MoTiVML: A Flexible and Customizable Variability Modelling Language for 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 efficiency. In the wake of the current pandemic, industries have had to deal with unforeseen challenges 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, is the implementation and deployment of service robots.**

Service robots can be described 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. Usually, the effective completion of tasks by a 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 fit multiple contexts, environments, or purposes. 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.

As a possible solution to this, 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 defining example systems while studying their existing variabilities; all in an attempt to implement a variability modelling framework, that provides a language together with mechanisms, capable of managing variability with respect to a feature's binding time and mode. In a domain where variability is typically performed in an ad hoc manner, our open source solution provides 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.

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 can 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 intermediate 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 implementations of such service robots include UVD's Model

---

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

C [3] and PAL Robotics's TIAGo Base[4] disinfection range of robots, which have been 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 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 doing so, a typical disinfection robot might make use of core assets such as navigation, obstacle detection, collision avoidance, disinfection, irradiation, and teleoperation when the necessity arises.

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 artifacts [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 operate in such heterogeneous environments, robots possess many unique characteristics in the form of functional and non-functional capabilities, which are reusable in multiple operational scenarios.

Due to this, 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. Such architectures often do not 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 offered 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 that can model features together with their possible bindings, implement them, and above all, one that can ensure model validity and correctness. Various ad hoc mechanisms can be identified in different implementations. However, none of these solutions posses the required capabilities for flexible and customizable modelling, implementation and validation.

2) *The lack of mechanisms and guidelines for implementing variability management techniques in robotic systems:*

---

[3]https://www.uvd-robots.com

[4]http://blog.pal-robotics.com/how-to-build-a-solution-for-fighting-coronavirus-using-the-tiago-base-robot/

Even though a plethora of techniques exist, there aren't any precise guidelines that detail how to integrate and use them to design viable robot configurations.

To the best of our knowledge, ROS based open sourced applications in general, lack a comprehensive set of usage guidelines that verify or provide proof that such robotic applications meet the specifications or requirements that define their capabilities.

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 and platform specific means of doing so. In many cases, robotic application developers resort to undocumented ad hoc methods of implementing variability.

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 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 experts (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 complexity when it comes to collaboratively engineering robotic applications.

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

- **RQ 1:** *What are example instances of feature realizations with different binding modes and times 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 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 modelling features 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 key relevant aspects of the language domain, through domain analysis. Next, based on the domain analysis step, the abstract syntax, concrete syntax and static semantics of the language can be derived.
- **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 our language design, we provide formalisms for modelling features based on the variant derivation concepts of feature models, as well as a novel variation mechanism based on feature binding time and binding mode. We also provide guidelines in accordance with our implemented variability mechanisms in the form of a concise documentation to assist domain stakeholders in implementing and validating custom models of their own.

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, architecture 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

---

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

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. Of these, only a subset of them are described in software engineering or programming language literature. Some best-known examples are PIC[7], KRL[8] and MARTe[9].

In this study, 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 we considered include Mauve[10], Robotml[11], LE[12], and eTaSL/eTC [13].

B. Runtime Adaptation

Runtime adaptation in modern day systems often involves some level of uncertainty. In most software systems today, handling uncertainty in advance is often infeasible and resource intensive. This implies that there may be the need to deal with uncertainty when the knowledge required in that scenario becomes available. At runtime, 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 is 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 in which models provide 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 as an abstraction layer over robotic components, where features can be bound with respect to time and mode, 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 made in our framework implementation where robotic features are assigned a specific binding mode and time to serves as the underlying policy upon which 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 and are sometimes referred to as dynamic Software Product Lines (DSPLs). DSPLs are usually built from coarse-grained components, which reduces the number of possible application scenarios.

---

[7]https://dl.acm.org/doi/10.1145/872730.806459

[8]https://pdfcoffee.com/kss-55-operating-and-programming-instructions-for-end-userspdf-pdf-free.html

[9]https://www.sciencedirect.com/topics/computer-science/domain-specific-modeling-language

[10]https://corlab.github.io/dslzoo/architectures-and-programming-subdomain.html#lesire2012mauve

[11]*#dhouib2012robotml

[12]*#gordillo1991high

[13]*#aertbelien2014etasl

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 that implementation complexities exist in software product lines. Such internal complexities are often due to the effect 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. To be able to effectively support runtime adaptation of programs, Rosenmuller, Siegmund, Apel and Saake integrated a customizable framework, called FeatureAce[14]. By including FeatureAce into a generated DSPL, they gained the capability to compose features and modify configurations at runtime.

Learning points from these studies bear similarities with 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 decided to take the approach of leveraging variability mechanisms that implement binding time and binding mode as a means of adapting our configurations. The difference here lies in the fact that our methods are 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 emphasis on the feasibility and usability of our chosen mechanisms in conjunction with our variability modelling language.

D. Feature Modelling Languages

Feature modelling—"*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,[15] Kconfig,[16] and CDL[17] appear to be quite popular amongst industry professionals.

---

[14]https://sourceforge.net/projects/featurecpp/
[15]https://www.clafer.org/
[16]https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html
[17]http://www.cse.chalmers.se/~bergert/paper/cdl_semantics.pdf

Clafer is one of the most expressive feature modelling 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. It 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 built on top of 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 concepts, they were developed independently from each other, and independently from 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 study conducted by Berger et al. [13], a feature modelling language that is intuitive, simple, and also expressive enough to cover a fair range of important usage scenarios must make provision for a base set of usage scenarios such as Exchange, Storage, Domain Modelling, Teaching and Learning, Mapping to implementation, Model generation, Benchmarking, and Analyses. These selected scenarios were derived from a large number of general usage scenarios elicited from researchers at a meeting during the Software Product Line Conference (SPLC) in September 2018 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 shortfalls 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 properties confirm the relevance and novelty of our feature-modelling language with respect to others, in the robotics domain. Such characteristic properties 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.

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 cover 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 dataflow 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. 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 consists of 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 in 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 a building blocks and hierarchically composed 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.

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 product line variants tremendously.

1) Time - Mode Binding Technique: In a time - mode binding scenario, product line variants can be generated by composing modules 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 the feature has stopped running. Examples of mechanisms that realize static binding include: C/C++ preprocessor and antenna[18] [2]. Implementing

---

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

this exclusively introduces a functional overhead where some features 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 in terms of resource consumption and performance through an increase in binary size and execution time. Examples of mechanisms that realize dynamic binding include: FeatureIDE's runtime parameters[19] and ROS pluginlib[20] 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.

In order to decrease some of the restrictions highlighted by Rosenmüller, different approaches for combining static and dynamic binding have been suggested to retain customizability and flexibility with a minimum 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 [2].

At binding time, an early binding classification implies that a feature is loaded or unloaded at compile time while a late binding classification implies that a feature is loaded or unloaded at runtime.

For both time and mode binding, our proposed variability modelling language supports an "Any" binding type which implies that at a given time (t), the binding attribute of a feature can assume both an early and a late value. Likewise at a given mode (m), a binding mode attribute can assume both a static and dynamic binding value.

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

- **Static Early:** Static Early binding implies that 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 implies 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.

[19]https://featureide.github.io
[20]http://wiki.ros.org/pluginlib)

- **Dynamic Late:** Dynamic Late implies that a feature that can be loaded and unloaded several times is bound at runtime time.
- **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 implies that a feature that can be loaded and unloaded several times is bound at compile time.
- **Any Late:** Dynamic Late implies that a feature that can be loaded and unloaded several times is bound at runtime time.
- **Any Any:** This implies that a feature can be bound either statically or dynamically at compile time or run time.
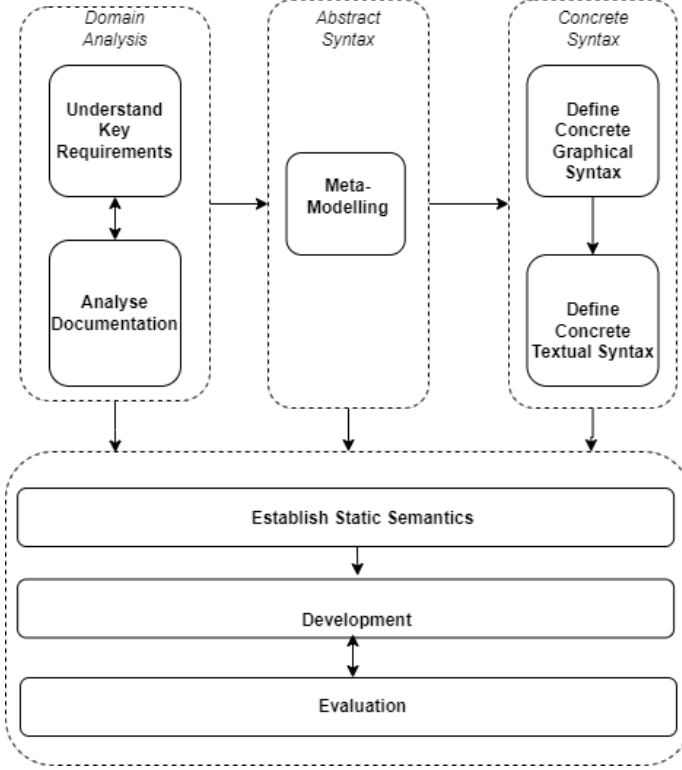
III. METHODS

Our research method of choice is the design science methodology. We chose the design science paradigm due to its deep roots in engineering science. In that, design science generally offers a problem-solving paradigm 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 artifacts that rely on existing theories that are applied, tested, modified, and extended through experience, creativity, intuition, and problem solving in this study.

Our language design and implementation methodology, 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 synergy where outputs from each phase served as input for subsequent phases. In our domain analysis phase, we clarified implementation requirements from sources such as domain experts and relevant documentation. Based on our findings, we conceived the 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.

Fig. 1. Methodology Overview

decomposition of a class diagram together with compositional relationships. 2) Verify the manner in which concepts are classified, and classes are organized in a hierarchy. 3) Concretize relationships if need be. If relationships between concepts posses properties, they need to be transformed into classes. 4) Remove redundancies by finding and removing multiple classes that have the same property. 5) 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 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. In a graphical context, feature models of example systems were conceptualised and implemented with FeatureIDE[21] and then further translated into JSON notation made up of multiple intuitive and configurable attributes.

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. The static semantics mentioned in this context, refers to the underlying rules that determine the validity of model variants that can be created with our language. Key aspects of the static semantics which we clarified and established include:
1) Outlining mechanisms that realise static and dynamic 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 the development phase of our methodology, we utilised domain knowledge gathered from our domain analysis, concrete syntax definition, abstract syntax definition and static semantics as input to implement our domain specific language of the Python programming language. Our development approach can be summarized in 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 trade-offs, as well as implementation algorithms.

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 robotics. We also identified and clarified the purpose of our language as well as the stakeholders our activities are targeting. Finally, we identified all the relations between our gathered domain concepts which we derived from both stakeholders and relevant documentation, by studying the properties of our domain concepts.

B. Implementation

In our language implementation step, we define the capabilities of our language based on prior knowledge from our domain analysis and then conceived both the graphical and textual forms our language, to represent the syntactic structure of our language tokens. Graphically, we used feature models to define and model features of robot examples. On an abstract level, we used class diagrams to describe the capabilities present in 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.1) Create a single

4) Realise language designs with Python based algorithms and data structures. Likewise interfaces for integrating encapsulated feature implementations will be realised in C++[22].

5) Test implementation by evaluating the interactions of all the various integrated system components.

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, which operate by detecting syntax and constraint violations.
- **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 time and binding mode has not yet been applied in robotics.
- **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 comparative 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 ability to model robotic features, define configurations for each feature, and then apply such configurations to multiple scenarios. Configurations are typically an amalgamation of features, relationships and constraints. To assess the validity of configurations described with our language, we have implemented mechanisms that can be used to evaluate the well-formedness of such user defined models and configurations. On a high level, our evaluation plan for our modelling language can be summarised as follows:

1) Realise mock-ups of example robotic systems.
2) Define binding configurations 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 test model variability.

For stakeholders such as end users to be able to verify the correctness of their models, all modelled systems defined in our modelling language must be validated on multiple levels of correctness i.e. syntax and semantic correctness.

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

Semantics Validation: To validate the semantic implications of our language, we evaluate the correctness of configuration bindings in relation to the corresponding binding constraints specified. We also evaluate 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)[23], and implemented as a domain specific language of Python. Architecturally, this language consists of three modular components. These include a syntax manager, a configuration manager, a source code implementation manager. As shown in Figure 2, all these components are encapsulated and offered as an open sourced ROS compatible framework.

The syntax manager component contains all implementations related to language tokens, as well as the language's lexical and syntactical schemas. Tokens represent the legal keywords or lexemes that are permissible in our language. There are two schema representations present in MoTiVML; a feature schema and a configuration schema. Language schemas are simply represented as an object of tokens mapped to selected data structures. Listing 1 and 2 show samples of a feature and configuration schema respectively.

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

Listing 1: Feature Schema

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

Listing 2: Configuration Schema

The configuration manager component of our library is made up of two parts. Namely, a binding semantics interpreter and a constraint checker. In general, variability management provides two main capabilities i.e. feature modelling and variation point definition and management. The binding semantics interpreter assimilates and displays the result of the interaction between the binding pairs of any pair of features. The constraint checker on the other hand parses the entire model definition 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 configurations based on feature binding time and binding mode. The source code implementation manager encapsulates a plugin management interface which provides an integration point where end users can include source code implementations of modelled features into their configurations.

Fig. 2. Architectural Overview of Variability Management Solution
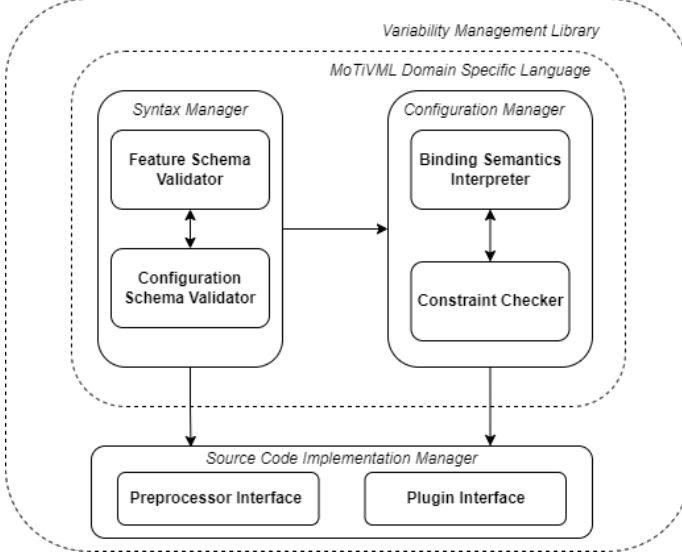
B. Domain Analysis

In our language design, we had to clarify key concepts that are relevant to the domain context of our language. Furthermore, we had to identify design concepts and relationships that determine the type of programs which domain stakeholders will be capable of writing with our language. All the concepts

and relationships we identified, were then formalized into a model, and iteratively refined to develop our language's abstract syntax.

During our domain analysis, we identified five standard questions which make up the core ideas and knowledge base that represents our language.

- **Purpose:** *What is the purpose of this language?*
  The purpose of our language is to provide roboticists with 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 that the ability to operate robotic applications with pre-configured configurations. (ii) Developers: End users in charge of developing new features and implementing them into our framework. (iii) System Engineers: End users in charge of adapting the framework to the specific requirements of a customer.
- **Concepts:** *What are the key domain concepts that targetted stakeholders care about?*
  From an end user perspective, concepts such as simplified feature modelling techniques, standardized configuration management techniques and flexible and 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

In our architectural implementation, configuration capabilities are decoupled from feature capabilities. This is to primarily separate concerns by detaching feature attributes that do not require end user manipulation to form configurations from the ones that actually require it. Each set of decoupled configuration attributes references a corresponding feature by an *ID* attribute.

1) Developing Mock-up Examples: An important use case of our language is that, it can be used to develop mock up examples that demonstrate the language's expressiveness. In the context of our study, examples refer to modelled features of robot product lines. Figure 3 and 4 show graphical representations in the form of feature models, of our selected example systems i.e. service robots. For each feature present in a model, we can express it textually with tokens from our language as shown in Figure 3 and 4. Inspiration for these examples were drawn from general utility robots and disinfection robot product lines such as TIAGo and UVD Model C.
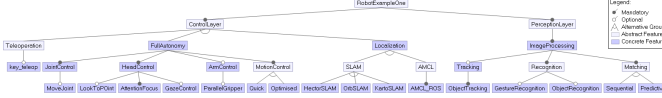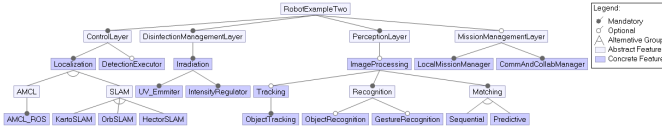

Fig. 3. Modelled Example One

Fig. 4. Modelled Example Two

Furthermore, in our model refinement step, that aims at extending our mock-up examples against our requirements, we analyse the requirements associated with our language, justify if these requirements are met and then proceed to provide examples or 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",
                ...
                }
            ]}]}
```

Listing 3: Textual Sample 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 robotic system. As indicated in our architectural design, configuration attributes are separated from feature attributes for the sole purpose of creating a clear distinction between configurable and no-configurable attributes. Listing 4 captures a sample representation of the concrete textual syntax of a feature's configuration attributes.

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

Listing 4: Textual Sample of Feature Configurations

2) Extending Mock-up Examples Against Requirements: In our domain analysis we were able to understand key requirements relevant to our language's desired capabilities. Based on this understanding, we extended our mock up examples in Figure 3 and 4 against our already established language requirements. Such established language requirements include:

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.
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 which we have highlighted in Listing 3 and 4, we identified example syntax representations that fulfil all the requirements listed above. Table II captures a mapping of each requirement along with its example syntax representation, that justifies that a requirement has been fulfilled.

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 checker and constraint checker model syntax, grammar and semantics can be evaluated

3) Identifying Language Tokens: From a lexical standpoint, our language possesses a defined set of legal or permissible 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 a feature in our language representation of a feature model. Attribute keys serve as meta-data for the data held by attribute values. Table III shows all identified tokens together with their corresponding regular expressions that define their legal or valid lexical structures.

For each identified token in our language, we translated a valid expression of it into a regular expression to demonstrate which valid forms of these token can exist in our language. Syntactically, features and their configurations in our language model attributes as key value pairs wrapped in an object. Table III capture our list of identified tokens mapped to their corresponding regular expression formalisms.

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

4) Specifying Language Terminals: Our syntax can be described as a production consisting of a left-hand-side and right-hand-side attributes. For each attribute, on the left-hand-side, we have non-terminals that represent our language meta-data. On the right-hand-side, 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 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 categories 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 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 identified in Table III with the syntactic categories identified, 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 ⟶ '{' 'id' ':' alphaVal, 'name' ':' alphaVal, 'group' ':' alphaVal, 'isMandatory' ':' boolVal '}'

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

bindingPropertiesBlock ⟶ '{' '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 an abstraction to the user interfaces of both a feature and a configuration objects in the form of class diagrams. Upon instantiation, a feature is selected by default. A feature has the potential of having zero or more sub-features. Two or more sub-features may be grouped together in an OR or XOR group. Each feature's binding property constraint may be set to an *ANY* value. This implies that a binding property may assume a value of either *EARLY* or *LATE* for binding time and *STATIC* or *DYNAMIC* for binding mode. Every existing feature in a model must bear a unique *ID* attribute which serves as a reference point for each feature throughout multiple points in our implementation. A feature bears a *NAME* that describes its capability.

Furthermore, every 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 *includes* and *excludes* constraints while binding property constraints define the possible state values of binding time and binding mode attributes within configurations.

Fig. 5. Feature Meta Model
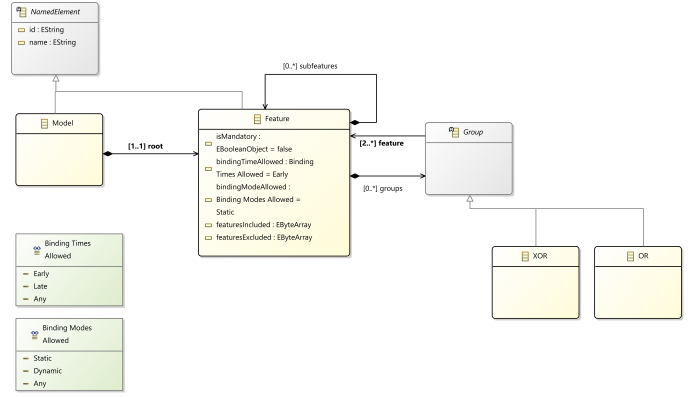
The configuration capabilities of a feature reflects the ability of the feature to be adapted to fit multiple contexts. Per our proposed variability management technique, feature adaptations are determined by a binding time and binding mode combination, specified or assigned based on end user preferences. Each configuration bears reference to an existing feature by way of an *ID* attribute. For each feature, there can exist multiple configurations, provided these configurations are valid with respect to the feature constraints.

Fig. 6. Configuration Meta Model
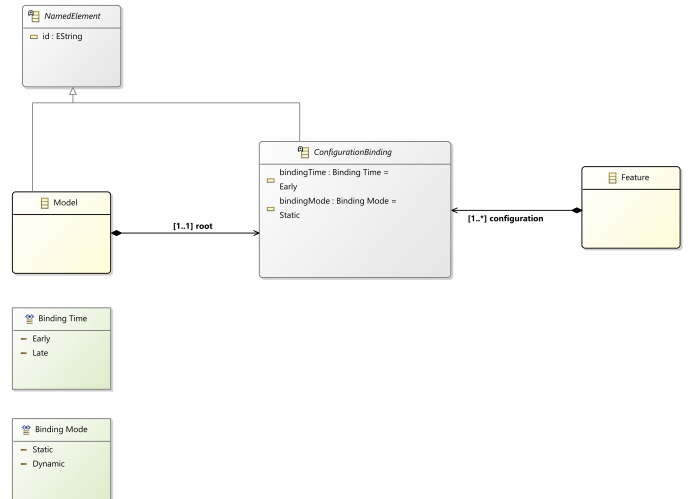
8) Identifying Binding Mechanisms: Realising static, dynamic, early and late binding in robotic applications can be tricky due to minor conflicts when it comes to programming language compatibility. For both time and mode based binding, there are available ROS based C++ and Python mechanisms that can be used to realise them.

In Table IV, we have highlighted some of such mechanisms which we used to realise each binding pair within our variability management technique. These include, the C preprocessor for static early, ROS pluginlib and ROS parameters for static

late, and ROS pluginlib again for both dynamic early and dynamic late.

TABLE IV
STATIC AND DYNAMIC BINDING MECHANISMS

Binding Pair	Binding Mechanism
Static Early	C++ preprocessor
Static Late	ROS Pluginlib, ROS parameters
Dynamic Early	ROS pluginlib, ROS parameters
Dynamic Late	ROS pluginlib

9) Implementing Binding Mechanisms: To provide roboticists with a SERA integrated solution that has the means and techniques to implement variability, we implemented and packaged binding mechanisms in the form of a framework within our solution. Binding mechanisms can be described as implementation techniques that provide our variability modelling language with variability realising capabilities. Model variation points implemented within our language, leverage such techniques by integrating ROS based feature source-code packages to evaluate static, dynamic, early and late binding techniques in user defined robotic programs.

The binding mechanisms we chose to implement in our ROS application, are robust and standardised open source implementations that are compatible across a number of platforms. Figure 7 shows the component interaction between our variability modelling language and our robot builder component on ROS. To complement this, we have provided a brief description of our chosen mechanisms with respect to their functions below.

Fig. 7. Binding Mechanisms Component Diagram
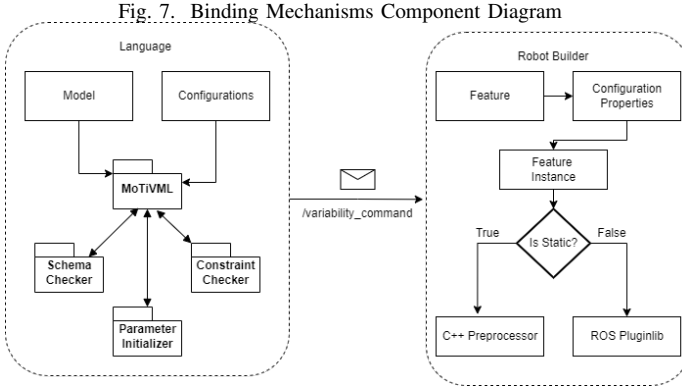
- **Language Component:** The language component of our variability management framework, enables domain stakeholder to translate feature models into MoTiVML object models where time and mode bindings can be defined with constraints. In addition to that, our language provides both a schema and constraint checker purposefully for validating user defined models for syntactic and semantic correctness respectively. Furthermore, the language component contains a parameter initializer which instantiates and initializes ROS parameters used by our framework to enforce time based binding.

- **Robot Builder Component:** The robot builder component of our framework is a ROS node within our source code implementation manager, that provides interfaces through which class implementations of registered features in our user defined models can be bound. It listens for configuration commands from the MoTiVML language interface over a ROS topic. For statically bound features, the C++ preprocessor together with conditional statements are used to include or exclude feature classes to and from robot programs. In the case of dynamic binding, the ROS pluginlib library is used to register, encapsulate, export and load feature classes dynamically into prebuilt robot programs. Finally, this component also reads from initialised ROS parameters to retrieve and classify program features into time based categories.

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. The effect of the interaction between feature binding attributes has a major effect on the configuration space. In addition, each feature possess dependencies and constraints that validate defined feature instances. With the aid of propositional logic, table V provides an outlook into the configuration space of model instances. Per the results displayed in table V, mandatory features exhibit a bidirectional logical relationship between features indicating that both features require each other in all instances in which they are implemented. Conversely, an optional feature indicates a unidirectional behavior 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 such as includes and excludes, includes translates to an implication constraint while excludes could be represented as $A \Rightarrow \neg B$ or $\neg (A \wedge B)$.

Attribute Name	Feature Model Notation	Propositional Logic
Mandatory		$A \Leftrightarrow B$
Optional		$A \Rightarrow B$
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 time and binding mode 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 effectively constrain the configuration space, it is relevant to enforce constraints on features. To do this, the following ground rules or pre-conditions were conceived as a fundamental basis for designing and developing our solution. We validate these constraints via an in-built constraint checker that parses configurations to identify constraint violations. The reasoning behind these constraints as well as their implied effects on configurations are discussed subsequently.

- **Inclusion Constraint:** More often than not, some robotic features tend to require others to function. 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 excludes 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:** A static child feature cannot be a child of a dynamic parent. Allowing that form of inheritance would imply that an unloaded parent would create an orphan child feature.
- **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 are able to constrain the main feature attributes of user defined feature model instances.

TABLE VII
VALID BINDING PROPERTY VALUES

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

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

E. Implementing Defined Features

As an extension to our modelling language, users do not only have the capability of defining and modelling features, but also have the capability of integrating source code implementations of features into their models. Through a language-agnostic interface, extracted source code implementations of features can be encapsulated and integrated into our library, where they can be 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 we are able to evaluate user generated artifacts realised with our variability modelling language. These artifacts 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 the models and configurations for syntax and semantic correctness or well-formedness. Validation mechanisms used for syntax and semantic validation include a schema checker and a constraint checker. The schema checker is used to evaluate the schema of all modelled features and configurations, whereas the constraint checker handles the semantic evaluation of bindings defined within our mock-up examples.

A. Schema Checker

Our library provides a schema checker which serves the purpose of validating the grammar and syntax rules of programs created with our language. Using our language, features and configurations can be modelled as key-value pairs of objects. From a grammatical perspective, object keys are represented as tokens that are terminal in nature that store meta-data. Value tokens on the other hand are generated from a finite set of data types which are unique to each object attribute.

The operations of our schema checker can be classified into two categories. i.e. validating the feature schema and validating the configuration schema. Feature schema validation involves validating a feature's model meta-data i.e. id, name, constraints, group and optional status tokens, for the absence or misrepresentation of any of tokens. A configuration on the other hand is validated based on meta-data such as id, props, time and mode. In both cases, an absent or misrepresented token triggers an error in the form of a console prompt indicating the specific feature that bears that invalid meta-data or data form.

B. Constraint Checker

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

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

By evaluating our variability modelling language with respect to the challenges and the 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 by the core mechanism that defines variation point implementations of models in our framework. In our light-weight architecture, features in a model can be modelled, configured and implemented based on different standards. Architecturally, both features and configurations are modelled separately with the intention of simplifying the modelling experience by decoupling configurations from features. This decoupled nature of models reduces complexity by separating concerns.

Secondly, our variability modelling language is a ROS specific solution. In that, the entire language implementation is standard to ROS. The advantage here lies in ROS's language agnostic nature which consists of a set of software 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 supports quick development and maintenance of robotic applications. Thus, providing a 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 binding time and binding mode modelling guarantees flexibility and customizability in robotic system designs.

Moreover, our library design and implementation targets a wide array of skilled professionals in the robotics domain. Considering the contrast in skill level of domain experts, we were able to engineer our 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 techniques in robotic systems

There have been some language propositions in the area of modelling variability in the robotics systems but none of them offer mechanisms together with operational guidelines to assist domain stakeholders in understanding and using such languages. This scarcity is particularly prevalent in the ROS community. Aside that, more often than not, variability management in itself, is handled in an adhoc manner by experts in the robotics domain.

To diminish this lack, our variability modelling framework ships with a comprehensive set of guidelines[24] offered as a supporting artifact that 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 fulfils the requirements stated in our implementation, while providing verifiable evidence through mechanism-driven examples to prove this.

We deem guideline documentations as a necessary requirement in robotics application development, for three main reasons. These include, for the seamless advancement of robotic

---

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

application development, to encourage the standard practice of building and maintaining open source robot applications and last but not least, to encourage transparency and reproducibility of relevant and useful robotic applications. Currently, this is lacking amongst open sourced robotic applications in general. Due to this lack, we have provided a guideline document in addition to this seminal study to supplement our implementation with the sole purpose of enhancing usability.

In addition to the 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 language implementation guidelines and mechanisms, we realised mock up example 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 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 through the use of mechanisms and guidelines that we have provided (*RQ 3*). All of which 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 architectural designs can be verified through in-built mechanisms that prove model and implementation validity on 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.

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. Stǎnciulescu, 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," Addision-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.