Variability Management in Robotic Systems:

A Variability-Modelling Language That Implements Variation Points Based On Binding Time and Binding Mode

Master’s thesis in Computer science and engineering

JUDE GYIMAH
Variability Management in Robotic Systems:

A Variability-Modelling Language That Implements Variation Points Based On Binding Time and Binding Mode

JUDE GYIMAH
Variability Management in Robotic Systems: A Variability-Modelling Language That Implements Variation Points Based On Binding Time and Binding Mode
JUDE GYIMAH

© JUDE GYIMAH, 2021.

Supervisor: Dr. Sergio García Gonzalo, Computer Science and Engineering
Examiner: Dr. Jennifer Horkoff, Computer Science and Engineering

Master’s Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2021
iv
Variability Management in Robotic Systems:
A Variability-Modelling Language That Implements Variation Points Based On Binding Time and Binding Mode
JUDE GYIMAH
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Technological advancements have led to a growing demand for efficient solutions that minimise risk and maximise efficiency when it comes to performing tasks in non-deterministic environments. In the wake of the current pandemic, humanity has been affected by labor shortages, logistic challenges, unexpected strain on supply systems and product shortages caused by safety regulations and health complications. Undeniably, these challenges have had a resounding effect across the global service industry; hence the emergence of viable robotic solutions to alleviate these problems. Service robots are a category of robots that render services to humans. Service robots are often designed to operate in highly heterogeneous environments in collaboration with humans, or other robots. The effective completion of tasks by service robots may involve the combination of a specified set of robotic capabilities. These capabilities are mainly driven by robotic features.

Valid combinations of robotic features to fit different contexts, give rise to some level of variability—i.e., the ability of a software artifact to be changed to fit different contexts, environments, or purposes. This constitutes a possible strategy to enable robotic applications to be changed, customized, or configured to fit different scenarios. Thus the need to have an effective mechanism for planning, designing, and implementing variability.

Cognisant of this fact, we present a technique that implements variation points with reference to binding time and binding mode. As a long term goal, this implementation comes as an extension of the Self-adaptive dEcentralized Robotic Architecture (SERA) framework architecture. SERA, which is a decentralized architecture that supports the building of autonomous, heterogeneous, and collaborative robotic applications, lacks the ability to manage variability dynamically. For that matter, SERA and its ilk typically do not provide roboticists with the means and techniques required to manage variability effectively.

With a design science approach, we define example systems as feature models, study the variability of these feature models, and then proceed to conceive a variability-modelling language, that provides mechanisms for managing variability with respect to binding time and mode. Our variability-modelling language is offered as an open-source library that provides basic support for binding features in robotic systems.

This study provides evidence of the extensibility of robotics reference architectures to support variability in a domain where variability is typically performed in an ad hoc
manner. Its implementation is expected to alleviate extension complexity, reduce performance costs, and minimize resource consumption in robotic systems while giving roboticists the flexibility boost they so desire when it comes to engineering robotic systems.

Furthermore, this conclusive study will provide evidence to back the claims that the proposed variability management technique is novel, realizable, useful in practice and provides a means for assessing configuration validity.

Keywords: Variability Management, Service Robots, Robotics Software Engineering, Domain-Specific Languages, Feature Reconfiguration
Acknowledgements

Firstly, I would like to extend my profound gratitude to my academic supervisor Dr. Sergio García Gonzalo, for his patience, support and guidance throughout this entire study.

I would also like to acknowledge Prof. Dr. Thorsten Berger and Prof. Dr. Patrizio Pelliccione for their immense contribution towards this study, in the form of expert advice, guidance and mentorship.

Lastly, I would also like to thank my parents Mr. Davies Gyimah and Mrs. Leticia Gyimah for their love and support throughout my educational journey thus far.

Jude Gyimah, Gothenburg, December 2021
## Contents

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>xv</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>xvii</td>
</tr>
</tbody>
</table>

### 1 Introduction
- 1.1 Problem Statement ................................................. 3
- 1.2 Purpose of the Study ............................................... 4
- 1.3 Significance of the study ......................................... 5
- 1.4 The Need for Efficient Variability Management Mechanisms in Robotic Systems .................................................. 6
- 1.5 Research Questions .................................................. 7

### 2 Background
- 2.1 Software Product Line Engineering ................................. 9
  - 2.1.1 Software Product Line Principles .............................. 9
  - 2.1.2 Benefits of Software Product Lines ........................... 10
- 2.2 Product Line Features .............................................. 11
- 2.3 Modelling Features .................................................. 11
  - 2.3.1 Domain Analysis and Abstract Syntax .......................... 12
  - 2.3.2 Feature Modelling Semantics .................................. 12
  - 2.3.3 Feature Modelling Notation ................................... 13
- 2.4 Variability Management .............................................. 13
- 2.5 Self-Adaptive Systems .............................................. 14
- 2.6 Dynamic Software Product Lines ................................... 14
- 2.7 Self-adaptive dEcentralized Robotic Architecture (SERA) ........... 15
- 2.8 Domain-Specific Languages ........................................ 15
- 2.9 Flexible Feature Binding .......................................... 16
  - 2.9.1 Time - Mode Binding Technique ................................ 17
- 2.10 Robot Operating System (ROS) .................................... 18
- 2.11 Design Science Overview ......................................... 18
  - 2.11.1 Why Design Science? ......................................... 19

### 3 Related Work
- 3.1 Combining Static and Dynamic Feature Binding In SPLs ............. 21
- 3.2 Managing Adaptation Complexity .................................... 22
### Contents

3.3 Feature-Modeling Languages ........................................ 22
3.4 Variability-modelling Frameworks, Middleware, and Toolchains .... 24
3.5 Runtime Adaptation .................................................. 25

4 Methodology ......................................................... 27
  4.1 Phase 1: Modelling ................................................ 28
    4.1.1 Feature Modelling ........................................... 28
  4.2 Phase 2: Implementation ......................................... 33
    4.2.1 Abstract Syntax Implementation .......................... 34
    4.2.2 Concrete Infrastructure Implementation .................. 34
  4.3 Phase 3: Evaluation .............................................. 35
    4.3.1 Correctness Evaluation ..................................... 36
    4.3.2 Coverage Evaluation ........................................ 36

5 Results ............................................................... 39
  5.1 Domain Analysis and Meta Modelling ............................ 39
    5.1.1 Domain Analysis ............................................ 39
    5.1.2 Meta-Modelling with Class Diagrams ...................... 41
  5.2 Mechanisms That Realise Static and Dynamic Binding .......... 42
  5.3 Feature Implementation ......................................... 42
    5.3.1 Feature Meta-data .......................................... 43
    5.3.2 Configuration Meta-Data ................................... 43
  5.4 Configuration Space Description ............................... 44
  5.5 Interface Nodes ................................................ 44
  5.6 Constraining the Configuration Space .......................... 45
    5.6.1 Feature Constraints ........................................ 45
  5.7 Derived System Artifacts ...................................... 46
    5.7.1 Usage Guidelines .......................................... 46
    5.7.2 System Documentation ..................................... 46
  5.8 Realised Example Robotic Systems .............................. 47
  5.9 Implementation ................................................ 48

6 Evaluation .......................................................... 51
  6.1 Correctness Test ............................................... 51
    6.1.1 Evaluation Plan for Correctness ........................ 51
  6.2 Coverage Test .................................................. 55
    6.2.1 Evaluation Plan for Coverage ............................ 55
  6.3 Threats To Validity ............................................. 60

7 Conclusion ........................................................... 61
  7.1 Future Work .................................................... 62

Bibliography ............................................................ 63

A Methodology Feature Models ........................................ I

B Guideline Document .................................................. III
  B.1 Environment ..................................................... III
B.2 How To ................................................................. III
B.2.1 Setup and Run SLED Command Line Tool .............. III
B.2.2 Create and Customize an Example System Model with SLED IV
  B.2.2.1 Create Example System ............................... IV
  B.2.2.2 Show Project List .......................................... V
  B.2.2.3 Activate a Project .......................................... V
B.2.3 View Configuration ............................................. VI
B.2.4 Add and Remove Features ..................................... VI
  B.2.4.1 Implement a Feature Node .............................. VI
  B.2.4.2 Add Launch File Entry ................................. VII
  B.2.4.3 Add Registry Mapping Entry ........................... VII
B.2.5 Alter Binding Time and Mode Properties .................. VIII
B.2.6 Set Constraints on Features ................................. VIII
B.2.7 Configure an Example System Model with SLED .......... VIII
B.2.8 Validate a Configuration with SLED ....................... IX
B.2.9 Load and Unload Features from a Running Configuration . IX
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Automated LoC Count Extraction With Cloc</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Single Relationships</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>OR Group</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>XOR Group</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Conceptual Framework for Design Science Adapted from Hevner et al. [10]</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>Research Methodology Followed in the Thesis</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Iteration 5 Feature Model Snippet</td>
<td>33</td>
</tr>
<tr>
<td>5.1</td>
<td>Feature Meta-model</td>
<td>41</td>
</tr>
<tr>
<td>5.2</td>
<td>Configuration Meta-model</td>
<td>41</td>
</tr>
<tr>
<td>5.3</td>
<td>Modelled Features of TIAGo</td>
<td>47</td>
</tr>
<tr>
<td>5.4</td>
<td>Modelled Features of UVD Model C</td>
<td>48</td>
</tr>
<tr>
<td>5.5</td>
<td>Conceptual model of Framework</td>
<td>49</td>
</tr>
<tr>
<td>5.6</td>
<td>DSL Concrete Syntax Representation</td>
<td>50</td>
</tr>
<tr>
<td>6.1</td>
<td>Correctness Evaluation Process Flow</td>
<td>51</td>
</tr>
<tr>
<td>6.2</td>
<td>TIAGo Validation Report</td>
<td>53</td>
</tr>
<tr>
<td>6.3</td>
<td>Model C Validation Report</td>
<td>54</td>
</tr>
<tr>
<td>6.4</td>
<td>Coverage Evaluation Process Flow</td>
<td>55</td>
</tr>
<tr>
<td>A.1</td>
<td>Phase 1: Iteration One Feature Model</td>
<td>I</td>
</tr>
<tr>
<td>A.2</td>
<td>Phase 1: Iteration Two Feature Model</td>
<td>II</td>
</tr>
<tr>
<td>B.1</td>
<td>Console Interface for Project Creation</td>
<td>IV</td>
</tr>
<tr>
<td>B.2</td>
<td>Application Engine Snapshot</td>
<td>IV</td>
</tr>
<tr>
<td>B.3</td>
<td>List of Projects</td>
<td>V</td>
</tr>
<tr>
<td>B.4</td>
<td>Project List Command</td>
<td>V</td>
</tr>
<tr>
<td>B.5</td>
<td>Activating a Project</td>
<td>V</td>
</tr>
<tr>
<td>B.6</td>
<td>Show all</td>
<td>VI</td>
</tr>
<tr>
<td>B.7</td>
<td>Show Specific Feature</td>
<td>VI</td>
</tr>
<tr>
<td>B.8</td>
<td>Add Feature</td>
<td>VI</td>
</tr>
<tr>
<td>B.9</td>
<td>Remove Feature</td>
<td>VI</td>
</tr>
<tr>
<td>B.10</td>
<td>Alter Binding Time and Mode</td>
<td>VIII</td>
</tr>
<tr>
<td>B.11</td>
<td>Set Include Constraint</td>
<td>VIII</td>
</tr>
</tbody>
</table>
List of Figures

B.12 Set Exclude Constraint ........................................ VIII
B.13 Select Command .................................................. VIII
B.14 Toggle Command ................................................... VIII
B.15 Validate Command ............................................... IX
B.16 Dump Server Settings .......................................... X
# List of Tables

1.1 Robotic Application Code Analysis .................................. 7

3.1 Characteristic Comparison of Feature-Modelling Languages .......... 24

4.1 Inclusion and exclusion criteria for Github Repo. selection .......... 29
4.2 Final Set of Feature Extractions ....................................... 31

5.1 Static and Dynamic Binding Mechanisms ................................ 42
5.2 Binding Effect on Configuration Space ................................... 44
5.3 Valid Binding Property Values .......................................... 45

6.1 Multiple Correctness Test Evaluation .................................... 52
Introduction

Service robots—"a type of robot that performs useful tasks for humans or equipment excluding industrial automation applications"\(^1\)—are gradually becoming an integral part of human existence. It is estimated that the service robotics market will be valued at 24 billion US dollars by the year 2022.\(^2\) According to the International Service Robot Association, service robots can be classified as machines that sense, think, and act to benefit or extend human capabilities and to increase human productivity \([11]\). This implies service robots are sometimes 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 either humans, other robots, or both. Practical implementations of such service robots include UVD’s Model 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 centers, airports, and hospitals.

UVD disinfection robots, for example, operate by moving at a sufficient speed in a 360-degree fashion, while emitting enough UV-C ultraviolet light on relevant surfaces to eliminate viruses and bacteria. In doing so the robots make use of core assets such as navigation, collision avoidance, disinfection, and teleoperation as and when they are needed.

Valid combinations of these core assets to match the operation of the robot in a given context gives rise to some level of variability in 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 \([8]\). The goal of managing variability is to create cost-effective and customisable core assets that are easy to build.

\(^1\)https://www.iso.org/standard/55890.html
\(^2\)https://www.marketsandmarkets.com/Market-Reports/ivd-bric-market-198.html
\(^3\)https://www.uvd-robots.com
1. Introduction

All decisions on variability in design have to be communicated and documented for future use. As an important consequence, it is necessary to have clear representations for variation points, variants, and mechanisms to realise variability. These said representations of product lines can be depicted using a reference architecture.

A reference architecture which is essentially a predefined architectural pattern, or set of patterns, captures the high-level design for the applications of a software product line [7]. These patterns could be partially or completely instantiated, designed and proven for use in specific business and technical contexts, together with supporting artifacts [17].

The ISO/IEC/IEEE 42010[5] is a standard for architecture description of software systems that represents architecture as an aggregated or high-level view of software in terms of computational elements and connectors that enable interconnections between components. Architectural components as computational elements represent an abstraction of executable code and communicate with each other via connectors [17].

Architecturally, robotic systems are often built with a layered approach where each layer encapsulates a specific functionality and depends on the layer(s) above and below to complete a system’s functionality [12]. As a frame of reference, robotic systems are made up of a control layer and an application layer. The control layer is essentially made up of drivers that communicate directly with hardware components and an application layer utilizes the drivers in the control layer to perform a specified set of tasks.

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

The SERA architecture and its ilk do not typically support variability management. This is, however, not desirable in a complex domain such as robotics. Robots are mandated to operate in a variety of environments, which are often unconstrained and human-populated. To operate in such environments, robots possess many different capabilities, for which they are equipped with a large variety of hardware and software components. In this study, the goal is to extend the architecture of SERA with the capability of managing variability to enable customization and improve current practices of building robotic applications.

Most architectures that manage variability are rather ad hoc and do not provide any means of controlling binding modes (i.e., static, dynamic) and time (i.e., early, late) in a systematic way. Another challenge is their inability to keep an overview or understanding of the variability and dependencies while assuring consistency [23].

That being said, one of the fundamental ideas of adaptable software architectures is to establish an overview of understanding, by systematically modeling the adaptation space using representations such as feature models and having dedicated techniques for the adaptation/customization.

In this study, our goal is to provide roboticists with a variability modelling language that has the means and techniques for planning, designing, and implementing variability. This will be achieved by implementing variability mechanisms. Variability mechanisms are implementation techniques to realize variability and they typically comprise techniques for modeling variability (e.g., feature models) and techniques to implement variation points [23].

The core contribution of this study is the technique to implement variation points. This technique builds upon a reference architecture, using time and mode bindings, as an extension to feature models [2]. Time defined as either compile time or runtime and mode defined as either static or dynamic. For this to be as lightweight as possible without requiring new tooling, we envision the realization of this extension as an internal domain-specific language (DSL) in Python.6

1.1 Problem Statement

At any given point in time, a robot only needs a defined subset of features derived from the main set of all its features to perform a given task. Features are logical units of behavior defined by a set of functional and non-functional requirements [5]. The decision to determine valid feature configurations based on the time and mode in which these features are bound can be tricky and somehow cumbersome. Thus, there needs to be an optimum mechanism for doing this for efficient performance results. To derive the configurations effectively, the dependencies and constraints that exist between these features must be well established.

The problem here lies in the re-configuration of robotic systems with respect to variability. SERA and architectures similar to it do not possess this functionality. These robotic architectures do not provide roboticists with the means and techniques required to manage variability effectively. Thus, there is the need to provide techniques and guidelines to do so in the form of a framework.

The semantics of such a framework with binding time and binding mode can be quite complex, since valid (re-)configurations are not only constrained by dependencies among features, but also by valid/invalid combinations of the binding time and binding mode of dependent features. For instance, a feature cannot be bound statically when it is dependent on a dynamic feature, since the latter can be activated or deactivated at any time.

When a scenario like this is not handled properly, there is the tendency of ending

6https://www.python.org/
up in an unspecified system state or a system crash. For instance, a UVD robot model might ship with different features of navigation algorithms that can be loaded dynamically at run time depending on the context it is operating in. These features, however, might have dependencies with the collision avoidance feature, which is bound statically. Unloading a navigation feature that is depended on directly by the collision avoidance feature might lead to a system crash.

In this study, we seek to support end users with a framework that provides flexible mechanisms that support the instantiation of example systems, into variability models. We also provide guidelines that explain how to implement new features to the framework. By allowing end users to implement their own features, they ultimately gain the power to instantiate customized applications.

In summary, the problem we seek to solve with this study and implementation can be visualised on two distinct levels:

- The lack of a standardised variability modelling language for managing features centrally, modeling features together with their possible binding times and modes, and assuring their correctness with valid configurations.

- The lack of a standardised implementation technique for variability management in robotics applications. Various ad hoc mechanisms exist, but no standardized solution exists on how to do that within the robot operating system (ROS). For example in ROS, dynamic features can be realized using parameterization\(^7\) or loadable ROS plugins.\(^8\) Static features on the other hand, might need a preprocessor and some inclusion into the build system. Thus, even though a plethora of techniques exist, there are no specific guidelines that detail which technique to use and how to use it with with ROS or in robotics in general.

### 1.2 Purpose of the Study

The purpose of this study is to provide roboticists with means and techniques for planning, designing, and implementing variability. This is to be implemented as an extension of SERA in the long term. SERA, which is a decentralized architecture that supports the building of autonomous, heterogeneous, and collaborative robotic applications, lacks the ability to manage variability especially in a dynamic way.

Our endeavour will purposely establish lightweight variability mechanisms, made up of techniques to model variability through time and mode bindings, as well as mechanisms for realizing that variability in the actual robotics source code. We will allow roboticists to define their own features, giving them the flexibility they so desire in their implementations. We will also provide an in-built constraints checker

\(^7\)http://wiki.ros.org/Parameter\%20Server
\(^8\)http://wiki.ros.org/pluginlib
to ensure the correctness of the implementation.

The established variability management techniques will be implemented in concrete technological support of SERA and architectures of that nature. We will also propose guidelines, templates, and design patterns to support roboticists in implementing their own features to extend our framework.

Our core contribution, which is realised as an internal Domain Specific Language in Python,9 will include mechanisms to allow roboticists to define feature models in a textual way while facilitating the definition of the time and mode bindings among features in a simpler manner.

It is also our goal to provide as examples well-defined scenarios of implementations that cover different combinations of binding times and modes. These examples will be part of an example system that will be used to support roboticists on the implementation side of the project by providing examples of the functioning of the techniques we have implemented. These scenarios will also serve as a means of evaluating configurations using our framework and language.

### 1.3 Significance of the study

Overall, this study will serve as a relevant contribution to the growing robotics ecosystem by providing a solution with a software architecture that is flexible and customizable, based on binding time and binding mode.

More specifically, this study would extend a robotics architecture to support variability in a domain where variability is typically performed in an ad hoc manner. The robotics domain is a fast-paced one in terms of growth. According to Moore’s Laws [19] the domain will only become more complex with time. Thus, roboticists need to manage variability when it comes to complex and computationally expensive tasks. Consequently, a framework that supports roboticists with a guided means/technique of customizing their products and model bindings of their features will ease the integration and implementation processes.

In summary, the proposed framework implementation would alleviate extension complexity, reduce performance costs, and minimize resource consumption in robotic systems while giving roboticists the performance boost they so desire when it comes to engineering robotic systems. We would however like to reiterate that, to the best of our knowledge, there is no other study that proposes the same solution in the robotics domain. Thus, making the technique behind our proposition a novel one.

---

9https://www.python.org/
1.4 The Need for Efficient Variability Management Mechanisms in Robotic Systems

A full-fledged robotic system is a complex system that consists of many modules. Whether single purpose or multi purpose, the versatility they offer in different scenarios requires the integration of heterogeneous modules. Many of these modules that need to be integrated are 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 robotic applications. These complications translate into complexity when it comes to engineering robotic applications.

To further assess the complex nature of robotic applications, so as to make a case for the necessity of an efficient variability management technique, we identified a handful of random robotic applications and analysed them based on quantifiable metrics such as Lines of Code (LoC) and the number of packages they contain. Granted that LoC is probably not the most reliable metric in software engineering to measure software complexity, it still sufficiently illustrates the problem we seek to highlight. To extract these metrics, we (i) Identified Github repositories of robotic applications, (ii) Retrieved application source code for robots such as TIAGo,10 Ari,11 Talos12 and RB1 Base,13 and (iii) Extracted package and LoC count from each application, either through manual or automated means.

![Figure 1.1: Automated LoC Count Extraction With Cloc](https://github.com/pal-robotics/tiago_tutorials)

<table>
<thead>
<tr>
<th>Language</th>
<th>files</th>
<th>blank</th>
<th>comment</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>92</td>
<td>3662</td>
<td>5839</td>
<td>16235</td>
</tr>
<tr>
<td>C/C++ Header</td>
<td>87</td>
<td>1424</td>
<td>5409</td>
<td>3973</td>
</tr>
<tr>
<td>reStructuredText</td>
<td>17</td>
<td>429</td>
<td>0</td>
<td>2185</td>
</tr>
<tr>
<td>XML</td>
<td>48</td>
<td>166</td>
<td>43</td>
<td>1259</td>
</tr>
<tr>
<td>Python</td>
<td>18</td>
<td>170</td>
<td>115</td>
<td>472</td>
</tr>
<tr>
<td>YAML</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>83</td>
</tr>
<tr>
<td>Markdown</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>GenCat NLS</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>9</td>
</tr>
</tbody>
</table>

**SUM:** 281 6382 11324 25683

---

10 https://github.com/pal-robotics/tiago_tutorials
11 https://github.com/pal-robotics/ari_robot
12 https://github.com/pal-robotics/talos_
13 https://github.com/RobotnikAutomation/rb1_base_common
A keen observation of the derived metrics above provides an idea of the space complexity of robotic applications. In the software engineering discipline, it is usually the case that the complexity of an application grows as the application increases in scope and functionality. The functionalities of a robotic system gives a general idea of the usage contexts in which that robot can operate. That is, a robot cannot perform a task which its functionalities do not support. Thus, a robot with multiple functionalities will most likely be in need of an efficient variability management mechanism. In that regard, the lines of code and the package count statistics presented in Table 1.1, indicate the need for efficient variability management mechanisms in robotic applications.

Based on this need, we present a solution that offers flexibility, customizability at a decent level of abstraction when it comes to variability management in robotic systems.

### 1.5 Research Questions

Our work focuses on providing a variability-modelling language that is capable of modelling variability and implementing variation points based on binding time and binding mode. In view of this, we have formulated the following research questions to guide our design and implementation of such a solution.

- **RQ 1:** What are example instances of feature realizations with the different binding modes and binding 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 modes.

**RQ 2:** How can a variability-modelling language that allows modeling features together with their binding times and binding modes be designed?

A variability-modelling language that allows modeling features together with their binding times and binding modes can be designed by clarifying the key relevant aspects of the language domain, through domain analysis and metamodeling. In that, relevant concepts and relationships between them can be identified and then further formalised iteratively to refine them.

**RQ 3:** What mechanisms and guidelines can be used to implement features with the different binding times and modes in a ROS-based robotic system?

Variability mechanisms are implementation techniques to realize variability and they typically comprise of techniques for modeling variability (e.g., feature models) and techniques to implement variation points. Guidelines to do this will be provided in the form of a concise documentation to assist users in implementing custom features of their own.
2

Background

In this section we introduce background knowledge required to gain an understanding of this study. Here we put together concepts and prerequisite knowledge from referenced sources of literature that are not only related to variability management in general, but also those concepts specific to the variability management technique that is being proposed.

2.1 Software Product Line Engineering

The switch from handcrafted products for individuals to mass production in the past was due to growing customer requirements. This was to design software products in close accordance with customer wishes from a variety of reusable components. The software development paradigm which involves the systematic combination of mass customization and the use of a common platform for the development of software-intensive systems and software products is referred to as Software Product Line Engineering. When developing software product lines, domain experts have to deal with the commonalities and variabilities generated from the use of reusable components of software artifacts [7].

2.1.1 Software Product Line Principles

In most cases, a product could be shipped in two forms. It could either be a customised product or a mass production product. Customised products tend to be more expensive in comparison to mass produced products. While the main drawback that comes with mass production is the lack of diversification in the product.

Generally, customers tend to be content for a while with a standardised product with the same features across board. This contentment usually begins to wane when multiple scenarios which that product line can be used in begin to arise. For example certain vehicles can accommodate a single passenger while others can accommodate multiple passengers. Some can be used in cities, others mainly in the countryside. These concepts customisation and mass production can be applied in the field of software engineering as well. In the field of software engineering, there has been a peculiar need for flexible software systems that have been fueled by the
ever growing needs of customers and end users.

To facilitate mass customisation, the artifacts used in different products have to be sufficiently adaptable to fit into different systems produced in the product line. This means that throughout the development process we must identify and describe the points where products in a product line may differ, in terms of the features they provide, the requirements they fulfil, and in some cases the terms of their underlying architecture. This implies that we need to provide flexibility in all related artifacts to support mass customisation [7].

In that regard, flexibility can be thought of as a precondition for mass customisation. Meaning, predefined realisations can be developed with the general constraint that there are a limited number of possible configurations per product. In the context of software product lines, the flexibility described here is referred to as “variability”. This variability is the fundamental basis for mass customisation.

2.1.2 Benefits of Software Product Lines

Software product lines have become an industry standard for a varied number of reasons. Notable amongst the bunch are effective complexity management, structured means of managing the evolution of a product and last but not least the product line paradigm has increased the maintainability of products.

- **Complexity Management:** The higher the number of customer wishes, the higher the complexity of the derived products. Most especially in the case of software, where code size and complexity has the potential to increase beyond being manageable due to the addition of more and more functionalities. High complexity leads to rapidly increasing error rates, long development cycles, and a higher time to market [7]. Reusable features reduce complexity significantly. A structure that determines which features are reusable at specific variation points reduces the error rate and the development time drastically.

- **Evolution Management:** The implementation of new features in a product line gives the opportunity for the evolution of all kinds of products derived from that product. For example, a developer may desire to introduce a trend towards certain product features for an intended business goal. Thus, product lines provide a means to better organise development for evolution of a product range while reducing development effort compared to single-system engineering.

- **Effective Maintenance:** Changes applied to a reusable component of the product takes effect in all areas that utilise that specific component. In this way, changes do not have to be replicated across products. Ensuring uniformity across products. This can be exploited to reduce maintenance effort.
2.2 Product Line Features

Software systems in product lines can be decomposed into various characteristics that can be classified as both functional and non-functional. The basic unit of such a decomposition is what a feature represents. For a feature to be accurately defined, important characteristics of that feature need to be identified.

It is important to note that different stakeholders may perceive different characteristics of a feature as important. This could adversely affect the robustness of subsequent models built with these features. Nonetheless, one of the most important characteristics of a feature is that it needs to represent a distinct and well-understood aspect of the system [5].

A feature is usually defined as “a logical unit of behavior specified by a set of functional and non-functional requirements” [5]. In a Software Product Line environment, features aid in distinguishing between individual products with the advantage of efficient software module reusability.

These features are in actual sense assets that provide functionalities for the fulfillment of functional requirements of the system. The Software Product Line concept consists of sharing assets across a given line of products where there is a minimal base set of assets running through all of them with the possibility of adding on new features.

While in operation, a robot typically performs a set of specialized tasks geared towards the realisation of a specified goal. However, some tasks are highly complex and require a team of robots, whose capabilities (e.g., perception, manipulation, actuation) are coordinated and supervised, potentially in direct interaction with humans. Such tasks are typically defined in terms of a high-level mission—i.e., a description of the goals that the robots shall achieve [1].

In the process of executing these tasks, it is noteworthy that, the higher the complexity of a mission, the higher the number of features that will be required to complete it and for that matter, the higher the resources those features are likely to consume. In view of this, the robot must be capable of achieving the specified high-level mission with the highest level of efficiency by strategically managing the resources it consumes.

2.3 Modelling Features

Feature models are arguably one of the most intuitive and successful notations for modeling the features of a variant-rich software system [13]. Proposed almost three decades ago, as part of the feature-oriented domain analysis (FODA) method, hundreds of variability management methods and tools have been built upon feature models [13].
2. Background

Feature models help developers to keep an overall understanding of the system, and also support scoping, planning, development, variant derivation, configuration, and maintenance activities that sustain the system’s long-term success [4].

Feature models provide flexibility for a varied range of product lines. The hierarchical representation of feature models shows clear dependencies that exist between features while providing a structure from which features can be loaded or unloaded. This provides us with the much-needed boost we need in our effort to create a solution that scales, is highly reconfigurable, and platform-independent.

2.3.1 Domain Analysis and Abstract Syntax

To design a language, key aspects of the domain need to be clearly identified through domain analysis and meta-modelling. Domain analysis brings to light all the relevant concepts as well as the relationships that exist between those concepts. Meta-modelling formalizes these concepts into a model, and iteratively refine it until the model precisely describes the abstract syntax of the language. This abstract syntax defines which models or programs can be written with the language [35].

A meta-model is a model that precisely defines the parts and rules needed to create valid models in a DSL [35]. A meta-model is the outcome of domain analysis and meta-modelling. The components of a meta-model refers to the domain concepts captured in the language, while the rules are constraints that prescribe the construction of valid models. Valid model constructs, refer to the abstract syntax, which is independent from the actual notation (concrete syntax) of the language. Therefore the correctness of the concrete syntax is dependent on other means, such as the language grammar [35].

2.3.2 Feature Modelling Semantics

Feature modelling is a domain analysis technique that is used to define the software product-line and system families. Within a feature model, features are used to identify and organise the commonalities and variabilities within a domain, as well as modelling the functional and non-functional properties [36].

The semantics of a feature model are defined by the set of feature configurations that the feature model permits. The common approach is to use mathematical logic to capture these semantics.

In a feature model, the features are connected either individually or in groups through unique relationships with semantic translations. These unique relationships have a direct impact on how variability can be managed within the feature model.

In the subsequent sections we discuss the various standard notations used to represent relationships between features as well as their overall effect on the variability of a configuration.
2.3.3 Feature Modelling Notation

Mandatory relationships are denoted by the *iff* operator while in optional features, the *implies* operator is used. Thus, from figure 2.1 in a mandatory setting, \( f \) as a sub-feature of \( p \) exists if and only if \( p \) exists while in an optional sense \( f \) implies \( p \).

On the flip side, when it comes to groups, we have both exclusive and non-exclusive options—i.e., one or more or exactly one option between a parent and sub-features, respectively. With reference to figure 2.2 and 2.3

\[
(f_1 \lor \ldots \lor f_n \leftrightarrow p) \land \bigwedge_{i<j} \neg (f_i \land f_j)
\]

define the sub-feature options of the parent feature \( p \). The difference lies in the exclusivity exhibited in figure 2.3 which restricts the options to exactly one at a time as opposed to one or more in figure 2.2.

2.4 Variability Management

In Software Product Line Engineering, the variability of a given product line can be summed up into two basic concepts. Which are, a core asset base and a variation mechanism. A core asset base indicates which aspect of the system is to stay the same throughout variants while variation mechanisms are used in core assets to help control the required adaptations and to support the product developers in their task.
Core assets are usually designed with built-in points in their architecture where they can be quickly tailored to fit a context [30]. These built-in architectural points are referred to as variation points. Once the core assets are thoroughly defined, a system can be built by (i) accessing the appropriate assets in the core asset base, (ii) exercising the variation points to configure them as required for the system being built, and (iii) assembling that system [30].

In the service robotics domain, variability management, reliability, flexibility, adaptability and short release cycles are some of the top research areas where academia and industry are earnestly collaborating to improve. The onus of managing variability in a software product line is to maximize the return on investment (ROI) for building and maintaining products over a specified period of time or over a number of products [8]. Systematically managing variability enables the product line organization to predict what goes into building a specific product. By so doing, a centralized model to extract the necessary information to create the production plan for a specific product can be created [8].

2.5 Self-Adaptive Systems

Systems capable to adapt their behavior or runtime structure are called Self-Adaptive Systems. Self-adaptation can be represented in two ways: static, where adaptation mechanisms are explicitly defined by system designers to choose from during execution; or dynamic, where the adaptation plans and monitoring requirements should be produced and selected by the system at runtime [26]. According to Salehie and Tahvildari [27], the adaptation can be divided into two categories: internal and external. In internal adaptation, application logic and adaptation logic are intertwined, being based on programming language resources such as conditional expressions, parametrization, and exceptions. In external adaptation, the adaptation mechanism is in a separate external engine of the application logic.

2.6 Dynamic Software Product Lines

As an extension of software product lines (SPL), Dynamic Software Product Line (DSPL) allows the generation of software variants at runtime. Particularly, SPL emphasizes the variability analysis, decision-making and product configuration at the design phase. DSPL on the other hand, emphasizes variability analysis at design time, while postponing the decision of the variability and the application reconfiguration to runtime. In DSPL, variation points are bound at runtime allowing the software product line to be reconfigurable at runtime. SPLs have the capability of using static binding to generate configurations at compile time.

Depending on the underlying composition mechanism, features are either bound statically (e.g. at compilation time or in a preprocessing step) or dynamically (e.g. when loading a program or at runtime). Both binding times have benefits: static binding facilitates customizability without any cost at runtime whereas dynamic
binding allows a programmer to flexibly select and bind features at runtime, however, at the cost of performance and memory consumption [31].

In the case of DSPLs, the binding time is restricted or constrained to runtime giving rise to product adaptation in support of dynamic variability. Dynamic variability (also called late or runtime variability) can be represented using dynamic compositions, which is a set of features with dynamic binding, whereas, static variability can be represented using static compositions, which is a set of features with static binding [26].

2.7 Self-adaptive dEcentralized Robotic Architecture (SERA)

SERA is a layered architecture that contains components that manage robotic system adaptations at different levels of abstraction by communicating through well-defined interfaces. According to García et al. [1], SERA was created to solve three main 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.

SERA’s components are defined and allocated into two main units. Namely the central station and the robot. The central station interprets high-level mission specifications, comprising a hierarchy of three components, i.e., High-level specification manager, Global mission manager and the Global mission decomposer. With these three components, high-level global missions can be specified and the feasibility of the mission can be assessed and further decomposed into local missions to be accomplished by a group of robots [1]. One of the selling points of this architecture lies in its ability to be implemented within a broad range of projects. To add to that, SERA’s architecture can also be realized using different middlewares and component frameworks related to robotics. This further emphasises the extensibility of the framework architecture in anticipation of future changes.

2.8 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
2. Background

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 modeling, 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 modeling 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. Best-known classical examples are PIC, SCATTER, CHEM, LEX, YACC, Make, KRL\(^1\), RAPID\(^2\) and MARTe\(^3\).

To be able to conceptualise such a language, inspiration had to be drawn from existing DSLs so as to offer a user friendly yet intuitive DSL that possesses a level of abstraction that fits multiple end users with varying skill sets. To do so, dsl zoo [29] was instrumental in providing an annotated bibliography of domain-specific languages in the area of robotics and automation technology. Some example DSLs in that capacity include Mauve\(^4\), Robotml\(^5\), LE\(^6\), and eTaSL/eTC,\(^7\)

### 2.9 Flexible Feature Binding

A plethora of binding techniques exist in SPLE. These binding techniques possess unique attributes that set them apart from each other. Two of such techniques are time and mode binding. Time in terms of compile time or runtime and mode in terms of how static or dynamic the feature is allowed to be. Flexibility can be achieved when these individual techniques are combined effectively to maximise each

---

1https://pdfcoffee.com/kss-55-operating-and-programming-instructions-for-end-users.pdf-free.html
3https://www.researchgate.net/publication/224130823_MARTe_A_Multiplatform_Real-Time_Framework
4https://corlab.github.io/dslzoo/architectures-and-programming-subdomain.html#lesire2012mauve
5https://corlab.github.io/dslzoo/architectures-and-programming-subdomain.html#dhouib2012robotml
7https://corlab.github.io/dslzoo/architectures-and-programming-subdomain.html#aertbelien2014etasl
other’s potential.

2.9.1 Time - Mode Binding Technique

Static and dynamic binding techniques are not new to software product lines. Both binding techniques provide a plethora of unique benefits when it comes to software product line design and implementation.

Using static or dynamic binding exclusively is often not feasible for several reasons. For example, static binding cannot be used when required features are not available or known at deployment time; as is the case for third-party extensions. To cope with the limitations of current implementation techniques, different approaches for static and dynamic binding are combined in practice.

A combination of both binding techniques can improve performance while providing flexibility and customizability. Additionally, supporting different binding times based on the same implementation mechanism simplifies Software Product Line development and maintenance [2].

By merging the unique concepts of time and mode binding, the following binding combinations can be realised:

- **Static**: Static implies that once a feature is bound, the feature cannot be rebound. That is, an activated feature cannot be deactivated anymore and vice versa. Examples of mechanisms that realize static binding include: C preprocessor, antenna⁸ [2].

- **Static Early**: Static Early 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.

- **Dynamic**: Dynamic implies that variation points bound dynamically in a software product line can be altered, however this usually entails an overhead in terms of resource consumption and performance. Examples of mechanisms that realize dynamic binding include: FeatureIDE’s runtime parameters⁹ and ROS pluginlib¹⁰ from ROS’s ecosystem [2].

- **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 implies that a feature that can be loaded and

---

⁹[https://featureide.github.io](https://featureide.github.io)
¹⁰[http://wiki.ros.org/pluginlib](http://wiki.ros.org/pluginlib)
unloaded several times is bound at runtime time.

2.10 Robot Operating System (ROS)

In the application development domain, practitioners usually rely on frameworks to develop software. As such, when it comes to robotics, ROS seems to be the preferred framework among roboticists [9].

ROS can be described as a flexible framework for building robot applications. It consists of a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.

A study conducted by García et al. [9] revealed that ROS is the current de facto framework for robotics. It is already available for a huge variety of robots including, among others, ground mobile robots, static manipulators, autonomous vehicles, humanoid robots, unmanned aerial vehicles, and underwater robots [9].

ROS was chosen as the go-to middleware for our implementation for several reasons. It is open source, stable, has a large ecosystem and provides an optimal level of abstraction for quick, flexible and robust robotic application development. With these quality attributes, it is evident that ROS is tailored to do the heavy lifting for roboticists and thus emerged as the perfect candidate to ensure the rapid development and quick delivery of a standardised solution like ours.

2.11 Design Science Overview

The general methodology used in this thesis is design science. The goal of design science research is utility. For that matter, truth informs design and utility informs theory [10].
With reference to the proposed conceptual framework for design science by Hevner et al. [10] in the context of information systems, design science is depicted in three major units in Figure 2.4. Namely, the environment or the problem space containing the phenomenon of interest, which includes people, organizations, and technology. The knowledge base composed of foundations (e.g., theories, frameworks, models) and methodologies and lastly, the research, which requires both business needs and knowledge to build and evaluate artifacts.

2.11.1 Why Design Science?

The design-science paradigm is deeply rooted in the engineering sciences. It is fundamentally a problem-solving paradigm that seeks to create innovations that define the ideas, practices, technical capabilities, and products through which the analysis, design, implementation, management, and use of information systems can be effectively and efficiently accomplished [10]. Such artifacts are not exempt from natural laws or behavioral theories. Leveraging on the design science paradigm, our goal is to create artifacts that rely on existing theories that are applied, tested, modified, and extended through experience, creativity, intuition, and problem solving.
2. Background
Related Work

In this section we explain and position our study with respect to the related work in the form of relevant knowledge, skills and technologies that have inspired and driven us in our quest to find a novel solution for the problem at hand.

3.1 Combining Static and Dynamic Feature Binding In SPLs

Instances of customized software applications differ by the features they possess. Depending on their underlying composition mechanism, features are either bound statically or dynamically. Combining both static and dynamic binding techniques when generating selections of features is a good way to maximise the advantages that come with both. However this does not alleviate us from the disadvantages that both static and dynamic binding techniques present i.e overhead. We can easily solve this problem by harnessing the power of dynamic binding units. Rosenmuller et al. [31] discuss the different type of overheads that come with both static and dynamic feature binding. They explain that static binding introduces a functional overhead when features are introduced in a configuration without being used whilst dynamic binding on the other hand introduces a compositional overhead that stems largely from memory consumption and performance.

Using dynamic binding units in SPLs is not a new concept. Many seminal studies have discussed and made a strong case for the role in which this technique can play in balancing out the overhead of combining static and dynamic feature binding techniques. Rosenmuller et al. [31] in their quest to propose an approach that integrates both static and dynamic feature binding stated that, depending on the inter-dependency of features, together with the application scenario, we can combine both static and dynamic binding in SPLs. According to them, to do this effectively, we need to understand which features need to be bound dynamically as well as which dynamically bound features should be composed into one binding unit.

With regards to our study, binding time—i.e static and dynamic binding form a core part of the mechanism with which we hope to manage variability in robotic systems. In managing variability, it is very important that we do so with the least overhead
possible. Understanding how features can be composed into dynamic binding units to enhance adaptability, provides us with the technical knowledge we need in constraining the configuration space such that configurations made with our solution will be as efficient as possible in terms of memory consumption and performance.

### 3.2 Managing Adaptation Complexity

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

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

To do this effectively, they generate a DSPL from an SPL by statically selecting the features required for dynamic binding and generating a set of dynamic binding units. To be able to effectively support runtime adaptation of programs, they use a customizable framework, called FeatureAce\(^1\). By including FeatureAce into a generated DSPL, they are able to compose features and modifying configurations at runtime.

Learning points from this study share similar characteristics to our own study in many ways. First off, Rosenmuller et al\([32]\) acknowledge the complexity that exists in software product lines and as such understand 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.

To remedy this problem they chose to use DSPLs composed out of SPLs that are combined into dynamic binding units which can be adapted at runtime. However, we decided to form variability mechanisms using binding time and binding mode as a means of adapting our configurations. The major difference here lies in the fact that our methods are novel and not so much focused on the quality of dynamic binding units. Rather we focus on the feasibility of the mechanism in practice.

### 3.3 Feature-Modeling Languages

Considering the current set of feature-modeling languages that are available, Clafer\(^2\), Kconfig\(^3\) and CDL\(^4\) appear to be quite popular amongst professionals in industry.

---

\(^1\)https://sourceforge.net/projects/featurecpp/
\(^2\)https://www.clafer.org/
\(^3\)https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html
\(^4\)http://www.cse.chalmers.se/~bergert/paper/cdl_semantics.pdf
Clafer is one of the most expressive feature-modeling languages that unifies feature and class modeling. The notion of a feature and that of a class is unified into Clafer. Clafer offers types, constraints, and attributes. Clafer supports multi-level modeling and has well-specified semantics as well as rich tooling, including instance generation, configuration, and visualization. As a textual language, it has one of the simplest, expressive and most intuitive syntax out there 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 languages that have the capability to describe the variability of systems. Even though *Kconfig* and *CDL* share similar concepts, they were developed independently from each other, and independently from feature modeling languages with research origin [37]. *Kconfig* and *CDL* are two of the most successful languages, primarily used in the systems software domain [13].

In a study conducted by Berger et al. [13], a feature modeling language that is intuitive, simple, and also expressive enough to cover a range of important usage scenarios must make provision for a base set of usage scenarios namely Exchange, Storage, Domain Modeling, Teaching and Learning, Mapping to implementation, Model generation, Benchmarking, and Analyses. These selected scenarios were extracted 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 concepts, they also present several limitations. These perceivable limitations have influenced our decision to propose yet another feature modelling language. In comparison to others, our proposed language has been evaluated based on five characteristic properties. These characteristics properties confirm the relevance and novelty of our feature-modelling language with respect to others, in the robotics domain. The five characteristic properties include: (i) C1: Expressiveness (ii) C2: Covers a wide range of usage scenarios (iii) C3: ROS compatible (iv) C4: Abstract syntax supports variability management base on binding time and binding mode (v) C5: Enforces and evaluates binding time and binding mode constraints.

Table 3.1 shows how our proposed feature modelling language compares to other languages with respect to the chosen characteristics.
3. Related Work

<table>
<thead>
<tr>
<th></th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyFML</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HyperFlex</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TVL</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clafer</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KConfig</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDL</td>
<td></td>
<td></td>
<td>✗</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>Proposed Language</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Table 3.1: Characteristic Comparison of Feature-Modelling Languages

From Table 3.1 we can deduce that the ambition of our proposed feature modelling language differs from other existing languages. In that, we are providing a solution that is expressive, a solution that cover a wide range of usage scenarios, a solution that is compatible with the robot operating system, a solution that possesses an abstract syntax that supports variability management based on feature binding time and binding mode and finally a solution that enforces binding time and binding mode level constraints that can be evaluated for correctness.

3.4 Variability-modelling Frameworks, Middleware, and Toolchains

Over the years many robotic component based frameworks and toolkits have been developed. These frameworks possess mechanisms for real-time execution, synchronous and asynchronous communication, control and data-flow flow management and system configuration. In this section we provide a brief walkthrough of some of these frameworks, middleware and toolchains with regards to their influence on variability and system configuration.

When we talk of frameworks, we cannot help but mention ROS, Orocos and SCA. These frameworks have a couple of things 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 systems [33]. (ii) They provide a runtime infrastructure, which is in charge of instantiating, connecting, configuring and activating the components that are part of the system. ROS and Orocos are more tailored to robotics while SCA is more along the lines of a service oriented architecture. As the years have rolled by, ROS has emerged as the de facto standard when it comes to robotic application development.

By design, ROS favors a software development approach that consists in designing components, which implement common robotic functionalities. The strength of this approach is in the possibility to develop a large variety of different control systems by composing in multiple ways, reusable software building blocks. The drawback in this approach is the lack of support to the reuse of effective solutions to recurrent
architectural design problems. Consequently, application developers and system integrators have had to solve these difficult architectural design problems always from scratch. To be specific, the main challenge here lies in selecting, integrating, and configuring coherent sets of components that provide the required functionality taking into account their mutual dependencies and architectural mismatches [34]. As a potential remedy for this, Hyperflex was developed.

*HyperFlex* is a toolchain developed for the development of SPLs 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 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. HyperFlex uses the Feature Models formalism to symbolically represent the variability of a functional system and the constraints 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. 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 combine binding time and binding mode, and apply the resultant combinations to the loading and unloading of features in our feature model. 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 solution.

### 3.5 Runtime Adaptation

Run-time adaptation in modern day systems often involve some level of uncertainty. Handling uncertainty up front in software systems is often not feasible and quite expensive. This implies that there may be the need to deal with uncertainty when the knowledge required becomes available. At run-time, robots need to manage huge amounts of different execution variants that can never be foreseen and completely pre-programmed and can thus not be analysed and checked entirely at compile time.

For this reason, Hochgeschwender et al. [20] employed a model-driven approach. Their approach involves capturing domain knowledge explicitly in the form of domain models. The models which are described by domain-specific languages are needed to be somehow accessed by robots at runtime to take decisions for adaptation purposes. Granting robots access to the software-related models at runtime implies persistently storing the different notations and formats of DSLs, composing the various domain models, and querying over multiple domains at run time. Portions of the methods employed in the study to achieve adaptation align with our study in that resources relevant to the operation of the robot are modeled using DSLs and stored. The stored models are accessed on demand at runtime, when
binding information is available, for the sake of adapting the system to the changing environmental conditions.

As mentioned previously, models are fundamental to the functioning of robots. Steck et al. [21] in their study provide in-depth insight into 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 parameterizations of robotic system components. A model-driven approach also feeds into the idea of making an implementation generic enough to be used by multiple platforms. This supports our choice to use feature models as a layer of abstraction 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 the implementation of their middleware framework for context-aware applications, designed the framework to keep a record of policies that are fed with contextual information obtained by an agent (e.g., a robot) interacting with the environment. Actions tied to functions are triggered when the conditions surrounding a policy are satisfied. The 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 with design decisions made in the implementation of our framework where robotic features are assigned a specific binding mode and time by the user of the framework and this serves as the underlying policy with which the robot will determine how adaptive it can be with respect to the unloading and loading a feature.
In this section, we introduced and thoroughly explain our methodology. By way of design science, we modelled, implemented, and evaluated our solution. Our methodology is broken down into three phases.

Each phase produces a measurable output which can be evaluated. By iteratively running through each concrete step of a phase, we refine the expected output of each phase. Figure 4.1 shows a diagrammatic summary of all phases, the steps involved in each phase, together with the process flow that governs the order of the steps.
4. Methodology

Figure 4.1: Research Methodology Followed in the Thesis

4.1 Phase 1: Modelling

The purpose of this phase is to realise a simple yet highly adaptable core asset base of features that can be instantiated and customised to fit multiple contexts. We achieved this with source code repository mining, feature extraction and feature modelling techniques.

4.1.1 Feature Modelling

Scenarios for each binding combination (i.e., static early, static late, dynamic early, dynamic late); as stated in Section 2.9.1, can be illustrated with several examples of robotic applications, based on real life example systems such as TIAGo, Bluebotics Mini and the likes. Illustrations of these scenarios can be linked to RQ1. To keep these scenarios as realistic as possible, we drew inspiration from Robocup.¹ RoboCup

¹https://atwork.robocup.org/
leagues target the use of robots in industrial and work-related scenarios. Robocup as a platform fosters research and development that enables innovative mobile robots equipped with advanced manipulators and sensors to be used in current and future industrial applications, with a high level of human - robot collaborations that range from manufacturing and automation all the way to general logistics.

Recall RQ 1: What are example instances of feature realizations with the different binding modes and binding times in a ROS-based robotic system?

The selected example systems serve as a reference point for existing feature realisations that demonstrate binding time and binding mode in a pragmatic setting. The goal of this phase is to extract features that can be used to model variability. Instances of features at variation points of our models will serve as example feature realisations that can be configured based on binding time and binding mode.

To achieve this, features from certified public GitHub repositories of example systems such as TIAGo 2, Solo 12 3, Bluebotics Mini 4, Conpleks Robotech 5, Neobotics MP-400 6, Autonomous Weeder 7 or Hibot Float Arm 8 were extracted and documented. The repository selection was for the most part random. However, the following inclusion and exclusion criteria was applied to diversify the range of extracted features.

<table>
<thead>
<tr>
<th>Inclusion criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Open-source Repositories.</td>
</tr>
<tr>
<td>2. Robots that are popular in the service industry.</td>
</tr>
<tr>
<td>3. Robots with autonomous capabilities.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exclusion criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Robots that possess just one feature.</td>
</tr>
<tr>
<td>2. Robots with complex implementations.</td>
</tr>
</tbody>
</table>

Table 4.1: Inclusion and exclusion criteria for Github Repo. selection.

Our Github repository choices of example system implementations, were made up of open-source implementations of popular autonomous service robots with medium

---

2https://github.com/pal-robotics/tiago_robot  
3https://github.com/orgs/pal-robotics/repositories  
4https://bluebotics.com/engineering-services/mini/  
5https://conpleks.com/  
7https://carbonrobotics.com/  
8https://www.hibot.co.jp/products/floatarm/
4. Methodology

complexity, that posses more than one feature. Feature were extracted from the
mined source codes and documented in Table 4.2. From this documentation, we
observed a pattern of overlapping features amongst the selected example systems.

The overlapping features we identified, prioritised, simplified and merged into a
single feature model, as illustrated in Figure A.2. This resultant feature model will
serve as the core asset base of our framework, which can be instantiated, customised
and extended by end users to fit varying contexts. The sum of our observations in
all five iterations of phase one are detailed and discussed in subsequent sections.
4. Methodology

<table>
<thead>
<tr>
<th>Features</th>
<th>TIAGo</th>
<th>Solo</th>
<th>Bluebotics Mini</th>
<th>Compleks Robotech</th>
<th>Neobotics MP-400</th>
<th>Autonomous Weeder</th>
<th>Hibot Float Arm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autonomy</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Control</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Motion Control</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perception</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Teleoperation</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Key Teleop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Irradiation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UVC Emission</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mobile Base</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Command Control</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Joint Trajectory Control</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Head Controller</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kinematic Differential</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Torso Controller</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arm Controller</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gripper Controller</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hey5 Hand Controller</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Look to Point</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distribution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Lift</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drivetrain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Joint Movement Control</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Joint</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Connectivity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Cloud Access</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Navigation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Obstacle Avoidance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Path Planning</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Table 4.2:** Final Set of Feature Extractions

**Iteration One:** In the first ever iteration of Phase One, we focused on ten GitHub repositories of example systems. Namely (i) TurtleBot ⁹, (ii) Atlas ¹⁰, (iii) TIAGo,

⁹https://github.com/ROBOTIS-GIT/turtlebot3
¹⁰https://github.com/openhumanoids/oh-distro
4. Methodology

(iv) Robotnik\textsuperscript{11}, (v) Amigo\textsuperscript{12}, (vi) Robotican Armadillo\textsuperscript{13}, (vii) Coex Clover\textsuperscript{14}, (viii) Crane x7\textsuperscript{15}, (ix) Cyton Gamma\textsuperscript{16} and (x) Gapter\textsuperscript{17}.

From these GitHub repositories, we extracted a wide range of useful robotic features from various categories of robots. These categories include ground (6), aerial (2) and static manipulators (2). A total of twenty-four features were extracted and documented. The features we extracted include, simultaneous localisation and mapping, navigation, teleoperation, collision detection, tracking, metrology, pick and place, perception, motion, laser scanning, pan-tilt movement, object detection, facial detection, take-off, landing, rotation, flight, navigation lighting, compass, anti-collision lighting, mapping, joint control, gripping, arm control. After this, we proceeded to analyse and model the extracted features accordingly. Figure A.1 captures the resultant feature model developed with the features we extracted.

**Observation:** In iteration one, we observed that our selection of Github repositories, though random, was slightly biased. In that, certain categories of robots dominated our selection. In addition to that, many of the features we extracted from the mined Github source codes were quite complex. Thus, we spent way more time than we had anticipated in the feature extraction phase. Furthermore, our feature extractions were not as granular as we would have wanted them to be. And that made our feature modelling experience quite cumbersome and time consuming.

**Iteration Two:** With previous observations from iteration one in mind, we decided to focus on the granularity of our extracted features in iteration two. Granularity in the sense that our extracted features could be further decomposed into finer and much more self-contained sub-features. For example, [perception - facial detection] was decomposed into [computer vision - detection - facial detection]. Figure A.2 captures the resultant feature model after our granularity refinements.

**Observation:** At the end of iteration two, we were content with the granularity of our feature decomposition so we decided to focus on the relationships that exist between various features on all levels of the feature model. These relationships were a mix of singular and group relationships. Fine tuning our model relationships effectively, meant that we had to pay close attention to understanding the constraints that exist between features in our model.

\textsuperscript{11}https://github.com/RobotnikAutomation\textsuperscript{12}https://github.com/tue-robotics/amigo_moveit_config\textsuperscript{13}https://github.com/robotican/armadillo\textsuperscript{14}https://github.com/CopterExpress/clover\textsuperscript{15}https://github.com/rt-net/crane_x7_ros\textsuperscript{16}https://github.com/selyunin/cyton_gamma_300\textsuperscript{17}https://github.com/aniskoubaa/gaitech_edu
4. Methodology

**Iteration Three:** The next point of call after granularity, was relationships that exist between features. We skipped through our entire model to verify the correctness of the relationships that exists between model features. Our model relationships were kept fairly simple with mandatory, optional, alternative and OR group relationships. Once we were satisfied with that, we decided to move on to the next iteration.

**Iteration Four:** In this iteration we collaborated with two domain experts. These domain experts have a strong and credible research reputation when it comes to software engineering design and implementation. We had about two to three brainstorming sessions together that touched on various topics relevant to our study. Together with them, we came to a consensus that due to time and human resource constraints, we had to prune and reorganise our model, into a concise one. We decided to use the Turtlebot feature structure as a reference point in the reorganisation of the model hierarchy. Based on this, feature extensions were made to our existing model. Once this was done and validated by all parties involved, iteration four came to an end.

**Iteration Five:** In the last and final iteration, we solicited feedback from the same domain experts as before, concerning useful adjustments that can be made to generally improve our model. Overall, the feedback was positive with only a handful of structural rectifications pointed out. For example, we were advised to further decouple our configurations from our feature model to be able to manage configuration adaptations better. Furthermore, as a rule of thumb, we also went through a checklist of tasks we have performed from iteration one through to iteration five just to verify that we had not missed any. Figure 4.2 illustrates a snippet of the final feature model we derived after five iterations, modelled with FeatureIDE. This Feature model was constructed with the features documented in Table 4.2.

![Figure 4.2: Iteration 5 Feature Model Snippet](https://featureide.github.io/)

4.2 Phase 2: Implementation

The second phase of our methodology deals with implementation which is realised in two concrete steps. These steps can be summed up as (i) Abstract Syntax Imple-
4. Methodology

mentation, and (ii) Concrete Infrastructure Implementation. These set of activities answer RQ 2 and portions of RQ 3.

| Recall RQ 2: | How can a variability-modeling language that allows modeling features together with their binding times and binding modes be designed? |
| Recall RQ 3: | What mechanisms and guidelines can be used to implement features with the different binding times and modes in a ROS-based robotic system? |

The sum of observations in both steps are illustrated and discussed as follows.

4.2.1 Abstract Syntax Implementation

In our initial step, we modelled the abstract syntax of our language. These models, otherwise known as meta-models were refined in multiple iterations until they finally reflect the capabilities of our variability-modelling language with respect to the problem space. The following steps were followed to realise the abstract syntax of our language.

Abstract Syntax Implementation Steps:

1. Create a single 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 abstract syntax based on how accurately it reflects the problem and how well the concepts of design were kept within scope.

4.2.2 Concrete Infrastructure Implementation

For each component of our application that was implemented, we followed the following steps iteratively to realise that system component.

Concrete Infrastructure Implementation Steps:
1. Gathering system requirements by establishing the purpose of the proposed system in terms of functionality.

2. Analysing requirements by transforming the specified textual functionalities into concrete functional and non-functional requirements. NB: Requirements however had to be prioritised due to time and resource constraints.

3. Establishing design decisions such as the general architecture of the system, dependency management, choice of technologies and trade-offs, as well as implementation algorithms.

4. Realising designs with algorithms and data structures from Python. However, given the fact that this is a language agnostic solution, even though majority of the implementation was in Python, there were still some minor C++ implementations. According to a study conducted by García et al [9], Python and C++ are the most popular programming languages amongst experts in the robotics domain.

5. Testing system functionalities with both exploratory tests and unit tests by verifying the test outputs against expected outputs. The Pytest framework was used to write simple unit tests as verification.

### 4.3 Phase 3: Evaluation

Evaluating our implementation means to assess our implementation from four main perspectives. These perspectives include (i) Correctness, (ii) Realizability, (iii) Usefulness in practice, and (iv) Novelty.

- **Correctness**: This tests the validity of the configurations that will be created from the system. More specifically, the effectiveness of the constraints of the abstract syntax or language will be closely evaluated by its ability to detect constraint violations.

- **Realizability**: Realizability proves that the technique which forms the basis of the implemented variability mechanisms is feasible.

- **Usefulness in Practice / Coverage**: Usefulness in practice has to do with the applicability of the example system configurations realised with the proposed solution.

- **Novelty**: Novelty will be assessed along the lines of the conceptual uniqueness behind the technique that drives the variability mechanisms that constitutes a fundamental part of the proposed solution.

---

19[https://isocpp.org/](https://isocpp.org/)
Realising our variability-modelling language to implement binding time and binding mode mechanisms in a manner that fulfils our research objectives, provides evidence that can be used to evaluate the realizability and novelty of technique that defines our study. In addition to that, we will implement comprehensive evaluation plans to test for correctness and coverage.

4.3.1 Correctness Evaluation

As a functional requirement of our solution, end users would have the ability to build and apply configurations of robotic systems to multiple scenarios. Configurations are typically made up of features, relationships and constraints. To assess the validity/correctness of end user realised configurations, we have implemented a constraint checker that can be used by end users to evaluate configurations in the following manner:

**Evaluation Plan:**

1. Realise a custom configuration
2. Load the realised configuration
3. Randomly introduce constraint violations into the configuration.
4. Use constraint checker to valid the configuration with regards to the inclusion, exclusion, binding and parent/child constraints.
5. Assess/analyse validation feedback and rectify constraints violation.
6. Re-validate configuration with the constraint checker to reaffirm rectifications made.
7. Repeat step 5 and 6 until all violations are rectified

4.3.2 Coverage Evaluation

To prove how practical our solution is, we will use scenarios designed to match the capabilities of service robots, which we extracted from Robocup\(^{20}\), to help end users better understand our usage guideline document. The evidence that user defined configurations can cover real life scenarios, provides proof that our variability management mechanism is feasible and applicable in varying usage contexts.

**Evaluation Plan:**

1. Randomly select and extract scenarios from Robocup\(^{21}\) industrial - logis-

---

\(^{20}\)https://atwork.robocup.org/

\(^{21}\)https://atwork.robocup.org/
tics league.

2. Decompose and simplify extracted scenarios

3. Review applicable example system configuration, i.e. compatibility of custom feature bindings with respect to the scenario selected.

4. For each scenario task, load appropriate feature to perform task and unload feature after task completion.

5. For each task, compare actual outcome to the expected outcome.

6. Review configuration bindings if required.
4. Methodology
Results

In this section we present results derived from multiple sections of our study. This includes, domain analysis and meta-modelling, realised binding mechanisms realised example systems, derived system artifacts and implementation details. For each of these we delve into their relevance to the overall scope of our study as well as the actions performed to realise each result.

5.1 Domain Analysis and Meta Modelling

In designing our language, we first clarified the key relevant aspects of the domain in which our language will be used through domain analysis and meta-modeling. During our analysis, we were able to identify important concepts and relationships that will define the programs end users will be capable of writing with our language. The concepts we identified, together with their relationships were then formalized in the form of a model, and iteratively refined to describe the abstract syntax of the DSL as accurately as possible.

5.1.1 Domain Analysis

During our domain analysis phase, we identified these five standard questions which make up the core ideas and knowledge base upon which our language would be built.

- **Purpose:** *What is the purpose of the 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 users 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:
  
  - All features in a model are not defined as mandatory by default.
  - A parent feature may have zero or more child features.
  - A feature may have zero or more groups but a group must have two or more features to exist.
  - A group could be of an OR or XOR type.
  - 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.
  - Binding time properties of features can only exist in three states. i.e. *Early, Late, Any.*
  - Binding mode properties of features can only exist in three states. i.e. *Static, Dynamic, Any.*

- **Examples:** *What examples of language instances are available?*
  None. Our language is built on novel principles crafted to effectively manage variability in the robotics domain.
5.1.2 Meta-Modelling with Class Diagrams

From an abstract point of view, as shown in Figure 5.1, an instance of our meta-model contains features with properties. Features may contain sub features. Fea-
5. Results

Features may or may not belong to groups. A feature could also be mandatory or optional. However only two or more features can exist in a group which may either be an OR or XOR group. The configuration of a feature on the other hand provides capabilities that enable Binding time and Binding mode values to be adapted according to well defined feature constraints. Feature are developed and implemented into the framework by developers. Through the configuration settings of a feature, selection to or deselection from a configuration may occur. With these capabilities, the framework with predefined features can be used and adapted by operators and system engineer to fit specific customer requirements.

5.2 Mechanisms That Realise Static and Dynamic Binding

<table>
<thead>
<tr>
<th>Binding</th>
<th>Mechanisms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>C++ preprocessor, antenna, rosrun, roslaunch</td>
</tr>
<tr>
<td>Dynamic</td>
<td>ROS pluginlib, ROS parameters</td>
</tr>
<tr>
<td>Early</td>
<td>roslaunch, ROS parameters</td>
</tr>
<tr>
<td>Late</td>
<td>ROS parameters</td>
</tr>
</tbody>
</table>

Table 5.1: Static and Dynamic Binding Mechanisms

Realising static and dynamic binding in robotic applications can be tricky due to programming language barriers. For both static and dynamic binding, there exists both C++ and Python mechanisms that can be used to realise them as shown in Table 5.1. However, integrating both C++ preprocessor and ROS pluginlib into our implementation was not feasible due to language compatibility issues. Thus, to realise static binding, we used roslaunch to bind feature nodes through a launch file. Dynamically, we initialised parameters on the ROS parameter server to keep track of bound robotic features and binding time states. To realise binding time (i.e. Early and Late), we used ROS parameters and roslaunch to set and update server parameters as markers for determining configuration execution states. Early or compile time consists of the time period before the configuration is run. After running the configuration, binding time switches to Late. Thus, at compile time, a ROS parameter is set to early through a ROS launch argument to mark early binding time. Immediately a configuration is run, the same ROS parameter is re-initialised to late indicated runtime.

5.3 Feature Implementation

Based on the knowledge acquired from our domain analysis and our meta-models, we were able to implement our features such that they reflect our modelled domain concepts and constraints.
5. Results

5.3.1 Feature Meta-data

```
{
  "id": "oocje_root",
  "name": "control",
  "constraints": {
    "inc": [],
    "ex": [],
    "tbind": "Early",
    "mbind": "Static"
  },
  "group": "OR",
  "optional": false
}
```

A defined feature has general meta-data such as id, name, group and optional. A feature could be of an XOR or OR group with a mandatory or optional relationship. By default, a created instance of a feature is presented as a static optional feature bound at compile time, which is deselected as depicted in Figure 5.1.

5.3.2 Configuration Meta-Data

```
{
  "id": "oocje_root",
  "props": {
    "mode": "Static",
    "time": "Early",
    "status": true
  }
}
```

The configuration meta-data of an instance of a feature has an id that references an existing feature id in the main feature model. Possible feature constraints include binding property constraints, includes and excludes constraints. Meta-data properties that serve as a means for implementing variation points in our feature models include binding time and binding mode. The binding mode property determines whether that particular feature can be loaded and unloaded on demand or not. Whereas the time property determines whether the feature is bound at compile time or runtime. The status property however provides the possibility to select or deselect a modelled feature to and from a configuration.
5. Results

5.4 Configuration Space Description

<table>
<thead>
<tr>
<th>Binding Combination</th>
<th>Load Operation</th>
<th>Unload Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static - Early</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>Dynamic - Early</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>Static - Late</td>
<td>Valid</td>
<td>Invalid</td>
</tr>
<tr>
<td>Dynamic - Late</td>
<td>Valid</td>
<td>Valid</td>
</tr>
</tbody>
</table>

Table 5.2: Binding Effect on Configuration Space

Table 5.2 describes the configuration space of a robotic model instance at runtime. The configuration space we have described is based on loading and unloading operations which may be performed in a variability management step to adapt the model to varying usage contexts and conditions. Table 5.2 is interpreted as follows.

- **Static Early:** At runtime, loading and unloading a *static - early* feature is impossible in both cases because the feature is bound at compile time and cannot be unloaded when loaded and vice versa.

- **Static Late:** At runtime, loading a *static - late* feature is possible but unloading is impossible because it is static. A static mode implies that a feature cannot be unloaded when loaded and vice versa.

- **Dynamic Early:** At runtime, loading and unloading a *dynamic - early* feature is possible in both cases because even though the feature is bound at compile time, it is dynamic. So it can be loaded and unloaded several times.

- **Dynamic Late:** At runtime, *dynamic - late* features can be loaded and unloaded because they are dynamic and bound runtime. Thus they can be loaded and unloaded several times.

5.5 Interface Nodes

Within models, features are organised in a hierarchical tree format. As a result of this, parent features may have multiple child features. In our implementation, we consider parent features as interfaces through which a path can be traversed to access a specified child feature. For this reason, we do not require end users to provide code implementations for such interface features. Rather, only features with no children i.e. (leaf nodes) in the tree hierarchy are required to have code implementations. These feature implementations are wrapped as ROS nodes that can be executed over a publisher - subscriber network via a dedicated ROS topic.
5. Results

5.6 Constraining the Configuration Space

To constrain the configuration space, we enforced the following ground rules or preconditions as a fundamental basis for designing and developing our solution. We validate these constraints via an in-built constraint checker that scans through a configuration to identify constraint violations. The reasoning behind these constraints as well as their implied effects on configurations are discussed below.

5.6.1 Feature Constraints

- **Includes 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.

- **Excludes 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.

- **Parent - Child 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 Figure 5.1, the binding time and binding mode properties of a feature are constrained to a set of valid states. By defining a set of valid binding time and mode state values, we are able to constrain the main feature properties that influence configurations that can be realised with our implementation.

<table>
<thead>
<tr>
<th>Binding Property</th>
<th>Allowed State Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding Time</td>
<td>Early, Late, Any</td>
</tr>
<tr>
<td>Binding Mode</td>
<td>Static, Dynamic, Any</td>
</tr>
</tbody>
</table>

Table 5.3: Valid Binding Property Values

- **Binding Time:** The binding time property 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 property 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 \textit{any}, a feature can alternate between static and dynamic mode states.

5.7 Derived System Artifacts

In this section we present artifacts that define our system. We provide two distinct artifacts in the form of a user guideline document and an inbuilt system documentation. The purpose of the user guideline document is to help users understand how to interact with our system while the inbuilt documentation provides comprehensive knowledge on our DSL command syntax.

5.7.1 Usage Guidelines

Our guideline document is referenced in the Appendix B. The general content of the document can be broken down as follows:

- **Application Environment Setup:** This covers the platform requirements, source code acquisition and system execution steps necessary to get up and running with our implementation.

- **Variability Modelling:** This section provides steps to guide users through tasks related to instantiating and customising variability models of robots. This includes tasks such as, adding, removing, selecting and altering features.

- **Variation Point Implementation:** Here we thoroughly describe steps for configuring variability models, validating configurations and running configurations.

5.7.2 System Documentation

In addition to the guideline document, we provide an internal documentation\footnote{https://sites.google.com/view/sled-dsl/documentation?authuser=0} implemented as a markdown file. The internal documentation mainly focuses on DSL commands that define user interaction with the system. Details such as command usage, command structure and command descriptions are provided to give users an in-depth understanding of internal DSL commands.
5. Results

5.8 Realised Example Robotic Systems

TIAGo is a mobile manipulator by PAL Robotics. TIAGo typically ships with control, navigation, perception, motion and a point cloud capabilities. Figure 5.3 shows the modelled features of a TIAGo configuration that captures all the above mentioned functionalities in the form of a feature model.

In the figure above, it can be observed that gripping, mapping, routing and locomotion features are deselected upon instantiation with the intention of being dynamically loaded in to the configuration when the need arises.
5. Results

Figure 5.4: Modelled Features of UVD Model C

Model C is designed for infection prevention through regular cleaning routines, and aims at reducing the spread of infectious diseases, bacteria, and other types of harmful organic micro-organisms and pathogens in the environment. With its UV-C light emission feature, it disinfects small and large areas by driving by areas potentially harbouring dangerous microorganisms.

Observably, our Model C representation possesses a uvc emission feature but does not come equipped with either an arm or a gripper. Thus, no arm control or gripper features are present in its configuration.

5.9 Implementation

Our implementation is realised as a Domain-Specific Language (DSL) of Python and offered as an open-source library that provides basic support for binding features in
robotic systems. According to a study conducted by Sergio et al. [9], Python and C++ are the most used General Purpose Languages (GPL) in the robotics domain. However, in our implementation, we leaned more towards python than C++ because Python as a language is simple, fast, and has a large ecosystem that offers a lot of technical support. We decided to realize our solution as a DLS because, an DSL adds to the expressive power of a GPL by extending the syntax and semantics with domain-specific notations and abstractions.

Our entire DSL design and implementation process was iterative. Throughout the process we went through the following steps. We collected the requisite requirements, then proceeded to build some prototypes which we refined in several iterations.

Fig. 5.5, provides a conceptual breakdown of the various components at play, their functions and how they interact with each other.

The solution comprises of four main components. The DSL command line tool serves as an entry point interface from which all other components can be accessed to realize and manipulate configurations. The Registry component stores mappings of feature nodes together with their topics. This platform’s current implementation relies on ROS and provides a set of functionalities for creating, manipulating and validating robot configurations using a novel binding time and binding mode variability management technique governed by constraints.

The command line tool component encapsulates the DSL, realized with Python and packaged as a open-source ROS library. The registry maps ROS nodes with their appropriate topics for easy shutdown and activation of nodes using in-built commands. This component also includes an in-built documentation which provides in-depth information on internal DSL commands such as command usage, syntax and general command descriptions.
5. Results

The robot configurations component consists of a model, configuration and indexing sub-components that are provided when a project is instantiated. All three sub-components are JSON documents that contain a generic model template upon instantiation. The model document provides a hierarchical representation of the features in a robot instance together with its constraints. The configuration document embodies the binding time and binding mode and status parameters required to manage the variability and configuration of an instance. The index document provides a internal mechanism of tracking parent child associations in the model hierarchy.

The command line tool has access to both user encapsulated feature nodes as well as customized model configurations. With this access, it is able to use its configurator tool and constraint checker to orchestrate feature loading and unloading as well as configuration validations.

Figure 5.6: DSL Concrete Syntax Representation

The concrete syntax of our DSL is depicted in Figure 5.6. The DSL component of our implementation functions as an instance of the Loader class. Through that instance, end users can use a command line interface to invoke encapsulated system functions and state. The Engine which serves as the core of the implementation contains the command interpretation and validation operations of the language. The DslState serves as the focal point of command execution to perform read, write, update and delete operations on the states of model definitions. In addition to that, there is a Configurator responsible for enforcing binding rules and validating user defined configurations.
6 Evaluation

As stated in our methodology section, our plan is to evaluate our work with respect to novelty, realizability, usefulness in practice and correctness. To do this effectively, we have explicitly designed and applied both a correctness test as well as a coverage test. The correctness test is to validate example configurations designed with the tools we have provided while the coverage test seeks to prove that user defined configurations are useful in practice. Overall, by successfully realising our variability management technique into a DSL, we have provided proof that our technique is realizable. Coupled with the fact that there is no known application of this technique in the robotics domain, novelty is also proven by default.

6.1 Correctness Test

Figure 6.1: Correctness Evaluation Process Flow

6.1.1 Evaluation Plan for Correctness

Figure 6.1 summarises the process flow of our correctness test. To evaluate the correctness of our configurations, we utilized a constraint checker. Our constraint checker is a self contained module in our DSL that has the ability to parse and validate each feature present in a configuration against four categories of constraints. These four categories of constraints are, includes, excludes, parent/child and binding property constraints. Upon completion, the constraint checker generates a report of features in the configuration that have violated constraints. Key steps that define the process flow for this evaluation plan are as follows:
6. Evaluation

- **Error Injection:** After creating and configuring our TIAGo and Model C example systems in our DSL, we randomly injected constraint errors into our configurations before validating them with our constraint checker. Injected errors for TIAGo include, gripper control inclusion violation as well as tele-operation and autonomy exclusion violations. The touch screen and astar path planning features were also injected with time binding constraint errors while the slam and touch screen features were induced with mode binding constraint errors. For Model C, features like routing, move joint, joint movement and prompter had inclusion, exclusion, parent/child and mode binding errors injected into them respectively.

- **Configuration Validation:** Both our TIAGo and Model C configurations were validated with our constraint checker and the results of the validation are presented below in Figure 6.2 and Figure 6.3. Figure 6.2, shows the report generated after validating our TIAGo configuration. In there, we are prompted about one includes constraint violation, two excludes constraint violation, two time binding mismatches and two mode binding mismatches. The points at which these constraint violations occur are identified by the feature IDs displayed in the console output. Likewise Figure 6.3, shows a similar report. However in this one, we are prompted about one includes constraint violation, one excludes constraint violation, one parent child constraint violation and one mode binding mismatch.

- **Fixing Errors:** Constraint violations in the error reports were fixed in one pass due to the comprehensive nature of the reports generated. Faulty features were tracked using the feature IDs provided in the violation reports. For inclusion and exclusion constraint violations, feature status values were altered to match what is expected while for parent/child errors, dynamic binding mode values for parent features were changed to static. Lastly for binding time and binding mode constraint mismatches, we made sure that binding time and binding mode values aligned with their corresponding constraints.

The same correctness test was repeated three more times with varying amounts of error injections. For each pass, we recorded the number of errors injected versus the number of errors detected. Table 6.1 shows the results of all five passes of our correctness test.

<table>
<thead>
<tr>
<th>Pass</th>
<th>Errors Injected</th>
<th>Errors Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 6.1: Multiple Correctness Test Evaluation
Overall, a sum total of fifty-nine errors were randomly injected into two user defined configurations, evaluated over five passes. As shown in Table 6.1, pass one had seven, pass two had four, pass three had twenty, pass four had twelve and pass five has sixteen injections. All fifty-nine errors were detected and rectified successfully. These randomly injected errors were constrain related errors. They cut across inclusion, exclusion, parent/child and binding property constraints.

Figure 6.2: TIAGo Validation Report
6. Evaluation

**Figure 6.3: Model C Validation Report**

```
dcflib@[uvmodelc]>>validate_config

+++ Validating Configuration: uvdmodelc.....+++  

Include Constraint Violation:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Inclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>ajpzcb_umevrdg</td>
<td>mhyvpln_umevrdg [Status=False]</td>
</tr>
</tbody>
</table>

Exclude Constraint Violation:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Exclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>dptoia_ovongf</td>
<td>ovongf_vtehli [Status=True]</td>
</tr>
</tbody>
</table>

Parent/Child Constraint Violation:

<table>
<thead>
<tr>
<th>Parent</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>ovongf_vtehli (Dynamic)</td>
<td>dptoia_ovongf (Static)</td>
</tr>
</tbody>
</table>

Mode Binding Mismatch Violation:

```

+++ Configuration (uvdmodelc) Validation Complete +++
```
6. Evaluation

6.2 Coverage Test

Figure 6.4: Coverage Evaluation Process Flow

6.2.1 Evaluation Plan for Coverage

Figure 6.4 summarises the process flow of our coverage test. To assess our user defined configurations on their capacity to fit/cover real life robot operational contexts, we identified and defined industrial and work-related scenarios from Robocup. In these scenarios, the capabilities of our example systems will be tested while giving end users an opportunity to better understand the guidelines we have provided for them. Two simplified scenarios from Robocup industrial in restaurant and hospital management contexts, have been outlined below.

- **Extracting Scenarios:** To extract our preferred scenarios, we skimmed thorough Robocup@industrial and Robocup@work 2019 edition. Here, we were able to find restaurant management and hospital management scenarios which we simplified and applied to evaluate our study. Robocup scenarios are usually formulated in a story-like manner with a lot of intricate details that might not be relevant to what we seek to achieve with this study. For that matter, the simplification of our extracted scenarios was necessary for the sake of brevity. Results of our extraction and simplification exercise are as follows.

  - **Scenarios 1: Restaurant Management**
    
    * An item is requested from a particular table
    * Robot plans a route from docking station to item inventory
    * Robot navigates around inventory to find item
    * Robot detects and confirms item matches requested item
    * Robot grabs requested quantity of item from stock
    * Robot plans a route from inventory to the table that issued the request
6. Evaluation

* Robot navigates to table that needs the item
* Robot hands item to customer
* Robot requests customer to confirm that they have received the item
* Robot plans a route from the table back to its docking station
* Robot navigates back to the docking station to wait for another request

– Scenarios 2: Hospital Management

* Instructions are issued concerning an area in the hospital that needs disinfection
* Robot plans a disinfection route
* Robot navigates to the infected area via the planned route
* Upon arrival robot identifies open door
* Robot navigates to starting position
* Robot issues safety instructions to prompt any people in the room to exit the room
* Robot navigates throughout the entire designated area while disinfecting simultaneously
* Upon completion, robot sends prompt to operator
* Robot plans route back to docking station
* Robot navigates back to docking station

• Decomposing and Applying Examples: In this step we defined the execution sequence of the tasks that make up our scenarios. We also applied the appropriate commands that are required to execute each scenario task. We applied TIAGo in the restaurant management scenarios and Model C in the hospital hospital management scenario based on their core competencies. Results of our decomposition and application exercise are as follows:
6. Evaluation

Scenarios 1: Restaurant Management With TIAGo
An item is requested from a particular table:

load egfuvsce_ajpzcbo => Path planning feature is loaded
load yuhejfl_znjmvuc => Localisation feature is loaded

Robot plans a route from docking station to item inventory

unload egfuvsce_ajpzcbo => Path planning feature is unloaded
load umevdg_tarecd => Navigation feature is loaded
load ayxqgu_evtokr => Rolling feature is loaded

Robot navigates around inventory to find item

unload umevdg_tarecd => Navigation feature is unloaded
unload yuhejfl_znjmvuc => Localisation feature is unloaded
unload ayxqgu_evtokr => Rolling feature is unloaded
load jdhdfif_ctdmtv => Object Detection feature is loaded
load netejm_btgivd => Matching feature is loaded

Robot detects and confirms item matches requested item

unload jdhdfif_ctdmtv => Object Detection feature is unloaded
unload netejm_btgivd => Matching feature is unloaded
load gnisyup_fmbcun => Parallel Gripper feature is loaded

Robot grabs requested quantity of item from stock

unload gnisyup_fmbcun => Parallel Gripper feature is unloaded
load egfuvsce_ajpzcbo => Path planning feature is loaded
load yuhejfl_znjmvuc => Localisation feature is loaded

Robot plans a route from inventory to the table that issued the request

unload egfuvsce_ajpzcbo => Path planning feature is unloaded
load ayxqgu_evtokr => Rolling feature is loaded
load umevdg_tarecd => Navigation feature is loaded

Robot navigates to table that needs the item
load umevdg_tarecd ⇒ Navigation feature is unloaded
unload yuhejfl_znjmvuc ⇒ Localisation feature is unloaded
unload ayxqgu_evtokr ⇒ Rolling feature is unloaded
load gnisyup_fmbcun ⇒ Parallel Gripper feature is loaded

Robot hands item to customer

unload gnisyup_fmbcun ⇒ Parallel Gripper feature is unloaded
load ihyvcs_gotsi ⇒ Touch Screen feature is loaded

Robot requests customer to confirm that they have received the item

unload ihyvcs_gotsi ⇒ Touch Screen feature is unloaded
load egfuvsc_ajpzcb0 ⇒ Path planning feature is loaded
load yuhejfl_znjmvuc ⇒ Localisation feature is loaded

Robot plans a route from the table back to its docking station

unload egfuvsc_ajpzcb0 ⇒ Path planning feature is unloaded
load ayxqgu_evtokr ⇒ Rolling feature is loaded
load umevdg_tarecd ⇒ Navigation feature is loaded

Robot navigates back to the docking station to wait for another request

unload umevdg_tarecd ⇒ Navigation feature is unloaded
unload yuhejfl_znjmvuc ⇒ Localisation feature is unloaded
unload ayxqgu_evtokr ⇒ Rolling feature is unloaded

Scenarios 2: Hospital Management With Model C
Instructions are issued concerning an area in the hospital that needs disinfection

load egfuvsc_ajpzcb0 ⇒ Path planning feature is loaded
load yuhejfl_znjmvuc ⇒ Localisation feature is loaded

Robot plans a disinfection route

unload egfuvsc_ajpzcb0 ⇒ Path planning feature is unloaded
load ayxqgu_evtokr ⇒ Rolling feature is loaded
load umevdg_tarecd ⇒ Navigation feature is loaded

Robot navigates to the infected area via the planned route
6. Evaluation

unload ayxqgu_evtokr => Rolling feature is unloaded
unload umevdg_tarecd => Navigation feature is unloaded
unload yuhejfl_znjmvuc => Localisation feature is unloaded
load jdhdif_ctdmtv => Object Detection feature is loaded

Upon arrival robot identifies open door

unload jdhdif_ctdmtv => Object Detection feature is unloaded
load ayxqgu_evtokr => Rolling feature is loaded
load umevdg_tarecd => Navigation feature is loaded

Robot navigates to starting position

load rlgkct_ocvflt => Prompter feature is loaded

Robot issues safety instructions to prompt any people in the room to exit the room

unload rlgkct_ocvflt => Prompter feature is unloaded
load gxhout_foubqr => UVC Emission feature is loaded

Robot navigates throughout the entire designated area while disinfecting simultaneously

unload rlgkct_ocvflt => Prompter feature is unloaded
unload gxhout_foubqr => UVC Emission feature is unloaded

Upon completion, robot sends prompt to operator

unload rlgkct_ocvflt => Prompter feature is unloaded
load egfuvsc_ajpzcbv => Path planning feature is loaded

Robot plans route back to docking station

unload egfuvsc_ajpzcbv => Path planning feature is unloaded

Robot navigates back to docking station

unload umevdg_tarecd => Navigation feature is unloaded
unload ayxqgu_evtokr => Rolling feature is unloaded
unload yuhejfl_znjmvuc => Localisation feature is unloaded
• **Scenario Execution, Analysis and Results:** With the aid of our guideline document we were able to load and unload the requisite features for each configuration to accomplish each given task. While executing the tasks in our scenarios, we kept track of the system feedback trail to confirm that we were getting the expected output.

### 6.3 Threats To Validity

Some identified threats to the validity of our work, with respect to the scientific contribution it offers to the robotics domain, are as follows:

- **Internal validity:**
  - Throughout our study, we liaised with domain experts in the field, on technical issues that influence the conceptualisation, design, implementation and maintenance of our proposed solution. These experts on the other hand have a strong interest in the study. Therefore, there might be some conflict of interest in their contributions. To mitigate this, domain expert consultation could be expanded to include a wide array of industry professionals that have little to no ties with this study.
  
  - The selection of example robotic systems for feature extraction and validation were tailored to fit a specific group of robots from specific manufacturers. This will introduce some form of selection bias in that regard. This could be further enhanced by including a wider variety of robotic systems from multiple manufacturers in our study.

- **External validity:**
  
  - A potential threat to the external validity of this study is the fact that our implementation is not tailored to any other robotics platform apart from ROS. To improve this, we could extend our study to cover other less popular middlewares like OROCOS and the likes.
7

Conclusion

This section highlights various discussion points gathered from our study. These discussion points are mostly overall observations at crucial points in our study. Here, we clearly explain these observations as part of conclusions drawn from the results we perceived. We also provide a summary of the study with insight on how this study can be extended in the near future.

Variability implies in its definition that variations in a product line context are anticipated. This presupposes that core asset developers have thought about the consequences of variations and, presumably, have implemented constraints on these variations, in a manner that enables core assets to support product requirements. By inference, anything that is used to create a product is considered to be a core asset. This could be the architecture and its documentation, budgets, schedules, plans, user manuals, test plans, tools, process definitions, analysis models, configuration management plans, interface specifications, and a myriad of other things [8]. Throughout our study, we have realised similar core assets specifically through design, implementation and evaluation. For that matter we conclude our study by providing a brief summary that recounts events that specifically led to the realisation of artifacts that in turn fulfil our research objectives.

Implementation:  The goal of our implementation was to realise an abstract syntax that strictly defines a mechanism where binding time and binding mode properties of a feature can be used to define variation points. In Figure 5.1 and 5.2 we clearly demonstrated the feasibility of our variability management technique through meta-models that represent model and configuration capabilities. Through these capabilities, feature instances at variation points can be adapted to fit multiple usage contexts and purposes.

Evaluation:  Evaluation of our work was done to prove four main things. i.e. the novelty of our variability management approach, the feasibility of using binding time and binding mode as variability mechanisms, the correctness of the configurations we are able to realise and the ability of end users to apply configurations derived from our framework to real life robotic scenarios. By virtue of our design and implementation success, our framework has proven to be novel and realisable. To ensure correctness we introduced a constraint checker that is able to parse and validate
user defined configurations based on key constraints we have enforced to obligate end users to create quality constraints. The quantification of errors injected in five different passes as shown in Table 6.1 proves that our constraint checker is able to catch all constraint errors related to inclusion, exclusion, parent/child dependency, and time-mode binding. Finally to prove that user defined configurations have coverage capabilities, we created and run a couple of configurations against realistic scenarios with match the operations of service robots to emphasis the fact our user defined configurations can truly be used to load and unload features in typical scenarios where robots are usually deployed.

**Results:** The results we have presented in this chapter are derived from configuration realisation, configuration validation and scenario realisation. Overall, we have designed a variability-modelling language that allows end users to model features together with their binding times and binding modes (*RQ 2*). We realised this design by clarifying relevant aspects that are key to the language domain, through domain analysis and meta-modeling. In effect, we were able to realise example systems like TIAGo in Figure 5.3 and Model C in Figure 5.4 (*RQ 1*).

To prove the correctness of our example systems and the quality of the our variability management language, we validated our examples based on multiple constraints we have enforced. After this, we went ahead to run our examples through likely scenarios that service robots tend to operate in. This was meant to evaluate the implementation techniques we used to realise variability in our language i.e. variability mechanisms.

Variability mechanisms typically comprise of techniques for modeling variability and techniques to implement variation points. As part of our implementation, we also provided guidelines in the form of a concise documentation to assist end users when modelling variability and implementing variation points (*RQ 3*).

### 7.1 Future Work

Generally, the practical and theoretical proof of feasibility, novelty and applicability this study provides, will serve as a very good reference point for future studies in the domain of variability management in robotic systems. This study has the potential to be extended in the direction of mission specification by investigating the possibility of using goal-orientated approaches to automatically trigger system re-configurations based on our proposed variability management technique.
Bibliography


Bibliography


of Waterloo, 2013.


Methodology Feature Models

Figure A.1: Phase 1: Iteration One Feature Model
Figure A.2: Phase 1: Iteration Two Feature Model
B

Guideline Document

B.1 Environment

- Ubuntu 20.04.3 LTS and above
- ROS Noetic 1.15.9
- Python 3.8.3

B.2 How To

B.2.1 Setup and Run SLED Command Line Tool

- Clone the ROS package from the https://github.com/jaythagod/ros_confapp
- Use roslaunch to run features.launch
- Navigate to the ros_confapp/src/ros_confapp/src/nodes directory and run the ROS node dslNode.py
B.2.2 Create and Customize an Example System Model with SLED

B.2.2.1 Create Example System

Use the command `create_default_config <project_name>` to create a new project. This instantiates a default model structure and configuration in the /usr/base directory of the package into your new project directory (Figure B.1). Likewise an entry is also made in the engine of the system (Figure B.2).
B.2.2.2 Show Project List

![List of Projects]

**Figure B.3:** List of Projects

![Project List Command]

**Figure B.4:** Project List Command

A project instance contains three files, a model file that contains the hierarchical representation of features in the robot, the config file contains different combinations of binding time and mode that can be customised, selected and deselected by end users through in-built DSL commands. As shown in Figure B.3.

The `ls` command can be used to list all projects in the system (Figure B.4). The currently active project is represented in the console output by a double asterix (**) identifier next to it.

B.2.2.3 Activate a Project

![Activating a Project]

**Figure B.5:** Activating a Project

Project instances can be activated with the `activate_config <project_name>` command. An activated project implies that all project specific commands you run subsequently will be executed on the currently activated project.
B.2.3 View Configuration

A configuration can be viewed with the show command. A configuration can be shown on three levels. show all, show <feature_name> and show config. Show all shows the entire configuration state, show feature_name shows a feature together with all its sub-features. show config only shows selected features excluding deselected features.

B.2.4 Add and Remove Features

To add a feature to the configuration, use the command add_feature <feature_name>to feature_id>. The feature name specified cannot contain spaces. When adding a feature, a registry mapping entry and a launch file node definition need to be specified as well.

Use the command remove <feature_id> to remove a feature from the model hierarchy.

B.2.4.1 Implement a Feature Node

```python
import rospy
from std_msgs.msg import String
```
nodeName = 'kmean'
topicName = "kmean"

def kmean_callback(data):
    rospy.loginfo("%s command published",data.data)
    if data.data == "unload":
        unloadThisFeature()
    elif data.data == "ping":
        runNode()

def unloadThisFeature():
    reason = "Kmean feature unloaded from configuration"
    rospy.loginfo(reason)
    rospy.signal_shutdown(reason)

def runNode():
    rospy.loginfo("Kmean feature executed")

def main():
    rospy.init_node(nodeName, disable_signals=True, anonymous=True)
    rospy.loginfo("Listening@kmean.node")
    rospy.Subscriber(topicName, String, kmean_callback)
    rospy.spin()

if __name__ == '__main__':
    main()

With the above specified template, you can implement a node after adding it to your model hierarchy. The values of variables nodeName and topicName must be the same and they must match the function of the node. The name of the node by convention should correspond to the node type and name in the features.launch file. After implementation, an corresponding node entry must be made in the features.launch file.

B.2.4.2 Add Launch File Entry

<node pkg="package_name" type="type" name="name" output="screen"/>

By convention, the type and name properties must match. Also, the feature_name property must match the value of the nodeName variable in the code implementation of the feature.

B.2.4.3 Add Registry Mapping Entry

{
    "fid":"feature_id",
    "endpoint":"ros_topic_name"
The \textit{fid} property of your entry must match the \textit{id} property of the created feature in the project \textit{model.json} file.

### B.2.5 Alter Binding Time and Mode Properties

![Figure B.10: Alter Binding Time and Mode](image)

Binding time and binding mode property values can be altered with the \textit{alter_feature} command. This command takes two parameters, i.e. mode and time. Mode can be either \textit{static} or \textit{dynamic} while time values can be set to \textit{early} or \textit{late}. These parameters can be passed to through the command as illustrated in Figure B.10.

### B.2.6 Set Constraints on Features

![Figure B.11: Set Include Constraint](image)

![Figure B.12: Set Exclude Constraint](image)

Two types of user defined constraints can be set using SLED. These are \textit{includes} and \textit{excludes}. An include command takes the form \textit{set\_include} whereas an exclude is represented by \textit{set\_exclude}. Both commands take two feature IDs as parameters. The first represents the feature the constraint is being set on and the second represents the feature that is either being excluded or included.

### B.2.7 Configure an Example System Model with SLED

![Figure B.13: Select Command](image)

![Figure B.14: Toggle Command](image)
B.2.8 Validate a Configuration with SLED

A configuration can be validated by checking three main constraint violations. Which are inclusion, exclusion and parent/child constraint violations. The command `validate_config` checks each defined feature against all three constraints and makes a record of the points in the configuration where the constraints have been violated as shown in Figure B.15.

B.2.9 Load and Unload Features from a Running Configuration

- **Run Configuration**: To run a configuration, first, the `features.launch` file must be executed to activate all node definitions and parameters on the parameter server. Running a configuration implies that the customised model hierarchy of features will be bound according to their binding times and the
state of the application will be programmatically flipped from an early binding time to a late binding time. To run the configuration use the `run_config` command.

- **Dynamically Load/Unload Feature from Configuration:** To load or unload a feature from a running configuration, the `load` and `unload` commands can be used. Every time a feature is loaded or unloaded from a configuration, the parameters on the parameter server are refreshed to reflect the change.

- **View Parameters on Parameter Server:**

  The command `dump_server_settings` is used to dump all the parameter set on the parameter server. These parameters include the current binding time as well as selected features in the currently active configuration.

  ![Dump Server Settings](image)

  **Figure B.16:** Dump Server Settings

X