mode can also assume three distinct values as well. It can either have a *static*, *dynamic* or an *any* state value. This implies that a feature can be static, dynamic or in the case of an *any*, both.

E. Implementing Defined Features

As an extension to our modelling language, users not only have the capability of defining and modelling features but can also integrate source code implementations of modelled features into their programs. Through the language-agnostic interface we have provided, extracted source code implementations of modelled features, can be encapsulated and integrated into our framework, and in turn referenced based on mechanisms provided within our implementation.

V. VALIDATION

For us to be able to thoroughly validate our language design and its implementation, we have to ensure that its possible to evaluate user-generated artefacts realised with it. These artefacts can be identified as feature models and configurations which contain both data and meta-data. To demonstrate this, we translated graphical representations of our mock-up examples in Figure 3 and 4 into model and configuration instances of our language. We then went on to validate these models and configurations for syntax and semantic correctness or well-formedness. In-built validation mechanisms used by our language for syntax and semantic validation include both a schema and a constraint validation tool. The schema validation tool is used to evaluate model schemas of all features and configurations, whereas the constraint validator handles the semantic evaluation of bindings defined within our mock-up examples.

A. Schema Validation Tool

Our framework provides a schema validation tool that certifies that the grammar and syntax rules of programs created with our language are structurally valid. Using our language, features and configurations can be conveniently modelled as key-value pairs of objects. Grammatically, a feature's object keys are represented as tokens that are not only terminal in nature, but also store meta-data on the features they represent. Value tokens on the other hand are generated from a finite set of data types that are unique to each object attribute present within a feature.

The operations of our in-built schema validation tool can be classified into two categories. i.e. feature schema validation

and configuration schema validation. Feature schema validation demands that a feature's model meta-data is validated by checking for absent or misrepresented tokens. A configuration on the other hand is validated based on meta-data such as its *id*, *binding time*, and *binding mode* properties. In both cases however, in the event of an absent or misrepresented token, an invalid schema error is triggered in the form of a console prompt, indicating the specific feature that bears the error.

B. Constraint Validation Tool

Embedded within our framework is another component, loosely referred to as the constraint checker. This implementation, purposefully evaluates the semantic implications of each feature's constraints against its configuration settings, in the grand scheme of the configuration space. Our constraint validator tool operates by parsing and interpreting feature models to validate two main types of semantics. The first is the set of feature configurations that have been defined within the scope of enforced constraints. The second is the validity of a feature's configuration binding pairs with respect to others within the context of a specific product line definition.

This capability of our solution backs the claim that our binding technique is flexible and customizable. And as such can be used to adapt product lines to fit varying usage contexts.

By evaluating our language according to the challenges and research questions we have previously highlighted in Section I, we can therefore assess the degree to which domain challenges that have driven this study have been fulfilled. In subsequent sections, we analyse each domain challenge in close alignment with our research goals to further prove the relevance of our research outcomes.

C. Lack of a Standard Variability Modelling Framework

This challenge is addressed on several levels. Language-wise, our syntax design is supported by a set of design contracts, enforced to introduce standard syntax and semantic rules that can be validated. Also, under our core mechanisms that define model variation point implementations, light weight features that are robust and platform compatible, can be instantiated per the models defined in MoTiVML. These features can then be implemented and configured based on a wide array of preferred configuration settings. Architecturally, feature instances and their configurations are separated from each other through a streamlined method of decoupling, to eliminate feature attribute duplicates, separate concerns, and reduce configuration complexity.

Secondly, our variability modelling framework is a ROS-specific solution. In that, the entire language and its implemented set of mechanisms are standard to ROS. The advantage here lies in ROS's language-agnostic ecosystem which consists of many libraries and tools that provide a robust abstraction when building robotic applications. ROS ships with drivers, state-of-the-art algorithms, and powerful developer tools that support the quick development and maintenance of robotic applications. Thus, providing an optimally standardised platform for our variability modelling framework to be built upon.

Our solution to this standardisation challenge ties directly into *RQ 2* in Section I which asks the question of how a variability modelling language that allows features to be modelled together with their bindings can be designed. Where the capability of binding time and binding mode modelling guarantees flexibility and customizability in robotic system designs.

Moreover, our framework design and implementation target a wide spectrum of skilled professionals in the robotics domain. Considering the contrast in the skill level of domain experts in the robotics industry, we were able to engineer a framework with a deep focus on usability to accommodate potential users from diverse backgrounds.

D. The lack of mechanisms and guidelines for implementing variability management in robotic systems

There have been some variability modelling language propositions in robotics systems, but none of them offer standard mechanisms together with operational guidelines to assist domain stakeholders in understanding and using such languages. This scarcity is particularly prevalent in the ROS community. Besides this, variability management in itself is often handled in an ad-hoc manner in the form of home-grown configurators and the likes.

To diminish this lack, our variability modelling framework ships with a comprehensive set of guidelines[27] offered as a supporting artifact that also serves as verification for our variability modelling language and the framework as a whole, as well as a means to demonstrate the results of this study. As a quick recap, this study aims to first and foremost, prove the feasibility of modelling variability in robotic systems using a feature's binding time and binding mode. Thus in our accompanying documentation, we verify that our language indeed fulfils the requirements stated in our implementation, while providing verifiable evidence through mechanism-driven examples to prove this.

We deem guideline documentation as a must-have requirement in robotics application development, for three main reasons. These include, (i) For the seamless advancement of robotic application development. (ii) To encourage the standard practice of building and maintaining open-source robotic applications (iii) To encourage transparency and reproducibility of relevant robotic applications. Currently, this is lacking amongst open-sourced robotic applications in general. Due to this lack, we provide a guidelines in addition to our framework, to supplement it by enhancing usability.

In addition to these guidelines, we have also identified and implemented mechanisms, that can be used to implement variability in any robotic system. To verify the relevance of our guidelines and mechanisms, we realised mock-up examples of robots with feature instances that possess different binding time and binding mode configurations to demonstrate the simple, lightweight, and easy-to-understand nature of our

---

[27]https://github.com/SergioGarG/sera-extension/tree/master/documents/guidelineDoc/main.pdf

variability modelling language as well as the underlying mechanisms which are capable of supporting variability modelling and management.

E. Complexity of Engineering Robotic Systems

Robotic systems are complex systems made up of complex application packages integrated into one unified solution. Going by this premise, we can infer that the more a robotic system grows in functionality, the more the internal complexity of the system also grows. Modelling a complex system requires a modelling language that is both syntactically and semantically intuitive. Such a modelling language needs to be able to abstract irrelevant details from end users. In the same vein, implementing a modelled robotic system requires mechanisms that are lightweight, robust, and platform compatible.

In our variability modelling framework, users have the capability of modelling and implementing robotic features with their preferred binding time and binding mode configurations, using the mechanisms and guidelines that we have provided (*RQ 3*). All of these are built on top of ROS which in itself is language agnostic and has become the de-facto standard when it comes to robotic application development. In the language component of our framework, syntax representations in our concrete syntax design demonstrate a close correlation between our user requirements and our language syntax. Furthermore, on a semantic level, through our abstract syntax representations, the configuration space has been well defined and analysed to understand configuration interactions between modelled features.

Overall, the above-mentioned syntax and semantic designs can be verified through in-built mechanisms that prove model and implementation validity on both a syntax and semantic level. Last but not least, validation mechanisms which consist of a schema checker and a constraint checker which have been designed to abstract the complexity of configuring valid models by parsing and validating model instances for correctness have been included in the results of this study.

VI. Conclusion and Future Work

In this paper, we have presented a novel variability management technique that ensures flexible and customizable binding of robotic features. Our contribution is offered as an open-source framework that consists of a variability modelling language and mechanisms, backed by a set of comprehensive guidelines. Thus, our main research goal in this study is to provide proof of the feasible adaptability of a robotic model, based on its assigned time and mode bindings.

Looking into the future, we are hopeful of applying both functional and non-functional adjustments to the work we have done in this research. Functionally, there is the possibility of integrating mission specification mechanisms, that can trigger variability in an automated fashion. This would enhance the coverage of our variability management solution with a mission specification capability that is driven primarily by a goal-oriented approach.

This study generally focuses on technique feasibility rather than operational quality. In view of this, we have identified that there is ample room for improvement in the area of robot execution performance refinements. Here, non-functional improvements in the quality of robot executions towards task fulfilment can be assessed and considered in future studies. This can be defined with regards to other non-functional properties such as safety and task efficiency, in scenarios driven mainly by spontaneous environmental occurrences and limited resources.

REFERENCES

[1] S. García, C. Menghi, P. Pelliccione, T. Berger, and R. Wohlrab, "An Architecture for Decentralized, Collaborative, and Autonomous Robots," in Proceedings - 2018 IEEE 15th ICSA, 2018.

[2] M. Rosenmüller, N. Siegmund, S. Apel, and G. Saake, Flexible feature binding in Software Product Lines, Springer Science + Business Media, LLC 2011.

[3] M. Wirkus, S. Arnold, E. Berghöfer, "Online Reconfiguration of Distributed Robot Control Systems for Modular Robot Behavior Implementation," in Journal of Intelligent & Robotic Systems, Springer 2020.

[4] D. Nešić, J. Krüger, Ş. Stănciulescu, T. Berger, "Principles of Feature Modeling," in Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019.

[5] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, K. Czarnecki, "What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines," in Proceedings of the 19th International Conference on Software Product Line, 2015.

[6] The Partnership for Robotics in Europe, "Robotics 2020 Multi-Annual Roadmap For Robotics in Europe," in Horizon 2020 Call ICT-2016 (ICT-25 & ICT-26), 2020.

[7] K. Pohl, G. Böckler, F. van der Linden, "Software Product Line Engineering Foundations, Principles, and Techniques," Springer, 2005.

[8] F. Bachmann, P. C. Clements, "Variability in Software Product Lines," CMU/SEI, 2005.

[9] S. García, D. Strüber, P. Pelliccione, T. Berger, and D. Brugali, "Robotics Software Engineering: A Perspective from the Service Robotics Domain," ESEC/FSE, 2020.

[10] A. R. Hevner, S. T. March, J. Park, S. Ram, "Design Science In Information Systems Research," MIS Quarterly Vol. 28 No. 1, pp. 75-105, 2004.

[11] M. Ceccarelli, "Service Robots and Robotics: Design and Application," SCOPUS, 2012.

[12] A. Ahmad, M. A. Babar, "Software architectures for robotic systems: A systematic mapping study," The Journal of Systems and Software, 2016.

[13] T. Berger, P. Collet, "Usage Scenarios for a Common Feature Modeling Language," Association for Computing Machinery, 2019.

[14] P. Juodisius, A. Sarkar, R. R. Mukkamala, M. Antkiewicz, K. Czarnecki, A. Wasowski, "Clafer: Lightweight Modeling of Structure, Behaviour, and Variability," The Art, Science, and Engineering of Programming Journal, 2019.

[15] D. Bozhinoski, D. Garlan, I. Malavolta, P. Pelliccione, "Managing safety and mission completion via collective run-time adaptation," Journal of Systems Architecture, 2019.

[16] L. Bass, P. Clements, R. Kazman, "Software Architecture in Practice, Third Edition," Addison-Wesley, 2013.

[17] E. Y. Nakagawa, P. O. Antonino, M. Becker, "Reference Architecture and Product Line Architecture: A Subtle But Critical Difference," 5th European conference on Software architecture (ECSA'11), 2011.

[18] D.C. Schmidt, "Guest Editor's Introduction: Model-Driven Engineering," IEEE, 2006.

[19] R. H. Bourgonjon, "Embedded Systems in Consumer Products," Springer, Berlin, Heidelberg, 1995.

[20] N. Hochgeschwender, S. Schneider, H. Voos, H. Bruyninckx, G. K. Kraetzschmar, "Graph-based Software Knowledge: Storage and Semantic Querying of Domain Models for Run-Time Adaptation," iIEEE ICSE, Modeling, Berlin, Heidelberg, 2016.

[21] A. Steck, A. Lotz, C. Schlegel, "Model-Driven Engineering and Run-Time Model-Usage in Service Robotics," ACM, 2011.

[22] R. P. Pinto, E. Cardozo, P.R.S.L. Coelho, E. G. Guimarães, "A Domain-independent Middleware Framework for Context-aware Applications," ACM/IFIP/USENIX International Middleware Conference, 2007.

[23] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, A. Wasowski, "A Survey of Variability Modeling in Industrial Practice," University of Waterloo, 2013.

[24] T. Berger, R. Pfeiffer, R. Tartler, S. Dienst, K. Czarnecki, A. Wasowski, S. She, "Variability Mechanisms in Software Ecosystems," Chalmers University of Technology, 2014.

[25] A. van Deursen, P. Klint, J. Visser, "Domain-Specific Languages," ACM SIGPLAN Notices, 2000.

[26] J. D. A. S. Eleutério, C. M. F. Rubira, "A Comparative Study of Dynamic Software Product Line Solutions for Building Self-Adaptive Systems," Technical Report - IC-17-05 - Relatório Técnico, 2017.

[27] M. Salehie, L. Tahvildari, "Self-adaptive software: Landscape and Research Challenges," ACM Transactions on Autonomous and Adaptive Systems, 2009.

[28] M. Galster, P. Avgeriou, D. Weyns, T. Männistö, "Variability in Software Architecture: Current Practice and Challenges," ACM SIGSOFT Software Engineering Notes, vol. 36, no. 5, pp. 30–32, 2011.

[29] A. Nordmann, N. Hochgeschwender, D. Wigand, and S. Wrede, "A Survey on Domain-Specific Modeling and Languages in Robotics", Journal of Software Engineering for Robotics," JOSER, 2016.

[30] L. Bass, P. Clements, R. Kazman, "Software Architecture in Practice," Third Edition Addison-Wesley, 2013.

[31] M. Rosenmüller, N. Siegmund, G. Saake, S. Apel, "Combining Static and Dynamic Feature Binding in Software Product Lines," Fakultät für InformatikOtto-von-Guericke-Universität Magdeburg, 2009.

[32] M. Rosenmüller, N. Siegmund, M. Pukall, S. Apel, "Tailoring Dynamic Software Product Lines," GPCE, 2011.

[33] L. Gherardi, "Variability Modeling and Resolution in Component-based Robotics Systems," Thesis, Universita degli Studi di Bergamo, 2013.

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

[35] T. Berger, A. Wasowski, "Principles of Software Language Design," Chalmers | University of Gothenburg, 2019.

[36] M. Ramachandran, R. Atem de Carvalho, "Handbook of Research on Software Engineering and Productivity Technologies: Implications of Globalization," SCOPUS, 2009.

[37] T. Berger, S. She, R. Lotufo, A. Wasowski, K. Czarnecki, "Variability Modeling in the Real: A Perspective From the Operating Systems Domain," ACM, 2010.

[38] K. C. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report," CMU/SEI-90-TR-21. Carnegie-Mellon University, 1990.

[39] 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," LNBIP,volume 147. Springer, Berlin, Heidelberg, 2013.

[40] 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," Lecture Notes in Computer Science, vol 5969. Springer, Berlin, Heidelberg, 2009.

[41] S. García, D. Strüber, D. Brugali, A. Di Fava, P. Schillinger, P. Pelliccione, and T. Berger, "Variability Modeling of Service Robots: Experiences and Challenge," VAMOS. Leuven, Belgium, 2019.