

CYCLICL - AN APPROACH TO
AUTOMATING REQUIREMENT
TRACEABILITY IN PRODUCT LINE
ENGINEERING

By

THOMAS CHIANG

A Thesis

Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements for the Degree

PhD

McMaster University

© Copyright by Thomas Chiang, June, 2025

PHD (June, 2025) McMaster University
(Software Engineering) Hamilton, Ontario

TITLE: CyclicL - An Approach to Automating Requirement
 Traceability in Product Line Engineering

AUTHOR: Thomas Chiang, B.Eng Computer Engineering, M.A.Sc
 Software Engineering (McMaster University)

SUPERVISORS: Richard Paige, Alan Wassynge

NUMBER OF PAGES: [viii](#), 160

Lay Abstract

The purpose of this paper is to

Abstract

Product Line Engineering (PLE) has become a common approach in engineering in the context managing product complexity and reuse across multiple industries. The practice is used for identifying reusable pieces of code, composing software components, and modelling product variation. There exists a gap however with generating system traceability when using PLE. Specifically, it is a challenging to generate traceability from a feature model through requirements to design elements. Further, traceability itself is both a costly and tedious task that is often completed at the end of development in order to meet certification standards or for assurance purposes. At the same time, supporting traceability in a PLE context is difficult as products go through their life-cycle. This thesis proposes a methodology to improve the identification of features through cro:MDEModel-Driven Engineering (MDE) techniques, define a hierarchical relationship between features and requirements, and provide a tool supported approach to generating and maintaining traceability in a PLE context throughout a product life-cycle. The contributions of this thesis will attempt to address the gap that exists between cro:PLEProduct Line Engineering (PLE) and requirement engineering to migrate traceability out of a retrospective task towards a more active portion of development.

Acknowledgments

An expression of thanks to supervisors, industry partners, colleagues, family, or friends.

Contents

Descriptive Note	ii
Lay Abstract	iii
Abstract	iv
Acknowledgments	v
Table of Contents	viii
List of Figures	ix
List of Tables	xii
List of Acronyms	xiii
Declaration of Academic Achievement	xiv
1 Introduction	1
1.1 Motivation	6
1.2 Hypothesis	8
1.3 Organization of Thesis	10
2 Literature Review and Definitions	12
2.1 Inclusion Criteria	14
2.2 Product Line Engineering	15
2.3 Requirement Engineering and Modelling	17
2.4 Traceability	20
2.5 Discussion	22

3 Methodology	24
3.1 Domain Analysis	29
3.1.1 Goals	31
3.1.2 Use Cases	34
3.2 Features	38
3.3 Goal Refinement and User Stories	42
3.4 Requirement Modelling and Feature Modelling	45
3.5 Feature-Requirement Traceability	52
3.6 Requirement Specification	54
3.7 Requirement Based Feature Identification	55
4 Implementation	58
4.1 Implementation Objectives	59
4.1.1 Goals and Limitations	64
4.1.2 Requirements	65
4.2 Abstract Syntax	67
4.3 Concrete Syntax	70
4.4 Design Decisions	73
4.4.1 Composition Implementations	74
5 Evaluation	76
5.1 Knowledge Capture & Domain Analysis	77
5.1.1 Goal Diagrams	78
5.1.2 Use Case Diagrams	85
5.2 Product Family Development	96
5.2.1 Composition	97
5.3 Requirement Engineering and Refinement	97
5.4 Iterations and Incrementation	104
6 Future Work	107
7 Conclusion	110
Appendices	114

A	115
A.1 Boston Scientific Pacemaker Requirements	115
A.2 CyclicL Complete Metamodel	151

List of Figures

3.1	List of steps from the Meyers Handbook for requirement engineering.	25
3.2	Legend of goal diagram elements.	33
3.3	Example goal diagram outlining the goals of a driver using a vehicle.	33
3.4	A use case diagram outlining what parts of the vehicle a driver will use.	36
3.5	cro:UCDUse Case Diagram (UCD) scoped by the loading cargo goals	37
3.6	Initial attempt at a feature model based on use cases identified in UCD.	41
3.7	Product variants for the initial vehicle feature model. Model is simplified to just the optional features that are mandatory for the variant.	43
3.8	CyclicL metamodel. Shows the specification for the feature modelling portion of CyclicL.	48
3.9	CyclicL metamodel. Shows the specification for the first half of the requirement canvas portion of CyclicL.	49
3.10	CyclicL metamodel. Shows the specification for the second half of the requirement canvas portion of CyclicL.	50
3.11	CyclicL metamodel. Shows the specification relationship between the requirement canvas and feature modelling portions of the metamodel. Also shows their relation to the model root.	50
3.12	Example of how a feature can be opened up to display encapsulated requirements.	54

3.13 Simplified example of an adaptive cruise control feature model	56
4.1 Definition of the relationship between features and requirements.	63
4.2 Example of a requirement canvas concrete syntax. The example uses a disposable coffee cup with a handle as the focus of the diagram.	63
4.3 Anticipated goals of users for CyclicL.	65
4.4 An example of the concrete syntax for the requirement canvas using requirements from the automotive domain.	71
4.5 An example of the concrete syntax for the feature model using features for an adaptive cruise control system from the automotive domain.	72
4.6 Gherkin specification for Wheel Heating on/off functional requirement. The high-level requirement description is: User shall be able to turn wheel heating on/off.	73
5.1 Pacemaker lifecycle as outlined by Boston Scientific requirements.	78
5.2 Doctor goals in the context of the pacemaker system.	81
5.3 Patient goals in the context of the pacemaker system.	82
5.4 Hospital goals in the context of the pacemaker system.	83
5.5 Nurse goals in the context of the pacemaker system.	83
5.6 Patient family goals in the context of the pacemaker system.	84
5.7 Technician goals in the context of the pacemaker system.	84
5.8 Combined nurse and doctor goals.	85
5.9 Use Case Diagram for the PG device based on the original pacemaker specification.	89
5.10 Intended operation modes for the pacemaker system.	91
5.11 Use Case Diagram for the PG device based on the original pacemaker specification enhanced with information from the goal diagram knowledge capture.	91
5.12 Use Case Diagram for the DCM based on the original pacemaker specification. This diagram combines the hardware and software sub-systems into a single boundary.	92
5.13 Use Case Diagram for the leads based on the original pacemaker specification.	94

5.14	Feature model based on the UCD for the PG_Device.	95
5.15	When right-clicking on a feature, a context menu appears allowing users to create a requirement canvas that is embedded in the selected feature.	99
5.16	Requirements for the ‘Provide programmable pacing’ use case. .	100
5.17	Gherkin specification for the first requirement of the PG_Device.	101
5.18	Gherkin specification for the second requirement of the PG_Device.	101
5.19	Gherkin specification for the third requirement of the PG_Device.	102
5.20	Various stages of approval and testing for the requirements. . . .	103
5.21	Showing how traceability is automatically maintained throughout requirement approval and testing.	103
A.1	CyclicL metamodel. Shows the specification for both the requirement canvas and feature modelling portions of CyclicL. . .	151

List of Tables

List of Acronyms

MDE Model-Driven Engineering

EMF Eclipse Modeling Framework

DSL Domain Specific Language

PLE Product Line Engineering

FDD Feature-Driven Development

FODA Feature-Oriented Domain Analysis

FORM Feature-Oriented Reuse Method

UCD Use Case Diagram

AQL Acceleo Querying Language

EVL Eclipse Validation Language

Declaration of Academic Achievement

The student will declare his/her research contribution and, as appropriate, those of colleagues or other contributors to the contents of the thesis.

Chapter 1

Introduction

What came first, the chicken or the egg? It is a silly question that people will often debate about over dinner or on the bus. In engineering there is a similar debate; what comes first, the feature or the requirement? Very often when we do engineering work we consider what we want to build in parallel to actually thinking about building it; exploring the problem space often gets overlooked in favor of exploring the solution space. As a result, an engineer may have to reflect on their requirements when they hit a wall during development. They may find that their requirements were incomplete for their feature. Perhaps the scope of their feature is much bigger than they anticipated; maybe what was originally considered to be a single feature could in fact be multiple features. Thus, the engineer will iterate back and forth between feature development and requirement development, incrementally changing each until they get to a system state that they consider to be complete, or at the very least a minimum viable product.

How do we capture this process? Feature-Driven Development (FDD) [1] is a common method for agile development whereby the engineers

will identify a set of features that will be built together to create a system, often leveraging MDE techniques as well. They will then work to identify acceptance criteria, or descriptions of each feature to specify what needs to be done for each feature and how to know when it is complete. These descriptions end up serving a similar role as requirements for the features, if not outright being written as requirements for the feature. Therefore, we can make the proposition that before we have requirements, we have features. The features come first, thus we should focus on identifying features before we start to think about what requirements we need for a given system. This situation will often arise when a new product is being developed, or a company is just starting up.

There is an unusual assumption made in this scenario; are there not already existing requirements before we begin to identify features? From another perspective, in more mature development teams and industries it is uncommon to start from scratch without any requirements, and even more rare to jump right into identifying solutions without first having an idea of what the problems are. Thus, even if limited, there are at least some guiding requirements before engineers begin to identify features of their system that will solve their problems. Therefore, we can say that requirement come before the features. In this environment, we already have a rich history to pull from to help inform any new development that may come up.

In reality, both scenarios are likely to happen, perhaps even within the same company! Innovation in industry is constant and there are always new products being developed both from scratch or iteratively on existing products. We may be able to pull from existing assets from one product to help with the development of an entirely new product. Conversely, we may be branching off into an entirely new domain with no existing documentation or reusable

assets. Thus we have to start from scratch at understanding what problems exist within the new domain. Both scenarios are equally valid!

The conflicting nature of reality generates a couple of problems for development. The relationship between requirements and features becomes ambiguous. In fact, what are the differentiating factors between a feature and requirement? For this thesis, we will follow the definition of a feature from the original work of Kuang et al. [2, 3] for their Feature-Oriented Domain Analysis cro:FODAFeature-Oriented Domain Analysis (FODA) and Feature-Oriented Reuse Method cro:FORMFeature-Oriented Reuse Method (FORM). According to their original definition, a feature is ”a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems.” Based on this definition, we could consider that any portion of a system that a user can interface with is viable to become a feature. In fact for the sake of inclusion, we extend this definition to not only be user-visible, but simply an interface, taking into account the multiple different senses that a user can use to interact with our system. This will increase the scope of a feature beyond what is only visible, including sound, touch, smell, and taste. Depending on product scope, this definition could be extended to other subsystems that interface with the targeted product.

What is a requirement? This is well developed and studied topic. To answer, we look to van Lamsweerde [4] where he outlines the purpose of requirements engineering is to answer three question; what, why, and who? What is the problem we are trying to solve? Why is it worth solving? Who should be involved in the solution? There are many more requirements engineering approaches and strategies that we can and will use to develop our methodology, however as a base line when understanding what a requirement is it should

answer what the problem is that we are solving supported by sound reasoning for why it is worth being solved and who is involved in solving the problem.

As such, what are the problems we are trying to solve with this thesis? For one, what is the relationship between features and requirements? This is a problem due to ambiguous relationship between them and the different development styles that exist. As there are dependencies between them, setting a definition of the relationship between them may help with both making sure that we elicit and refine requirements that are correct and reasonably complete. At the same time, it can help to ensure that the features we identify are correct and reasonably complete.

Another issue is that iterations between features and requirements do not happen in isolation. There is usually a lot of supporting documentation and software that surrounds and depends on both the requirements and identified features. This is usually expressed through traceability documentation, most commonly in a traceability matrix. However, maintaining a traceability matrix with all the incremental changes that happen between requirements and features is an extremely tedious and time consuming task. For each change it is usually a low return for high effort task, until such a time that the traceability matrix is so out of sync with both the requirements and features that it is no longer usable. One implication of this is that we will need to create an entirely new traceability matrix and thus the cycle continues. Therefore, traceability documentation is a very large problem during iterative and incremental development.

We also examine implication for agile development styles. FDD is a feature focused development methodology that already has some set rules for implementation. We also want to explore some possibilities that may exist for

adoption into existing development approaches. The objective here would be to explore methods for improvement rather than redefining entirely how FDD might be implemented, especially tying it in with MDE and PLE efforts. This is true both at the conceptual level and the tool support level.

Tools like FeatureIDE [5, 6] which supports feature composition of software artifacts. Or a tool like GEARS [7] which emphasizes its 3-tiered software product line methodology. There is also formal methods research for PLE such as the work of Peter Höfner et al. [8, 9], which created a formal algebra for how to compute product lines and feature models. What these tools lack however is an self-contained way to manage traceability between features and requirements; FeatureIDE focuses on code composition where GEARS is not cohesive. GEARS requires hooking into separate requirements engineering tools and does not have a way to handle requirements independently. Thus we identify a lack of tool support for traceability and iterations between requirements and features as another problem.

Finally, the problem of how we accomplish all of these tasks? What is the process to help make sure the features we identify are correct and sufficiently complete? What is the process to make sure that we identify requirements that are correct and sufficiently complete? How can we be sure that our mappings between requirements and features also make sense? This is another problem as we want to make sure that the mappings between features and requirements are correct and complete for development tasks as much as for assurance purposes. We want developers to be developing the correct features, and we want to be capable of reasoning about those features so that we can develop assurance that properties of our system are true.

It is important to note that while not mandatory, many PLE efforts also

use modelling in their approach. There is a lot of cross-pollination that already exists between the two domains. For our purposes, we want to examine these problems through the lens of MDE primarily; can we solve these problems using MDE techniques and processes? We support PLE through MDE techniques and processes rather than using modelling as a means to an end.

In summary, these problems lead to the following research questions:

RQ.1 How can we improve the state of implementation and usability of FDD?

RQ.2 How can we improve the maintenance and generation of traceability between features and requirements?

RQ.3 How can we improve tool support for iterative and incremental development between features and requirements?

RQ.4 How can we leverage MDE techniques for domain analysis, problem space exploration, and requirement development?

1.1 Motivation

In many industries, we can find companies that have a catalog of products they offer to customers. There are also many industries that focus on safety-critical application development. Some areas where these two categories overlap are the automotive domain and the medical device domain. Within automotive industry, companies offer a range of vehicles customers can choose for purchase. Further, each of the vehicle models on offer can have variants available. These vehicle model variations can be due to aesthetic difference or functional differences. There can also be variations due to where in the world the vehicle

is being sold, such as the driver seat location depending on if the country of sale drives on the left or right side of the road.

Further, vehicles are inherently dangerous products. According to the Canadian government, there were a total of 91533 vehicle collisions reported in 2022, resulting in 1931 total fatalities [10]. It is well documented that automotive manufacturing is a safety-critical industry between which is why the companies involved are constantly innovating and evolving to improve the safety of vehicles to reduce accidents, fatalities, and injuries. Thus, we can see that the automotive industry is one that can benefit from PLE to help with managing the product catalog, aid in managing documentation to develop safety cases (or assurance cases in general), and support iterative and incremental development as they release new vehicles year after year.

For medical device development, product variant also exist as companies may be required to fine tune or adapt a product to different country regulations. Another point of variation can come from the intended use case of a medical device. The same device may be used to treat different diseases or ailments under different conditions and enabling or disabling this kind of functionality can be critical for regulatory approval. It can also be part of the discovery process as a new application for existing technology can be discovered. This event occurs all the time in the medicine and medical technologies. The ability to plan out new product variants can be extremely beneficial for product planning and development

Throughout this thesis we will be using examples from the automotive domain. These examples will aim to help provide context for the work and aid in the evaluation of the methodology and tooling. Then we use an example pacemaker project in the evaluation section of this thesis to examine the pro-

posed methodology side by side with an existing requirement document for a medical device.

1.2 Hypothesis

With the research questions defined and the motivation outlined, we can start to take some guesses at how to answer them. Beginning with **RQ.4**, there are several modelling techniques to use. For the domain analysis and identification of system features, we propose the use of Use Case Diagram (UCD) from SysML [11]. As a modelling technique it is informal, however it is also easy to use for analysis and problem space exploration. The biggest reason to use UCDs is the requirement to identify actors/stakeholders of the system and how they might use the system. The idea of a use case is very similar to what a feature of a system is and thus allows for a relatively easy mapping between system use cases and features.

Another MDE technique we use for domain analysis and problem space exploration is Goal Diagrams as outlined in Van Lamsweerde's requirements engineering textbook [4]. Lamsweerde has formal semantics defined for goal diagrams which helps with making sure that we can properly reason about the goals of the system, as well as the goals of the users. However, in spite of the formal semantics Lamsweerde has prescribed to goal diagrams, there are variant syntaxes that exist for goal diagrams that do not strictly adhere to the syntax and thus the semantics that Lamsweerde has defined. This has pros and cons. As a benefit, the reduction of formality can lower the barrier for entry, thus making it easier to use across professions as back of the envelope forms of expression. The loss of semantics presents difficulty to use formal

validation goals and the possible requirements they can be used to derive. Given that we are more concerned at this stage with capturing knowledge, we are not concerned about the loss of formality. We found the increase in flexibility more valuable as it helps with the usability of goal diagrams and their use for exploring the problem space to elicit requirements.

The reason we chose these two MDE techniques is part of the answer to **RQ.1**. For the rest, we turn to the Handbook of Requirements and Business Analysis by Meyer [12]. In his book, Meyer outlines several steps towards the requirement elicitation process, which also apply themselves to feature identification. Parts of his book outline the importance of identifying goals of both the system and the users, along with identifying the importance of user stories and use cases. Further, he supports the use of both formal and informal methods, leaving it up to the engineer to decide when it is necessary to use one over the other based on context. We support that notion, and much of the methodology we outline is inspired from his book. The UCDs created for analysis will also be used to outline user stories to refine and justify the identified features. The goal diagrams from van Lamsweerde’s requirements engineering are used to support the goals book from Meyer in either a formal or informal capacity based on the engineer’s discretion.

For **RQ.2**, we propose the following hierarchy: features shall encapsulate requirements. The reason for this proposal is two-fold. For one, there are many more semantics around feature modelling and PLE compared to requirement modelling. And for the second point, in FDD we often list requirements as scoped by a feature. This presented a history supporting the concept of a feature containing requirements. As FDD is a common approach to development, we felt this would ease the conceptual learning curve of the proposed hierarchy

and its implications. By having features encapsulate requirements, we can get feature-scoped requirement traceability as every requirement will be owned by a feature. We will henceforth refer to this as feature-requirement traceability.

This also helps with **RQ.3** as the encapsulation will aid traceability efforts. By formalizing the relationship between features and requirements, tool development becomes streamlined as we can leverage existing Object-Oriented techniques to develop a cro:DSLDomain Specific Language (DSL) to support iterative and incremental development of both features and requirements. We can leverage a tool that is self-contained to attempt to partially automate traceability maintenance when making changes to either features or requirements in either a feature model or requirement model.

In summary, the hypothesis is that we propose that we let features encapsulate requirements for supporting traceability. Leveraging this definition we can build a tool that supports partial automation of feature-requirement traceability. We can use existing MDE techniques in UCDs and goal diagram for domain and problem space analysis inspired by Bertrand Meyer’s style of requirements engineering.

1.3 Organization of Thesis

The thesis is structured as follows:

Chapter 1: The introduction where we discuss motivations for this thesis, research questions, and the hypothesis.

Chapter 2: The literature review for this thesis where we examine previous work and supporting efforts for traceability in PLE.

Chapter 3: The definition and explanation of the proposed methodology for this thesis.

Chapter 4: The implementation of the propose methodology which introduces the tool, CyclicL, to support modelling, specification, and traceability efforts.

Chapter 5: The evaluation of the proposed methodology. Done with a requirement specification for a pacemaker. Analyzes the value of the proposed methodology in the medical device domain.

Chapter 6: The future work identified for this methodology.

Chapter 7: The conclusion for this thesis.

Chapter 2

Literature Review and Definitions

There are four main topics of research that support the work in this thesis: MDE, PLE, traceability, and requirement engineering. Individually, each of these topics have many branches of research, both academic and industrial. In order to narrow down the selection of work to support this thesis we aimed to find papers that explored at least two of the four topics. As bonus topics, we also considered papers if they involved a supporting tool or an exploration of adoption in industry.

We also include some definitions in this chapter to establish nomenclature that will be used within the rest of this thesis. For PLE, we consider a product line, product family, and feature model to be synonymous ways of capturing product information by connecting features together. Similarly, we consider model-based system engineering and model driven engineering as synonymous practices.

Traceability is scoped to requirement traceability and feature traceability,

though not necessarily to each other. Much of the literature is focused on traceability itself with a smaller library of traceability between specific phases of development. As such we look for previous work that explores a variety of traceability methodologies and approaches. With these guiding topics we use the following strings to generate our search results:

- (“model driven engineering” OR “model based engineering”) AND (“product line engineering” OR “feature modelling”) AND “tools”
- (“model driven engineering” OR “model based engineering”) AND “traceability” AND “tools”
- “traceability” AND (“product line engineering” OR “feature modelling”) AND “tools”
- “traceability” AND (“requirement modelling” OR “requirement diagrams”) AND “tools”

Results using these search strings were chosen from the initial search results from Google Scholar and the following conferences:

- International Conference on Software Engineering (ICSE).
- International Conference on Model Driven Engineering Languages and Systems (MODELS).
- Software Product Line Conference (SPLC).
- Variability Modelling of Software-Intensive Systems (VaMoS).

2.1 Inclusion Criteria

In order to deem a resulting publication to be relevant to this body of work the following criteria needed to be met:

- The publication should use MDE techniques AND address PLE OR requirement engineering. The publication can focus on development in any applied domain, though of particular interest are automotive and medical device development.
 - Of particular interest are applications of feature modelling in industry.
 - Of particular interest are application of MDE techniques for requirement engineering and modelling.
- The publication should explore traceability. This can be done either for requirements OR PLE though ideally the publication should explore traceability explicitly between PLE and requirements. This also includes methodologies for automated traceability.
- The publication should have tool support for traceability, PLE, or/and requirement modelling.

We parse the results and separate them into the three domains; PLE, traceability, and requirement engineering. Though we setup MDE as one of the topics for this thesis, it is more of a generalized one as we explore PLE, traceability, and requirement engineering through the lens of MDE. MDE is also used to generate tool support for the task of feature modelling, traceability, and requirement engineering.

Another reason for separating the results in this manner is to identify synonymous definitions and specifications across domains. Looking at the various topics through the lens of MDE allows us to find patterns of similarity across the various publication.

As part of this work we explore two domains; the automotive domain and medical device domain. We want to show that the proposed methodology and implementation can be effective in those two domains as they have safety-critical requirements. As such we also include some literature for development in the automotive industry and medical devices, specifically pacemakers for the latter. For the supporting literature however we were more broad and looked across multiple industries to see what the state of the practice is for PLE, traceability, and requirement engineering.

2.2 Product Line Engineering

PLE has become a popular approach to handling diversity and complexity in a companies product portfolio since its original conception by Kang et al. [2, 13]. We scoped the related work to some fundamentals of PLE and some concrete implementations (tool support). As one of the objectives for this thesis is to add little extensions to the existing theories of product families we paid special attention to formal specifications and existing tools.

Many tools exist for feature modelling. Two examples of tools come in FeatureIDE and GEARS. FeatureIDE [5, 6], developed originally by Kästner and Thüm et al, is a feature modelling plugin tool for Eclipse that lets a user define their product software as a feature model, write code within the model, and seamlessly integrate the features together to form an executable product. It

has more recently become the research community standard as a feature modelling tool. Another example for feature modelling and PLE is GEARS [7] by Big Lever, which emphasizes its 3-tiered software product line methodology. The 3-tiered approach outlines three different ways to use GEARS: handling variation and automation within a single product line, an approach to identifying reusable assets across product lines, and product evolution over time.

As a formal approach to PLE, feature modelling can also be defined through algebraic methods as shown in the work of Peter Höfner et al. [8, 9] which uses set theory as the basis for proving how their algebra works. This algebra can be extremely useful for tool development that implements feature modelling as it allows for the computability of feature models, thus allowing for easier implementations of feature modelling and product line engineering tools.

Another direction that feature modelling has taken can be found in the work of Czarnecki et al. [14]. While still involving formalisms for the development of feature models, there is more focus on the graphical modelling of feature models as opposed to mathematical specification and computability. They expand on the original syntax defined in FODA and show by using context-free grammar (in this case metamodeling) how to add cardinality to feature models and their relevance to feature modelling paradigms. This specification’s main benefit is handling feature duplication in contrast to the work of Höfner et al.

We know that there is a desire for better PLE practices for industry. In a survey paper by Horcas et al. [15] they conclude a lack of tool support for the various approaches to PLE as roadblock to more ubiquitous adoption in industry. This is further supported by the workshop conducted by Becker et al. [16] where they identified 7 themes to the challenges of PLE across var-

ious industries: Modularity, Product Line Scoping, Variability Management, General Process, Verification and Validation, Evolution and Maintenance, and Key Performance Indicators. There is some ongoing work to address some of these issues. Bilic et al. [17] present an MDE tool chain and methodology to manage variability in system design. Co-evolution is another problem that Jongeling et al [18] present a solution for in Simulink models in a product line. For the automotive industry specifically, we can see some initial benefits based on an implementation by Wozniak et al. [19] and their usage of GEARS. Their act of capturing their product line in a consistent model based environment was helpful for managing complexity. The ability to automate configuration calibrations were a great benefit for their purposes as well. As these were just their initial efforts, they also called for more research and work to be done in this domain with respect to industry practices. These publications highlight that a gap exists between academic and industrial practices.

2.3 Requirement Engineering and Modelling

We consider requirement engineering as the act of eliciting, refining, and specifying requirements for a given item, be it a product, feature, or component. However, we found a gap in the literature with respect to MDE techniques and requirement engineering, specifically around requirement modelling. This will be a significant contribution of this thesis as we explore this gap and identify how MDE techniques can enhance existing requirement engineering approaches.

We also explore informal and formal approaches to requirement engineering. They both have merits in different situations, with informal methods

being more approachable, while the formal methods inspire more confidence due to their natural rigour. We also found various definitions for functional and non-functional requirements that were helpful for informing our own design decisions about how we approach requirements in this thesis.

The idea of using feature models to encapsulate requirements was partially inspired by the requirement engineering book by Bertrand Meyer [12]. His style emphasizes a lot of stakeholder analysis for determining their use cases and scoping system requirements around user interactions. It emphasizes a more informal approach to requirement engineering, lowering the bar for entry and making it more approachable, while recommending the occasional “formal picnic” as he refers to it as needed. The purpose of the “formal picnic” is to supplement requirement decisions and specifications with more rigorous methodologies as needed, rather than mandating formal methods be adhered to throughout the requirement engineering process.

We also have Axel Van Lamsweerde’s style of requirements [4]. There is a lot of overlap that exists between Meyer requirements and Van Lamsweerde requirements, however, there exist some key differences in the ease of implementation of the requirements processes. Van Lamsweerde’s requirement style dives deeper into the formalism and MDE approaches to requirement engineering, in contrast to Meyer’s requirement style. Van Lamsweerde used formal goal diagrams as a way to capture knowledge about the stakeholders of a product and to justify the emerging requirements from the goals. Another difference is in the definition of a goal. Van Lamsweerde defines a goal as a “prescriptive statement of intent that the system should satisfy through the cooperation of its agents.” In contrast, a goal is defined by Meyer as the “needs of the target organization, which the system will address”. To generalize, Van

Lamsweerde’s style requirements are a more in-depth version of Meyer’s requirements due to the more formal approach to requirement engineering, at the cost of the approachability found in Meyer’s style of requirements.

Due to the gap we discovered in requirement modelling, we focused many of our efforts on other requirement engineering resources that did not include MDE as a main component of their approach. Suzanne and James Robertson [20, 21], heavily emphasize stakeholder identification and involvement early on in the project. There is more emphasis on business use cases and functional vs. non-functional requirements in contrast to Meyer’s requirements which focuses more on the use cases for requirements concerning stakeholders. Meyer’s requirements style is also structured with four ‘books’ and each book represents a different portion of the requirement engineering process, each with a suggested structure for organizing their respective section. By contrast, Robertson’s requirement engineering approach focuses more on how requirements are written, and their specific categorizations. The requirement engineering process of Sawyer and Sommerville [22, 23] discusses the different viewpoints within system requirements. Their work stresses that not all stakeholders are people, and could be anything from an organization to another system to be integrated/interacted with. There are many more that could be talked about, however, we can generalize a pattern:

- Identify who/what you are developing for.
- Identify what/how the stakeholder interacts with a target system.
- Specify the behaviour around that interaction.

While requirement diagrams from SysML [11] are a helpful approach for requirements engineering and modelling, there are other modelling schemes

that have been explored in this topic. Use case diagrams from UML [24], have been explored by many people for their value in requirements engineering. Siau and Lee ultimately concluded that use case diagrams helped communicate requirements compared to other modelling [25], but have limited usability beyond communication. They have also been used by von der Maßen and Horst Lichter [26] for software product line development, having to customize the metamodel to make it work. Wegmann and Genilloud [27] formalized use cases and use case diagrams for functional requirements for systems, though they faced difficulty in scaling the model to handle more complex systems in detail and demonstrate limitations for use case diagrams. FORML, a Feature Oriented Requirement Modelling Language by Joanne Atlee and colleagues [28, 29], is another approach to feature scoped requirement based on product families. They use superimposed state machines to express behavioral requirements of the features that compose a product line.

2.4 Traceability

Traceability is a commonly studied topic in academia and industry. We found that traceability was commonly considered a retrospective task of product development due to the difficulty and cost in maintaining it throughout development. We found this is due to a common philosophy of separating traceability as a task from the development process instead of including as a natural outcome of development. This has led to a variety of post-release analysis techniques to build traceability artifacts.

As this thesis aims to produce traceability as a natural part of development instead of a retrospective task, we explored various ways of both automated

and manual approaches to traceability. We kept the scope of traceability artifacts broad to help explore traceability throughout the phases of development beyond the feature and requirement phases.

The concept behind this paper shares some similarities to the work done by Marques et al. [30]. While we both aim to automate traceability matrix generation and maintenance by using requirements diagrams, there are distinct differences in our implementations and end goals. We focused our efforts into distinct MDE environment (requirement modelling, feature modelling, use case diagrams, and goal diagrams). Meanwhile, the approach from Marques et al. aims to generate traceability between requirement, design, and testing.

Dronology, the work of Cleland-Huang and colleagues, is a project that has produced comprehensive work around and including traceability. As part of their project they explore traceability through the lens of safety assurance [31, 32, 33, 34].

The work of Heisig et al [35] proposes a generic traceability metamodel for end-to-end traceability in software product lines. Their proposed method is supposed to facilitate and enable comprehensive traceability throughout the development process from requirements to implementation across multiple MDE tools. As such they developed a traceability metamodel to handle a variety of artifacts they anticipate will require traceability links. In contrast, our work does not focus on developing a traceability metamodel. Instead, we focus on mapping requirements to features and generating traceability from the mappings. Another example of traceability in product lines is in the work of Tsuchiya et al. [36]. Their work focuses on recovering lost traceability links due to incremental development. They propose a framework and implement a tool that shows potential in recovering lost traceability links due to natural software

maintenance over time. This is relevant as it is another approach to traceability maintenance through incremental software changes. Where they focus on recovering lost links, we attempt to update the links parallel to changes.

Similarly, the work of Kelleher [37] also required the definition of a traceability metamodel, though this was developed as a means to handle complexity in software systems as opposed to product lines. Another approach can be found in the work of Maletic et al [38], whose primary focus was flexibility and interoperability of traceability artifacts, allowing for the direct modelling of artifacts through model transformations using XML. Asuncion et al. [39] used machine learning to generate traceability prospectively and retrospectively; during development and after development.

2.5 Discussion

Having gone through a variety of publications exploring MDE, PLE, requirement engineering, and traceability we notice a few patterns. While there are some MDE techniques that are used to help with requirement engineering, both formal and informal, there is a lack of literature about modelling requirements themselves. Specifically, the use of requirement diagrams from SysML is an under-explored theme in comparison to more popular modelling strategies like UCDs, and to a lesser extent goal diagrams.

There are many different strategies for handling traceability, from machine learning to manual processes. We also found some previous work for generating traceability in a software product line. We found that we share a similar philosophy to Asuncion et al. that traceability should be a prospective task rather than a retrospective one. While we share that philosophy, we approach

it very differently as we do not use machine learning as part of our traceability generation. We consider traceability should be a natural outcome of good engineering, emerging naturally as various phases of development are completed.

We generally agree with Meyer’s approach to requirement engineering in supporting a more informal and flexible approach to requirements, with some detours into more rigorous methods to supplement and support requirement specification as needed. This is the main reason why we follow a similar development path that he presents in his handbook.

Finally, PLE and industry require more support for state of the practice. We’ve identified across multiple industries a desire for better PLE in industry and many of the difficulties they face with implementing it. While we do not aim to solve all of them in this thesis, we do aim to lower the bar for entry in implementing PLE techniques for industry, and creating tools to help support these efforts.

Chapter 3

Methodology

In this chapter we explain the how we combine Model-Driven Engineering (MDE) technique with Product Line Engineering (PLE) techniques to identify features and requirements. This is then followed by establishing the relationship between features and requirements into a hierarchy to facilitate a well defined Feature-Driven Development (FDD) practice. Parts of this methodology were inspired by significant overlap between these seemingly very different domains of engineering practice. In the world of MDE, Use Case Diagrams (UCDs) are often used to capture high-level knowledge for how stakeholders will interact with certain parts of a system. While the modelling approach is relatively informal in its semantics and syntax [24, 4], it has been claimed to support both technical and non-technical users alike in describing how they want stakeholders and customers to interact with their products or system.

The overall methodology for this thesis is heavily inspired by the Goals and System books from Meyer’s requirement engineering book [12]. The Goals book is primarily focused on capturing conceptual models of the stakeholders

Goals (G)
G.1 Context and overall objective
G.2 Current situation
G.3 Expected benefits
G.4 Functionality overview
G.5 High-level usage scenarios
G.6 Limitations and exclusions
G.7 Stakeholders and requirements sources
System (S)
S.1 Components
S.2 Functionality
S.3 Interfaces
S.4 Detailed usage scenarios
S.5 Prioritization
S.6 Verification and acceptance criteria

Figure 3.1: List of steps from the Meyers Handbook for requirement engineering.

of the desired product and the purpose of building it. The System book is essentially a refinement of the concepts captured in the Goals book to actionable requirements ready for design and development. It dives into more of the details of stakeholder interaction, seeking to answer how to use a product while the Goals book seeks to answer question more focused on why the product should exist and what it will do for the stakeholders. We do not completely implement everything outlined in those books and instead have adapted them to suit our purposed. The steps of our proposed methodology are as follows:

1. Identify stakeholders, users, and customers. Who will be impacted by the product?
2. Identify stakeholder, user, and customer goals. Why would they need the product we are developing? Why would they not need the product in question?

3. Identify stakeholder, user, and customer use cases for a designated system. Each use case should work to satisfy at least one goal. What will the product do to help the users and stakeholders?
4. Identify system features based on use cases. These are the high-level features for our system. This is where the overlap between features and use cases can be leveraged to equate use cases from ~~cro:UML~~**UML!** (**UML!**) to features in PLE
5. Refine use cases to user stories. These become our feature requirements.
6. Support stories with identified stakeholder goals. We want to validate our requirements against the goals and use cases previously defined.
7. Build high-level features into feature model. The feature model is where we begin to capture the product family
8. Decompose features as needed in feature model. Iteration possibilities between UCD and feature model here. At this point we can begin to identify variant requirements and features. We anticipate that this step will likely lead to increased insight into the product leading to potential decision changes that will create opportunity for iteration between the UCD and feature modelling.
9. Use features to encapsulate requirements.
10. Model requirements from user stories in requirement canvas.
11. Use behavioural specification for requirements within modelling environment.

12. Generate bi-directional traceability between features and requirements.

In the world of PLE, a critical process is the identification of relevant system features. In the original work introducing the concept of Feature-Oriented Domain Analysis (FODA) Kyo Huang et al. defined a feature as “A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems” [2]. While this definition has evolved beyond its original scope, the spirit of what a feature is has remained largely the same; an identifiable aspect or artifact of a system that is interacted with. For software, this can be as small as a variable assignment or entire modules and components. In a hardware system, a feature could be the different materials and components used to build a base model versus and top model trim, from the smallest screws to completely different chassis materials.

Finally, in FDD the concept of a feature is also heavily used as a means of organization and task setting [1]. Features in this domain can often be client-focused tasks or actions that the software can perform. The features are used to scope the various tasks required to complete development of a product.

Thus, the overlap becomes apparent; use cases from MDE, features from PLE, and features from FDD all have some forms of overlap. They all have an emphasis on user or stakeholder interactions. They all attempt to capture domain knowledge of what the interactions are. Therefore, for this methodology we use these overlaps to develop strategies for feature/use case identification, requirement elicitation, and traceability. We also establish an pseudo-equivalency between features and use cases between PLE and MDE respectively.

To demonstrate the process, we focus our attention on the automotive

domain as it is both safety-critical and complex with many product variants. It is a domain that combines both software and hardware. There are interactions both at the stakeholder level and internal to the vehicles. The combination of these factors means there is plenty of opportunity for requirement variants to emerge. This presents many challenges with project planning for vehicle development, designing hardware and software for a variety of vehicles, and maintaining documentation throughout the product lifecycle. The objective of this methodology is to capture variability earlier in development in a way that allows for actionable planning. By capturing variability in the requirements, engineers can more easily anticipate and plan around how to build modular systems from the start of a project.

Some examples of variability in the automotive domain can be how to adapt an engine to work in various drive train configurations (4x4, AWD, FWD, RWD) or an HVAC system working in a 1-row, 2-row, or 3-row vehicle. Further, it is a domain that already implements various versions of MDE, PLE, and FDD. The examples shown are not complete and are focused on explaining the process involved for development.

Another contribution of this methodology is in the level of abstraction for development. FeatureIDE [6] is mainly used for reusing software implementations. GEAR [7] is spread out across adaptable ways for a company to be introduced to PLE for design variants. However for industries that continue to grow in complexity capturing variants even earlier in the development phase, such as requirements, becomes increasingly necessary to handle complexity. We aim to introduce PLE techniques earlier into the development process to capture complexity and variations at the requirement level to reduce uncertainty later in the development process.

In summary, once the stakeholders are identified, we want to capture what their goals are when interacting with our system. Those goals are used to guide and justify use cases. They are also used to reason about and refine requirements. By equating features and use cases, we take the use cases to begin feature modelling efforts and feature decomposition. Finally, we specify the requirements within each of the features. Naturally there is opportunity for incremental development within each stage and iterative development between them. We anticipate that most incremental and iterative development will occur between feature decomposition and use cases, between requirement specification and user stories, and feature decomposition and requirement specification. This is due to previous experiences working with industry partners to explore how to handle product variance and decomposition. We observed significant iteration at this level as project planning changes occurred due to changing environmental factors (changes in regulation, economic incentives etc.) and due to engineers learning more about the product they are developing and refining their requirements and designs over time.

3.1 Domain Analysis

Before any engineering work can take place, we must answer the question of who we are building this system for. Without knowing who we are building for it is impossible to properly identify features of the system as we will have no idea who will be interfacing with our system. Further, without knowing who we are building for we have no idea what goals the system will satisfy, and therefore what requirements we want to implement. This is evermore important as we consider the safety, security, and lifecycle implications of who

will be using our system and who will be affected by our system. In the case of the automotive domain, at least two of our stakeholders would be the driver and a pedestrian. Driver as a category however is still quite broad; drivers come in all sorts of different shapes and sizes. Would a young 20 year old male interact with a vehicle the same way a 40 year old female would? What about an 80 year old, healthy male compared to a 30-year-old, overweight male? Or perhaps a 25-year-old female with dwarfism compared to a 25-year-old female with only 1 hand. In all these examples would they all interact with the vehicle the same way?

When we consider how they may all use a vehicle, their use cases, these will eventually be refined into features. The features identified should allow for the widest range of stakeholders to interface with the vehicle. A unique part of the automotive domain is that all stakeholders identified as a driver are equally pedestrians. Therefore we must consider not only how they will interact with the vehicle, but also how the vehicle will interact with them. Would a blind spot sensor identify only vehicles or also pedestrians. How big does a pedestrian need to be for the front object detection system to recognize it as a person?

As such there are some clarifying assumptions that we must make as part of our stakeholder identification. These assumptions are made to help with identification and justification of the goals we elicit. Not every domain may require assumptions. This will vary depending on domain and scope of the project. For automotive we can make some of the following simplifying assumptions (as these are assumptions we anticipate the possibility they may change as development continues or new information is gathered):

- We assume that drivers are at or above the legal driving age in Canada (16 years old).
- We assume that drivers are at or below 80 years old.
- We assume that drivers are able bodied enough to legally operate a motor vehicle.
- Assume height between 151.895cm and 183.24cm. [40] Average range determined between 5th and 95th percentile of male and female population in Canada.
- Assume weight between 48.82kg and 106.60kg. [41] Average range determined between 5th and 95th percentile of male and female population in Canada.

3.1.1 Goals

Once we have identified our stakeholders, we then consider their goals. The goals we identify are highly coupled with how we categorize our stakeholders.

The goal of a pedestrian is different than that of a driver. For example, a customer who purchases a vehicle is a stakeholder that can be both a driver or a pedestrian (just because they own a car now does not mean they will never walk!). The stakeholder goals will likely be very different based on their current role. However the goals between various stakeholders within a category, such as a 20-year-old male or a one-armed 25-year-old female may be quite similar. As such, identifying the goals of the categories should facilitate eliciting requirements of the stakeholders in each role.

This is where we propose the use of goal diagrams to capture this knowledge and information. Goal diagrams present a unique method of capturing this knowledge as it can be both formal or informal to suit the engineers needs. This flexibility supports the notion of formal picnics explained in Meyer’s requirement engineering book [12]. The engineer can start informal if needed and can later formalize the model if required to support further analysis.

According to van Lamsweerde, “a goal is a prescriptive statement of intent that the system should satisfy through the cooperation of its agents”, where an agent can be a human, a device such as sensors or actuators, existing software, or new software [4]. Based on our interpretations, we equate these definitions of an agent to our stakeholders of the system so we will carry on referring to them as stakeholders. Thus for a high-level of abstraction at the vehicle-level, we consider the driver, the pedestrian, and the vehicle as agents of our system. As we reduce the scope of our system to smaller portions of a vehicle, to automatic braking, cruise control, or lane-keeping assist, we may consider other sub-systems in the vehicle as our stakeholder and consider what goals those agents may have for the new system boundary.

We propose a syntax which loosely follows the syntax from i* Strategic Rational Diagram [42, 43]. The legend is shown in figure 3.2. As we are initially using an informal approach we are not as concerned with how the model elements work together or patterns. What we want to convey at this point is what the goals of a stakeholder are, what they might be informed by, and what they might do or use to satisfy those goals. An example goal diagram at the vehicle level can be found in figure 3.3. This goal diagram captures, at a high-level, what the goals of a driver might be when using their vehicle.

Generally, we propose that a driver will use their vehicle to go from one

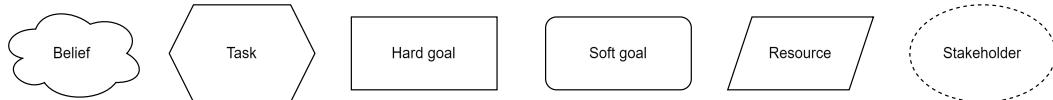


Figure 3.2: Legend of goal diagram elements.

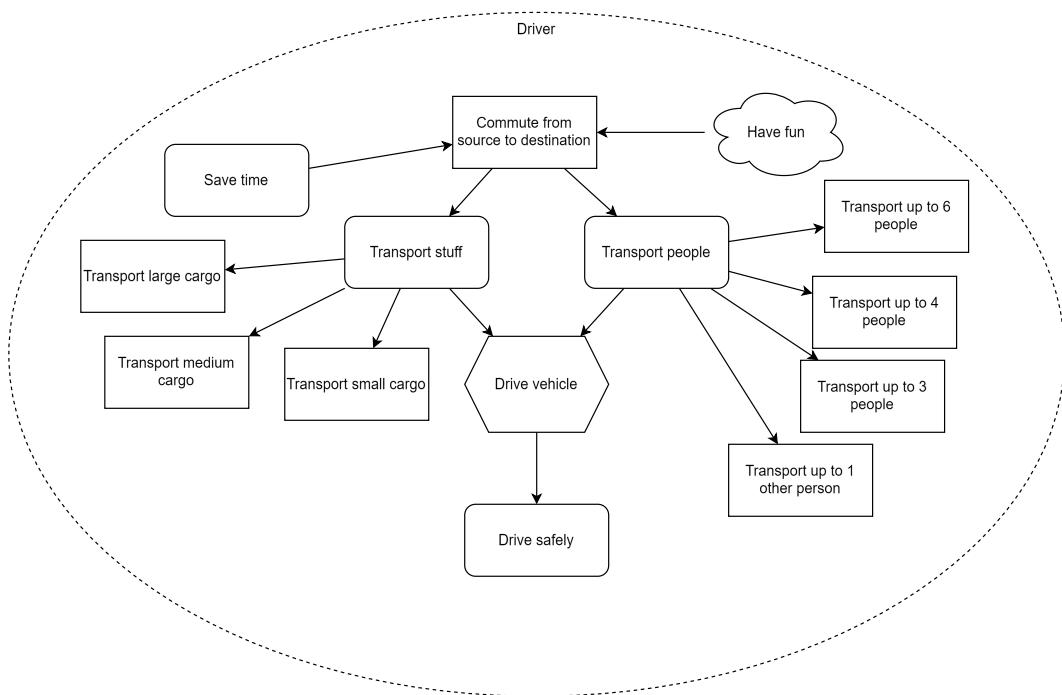


Figure 3.3: Example goal diagram outlining the goals of a driver using a vehicle.

place to another. They are also likely to either carry cargo, people, or both during the commute. They may have some variation in terms of how much cargo or how many people as well. They will also have to drive the vehicle as a task since that is still the main way that people use their vehicle. As part of that task, they will typically want to drive safely to make sure that they get to their destination without issue. While still relatively simple, and without diving very deeply into formally defining the relationship between the model elements we have been able to convey what the goals of a driver could be, introduce some possible variations in goals, and highlight possible goal decompositions. As this is still informal a different engineer may come up with a different goal model for what a driver might do, but they can still be relatively easily merged manually and convey the story of what goals a driver might have as a stakeholder for a vehicle.

As we we will be introducing more formalism later on with the feature modelling and requirement modelling, there is little benefit to introducing that complexity in this stage of development in comparison to the ease of use that we can have with an informal approach to goal modelling. We found a less rigid method of modelling to be more beneficial at this stage to promote wider adoption and understanding for both technical and non-technical individuals. More formalism can always be added by engineers or organizations as needed or desired.

3.1.2 Use Cases

The purpose of the goal diagram is to provide the context of why our identified stakeholders would want to use our system, it does give us answer to what the

connections are between our stakeholders and the system. In other words, along with the goals of the stakeholders and our system, we need to identify how the stakeholders will use our system to satisfy their goals. This allows us to capture what the stakeholders will use the system for, supported by the goals of both parties. We may find during this stage that we missed some goals to provide context for some use cases identified. This is also the first opportunity for iteration in the methodology. As we explore the problem space more thoroughly we hope to fill in these gaps as much as possible before we get to the feature modelling and requirement modelling.

In figure 3.4, we show the parts of the cabin that a driver is likely to use. It is easier to justify some of the use cases compared to others based on the goal diagram we have already created. For example, the goal of ‘have fun’ is hard to trace to any single use case and can be ambiguous with traceability and justification. Drive safely however can be traced to several use cases, such as brakes, gas pedal, and steering wheel. We can see our goals of ‘Transport stuff’ and ‘Transport people’ are also untraceable to the current UCD as we have not specified any use cases around cargo space or passengers. This highlights the first possibility for misalignment between these two modelling efforts; system scoping. The goals of the driver are focused primarily around why they would use a vehicle. When we start to identify use cases there are other questions we want to begin answering. What purpose does the product serve? What problems does it solve for the stakeholder? What do they need to do with our product to solve their problem? The UCD has been written with an implied assumption; a driver only interacts with the driver cabin. During its inception it did not consider what other parts of the vehicle might interact with outside of the driver cabin. These kinds of assumptions are typically easy to make and

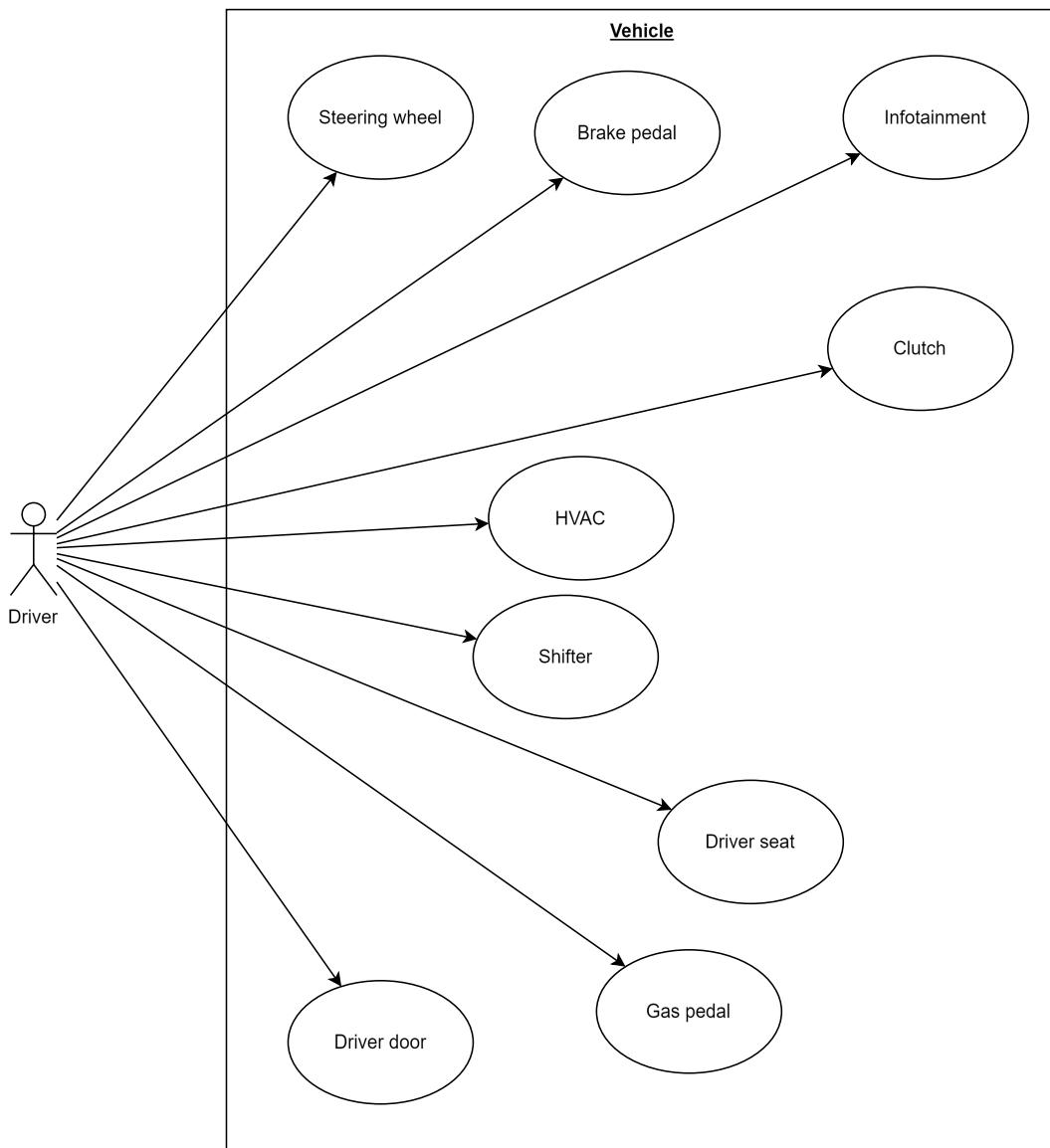


Figure 3.4: A use case diagram outlining what parts of the vehicle a driver will use.

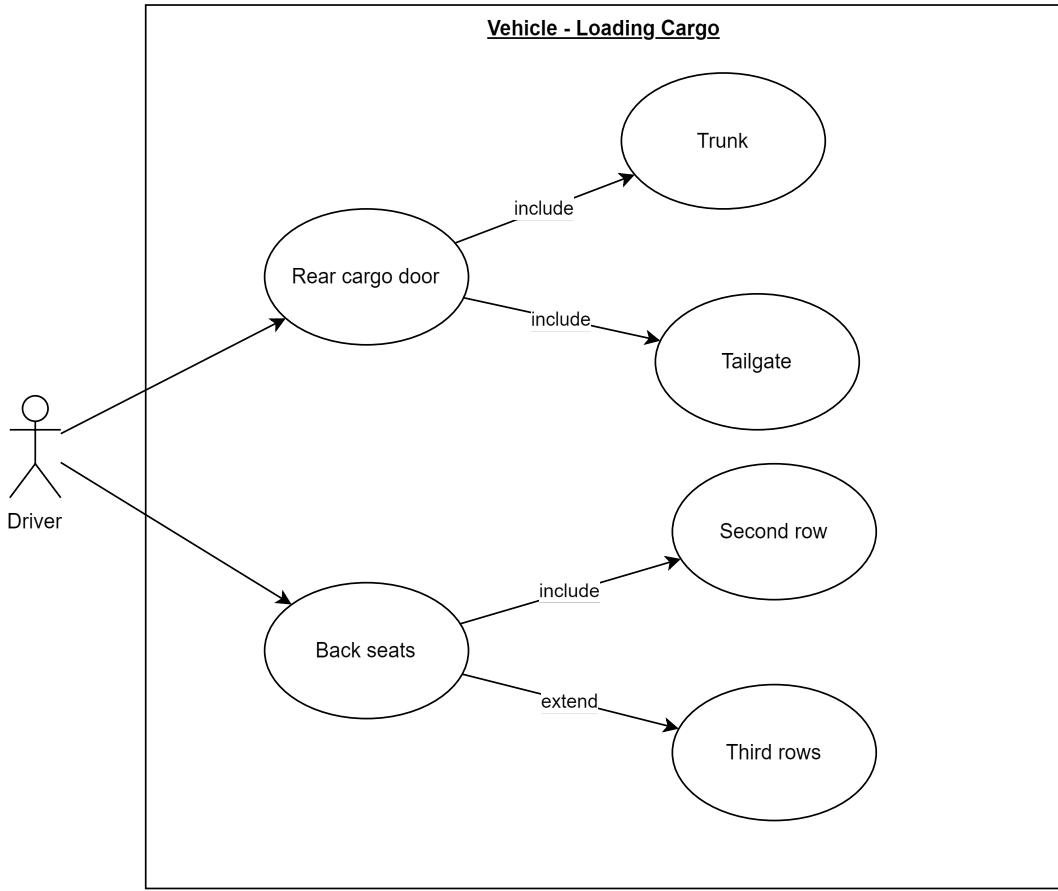


Figure 3.5: UCD scoped by the loading cargo goals

hard to detect.

As a helpful convention to handle complexity, we consider scoping the system boundary based on the goals from the goal diagram. Thus we can re-title the system boundary from figure 3.4 to ‘Vehicle - Drive Vehicle’. This allows a couple of benefits. We can treat each of these UCD boundaries as sub-system and having them directly tied to one or many goals from the goal diagram helps with justifying the existence of the newly identified sub-system.

This simple convention also helps with limiting state space explosion with a UCD of complex system, such as a vehicle, and also helps with scoping the UCD to facilitate system decomposition. We can create another UCD to

specifically target another goal; ‘Transport stuff’. As this effort is not currently tool supported, an external way of maintaining traceability is required. Our initial efforts for this convention have been to maintain a syntactic match or mix between the boundary title and the goal(s) that support its justifications.

For the UCD in figure 3.5, we label the boundary ‘Loading Cargo’ as this is the specific portion of the goal that we are focused on; the original goal of ‘Transport stuff’ also has the implication of driving built into it. With this we can separate the UCD by goals to identify what use cases will work together to satisfy a goal. This is critical as with this proposed methodology, the identified use cases will be treated as the features of our system. The requirements for the features can therefore also be traced directly to the goals that are used to scope them.

3.2 Features

After identifying stakeholders, goals, and use cases, we can begin to identify features of the system. Completeness and correctness is not a main focus during initial efforts as we anticipate iteration and incremental changes throughout development. As such we expect that we may learn something new at any step in the process that informs a conscious decision to return to a prior step to improve a previous engineering artifact.

For this methodology, we want to establish an transformation mapping between use cases and features. Therefore the UCD serves two purposes: it identifies what parts of our system a stakeholder will use to satisfy a goal and what features are likely to exist in our system. However, in feature modelling we can have more granularity in how we decompose features compared to in

a UCD. Usually, this mapping holds true primarily at the higher levels of the feature model. As the feature model is created, further feature decomposition is expected and encouraged further than initial efforts in the UCD. As the UCD is not considered a static document, iterations between the feature model and the UCD are expected. This is especially helpful as there is more granularity in the relationship between features in a feature model compared to use cases in a UCD. Generally, the `include` and `extend` relationships in UCDs are meant to express mandatory and optional use cases respectively according to the SysML specification [11]. Those are loose definitions however, as the semantics of those relationships are often redefined and toyed with by engineers. Using figure 3.5 as an example, regardless of how the drive uses the rear cargo door there must be either a trunk or tailgate; it is mandatory that at least one be present. Conversely, for the backseat use case, given the context of loading cargo into the backseat there must at least be a second row in the vehicle. However, there can be the option of the third row in the vehicle that is not mandatory for the use case to be satisfied, but can still be used based on an arbitrary trigger from the actor.

Meanwhile, the relationships in feature models are well-defined and understood. Given the nature of the UCD specification there is a lot of information embedded into those relationships that needs to be refined in the transformation from use case to feature. The `include` relationship can be used to define mandatory features of a system, however, it does not support enough semantics to differentiate between mandatory features that are mutually exclusive. Similarly the `extend` relationship is not equipped to handle cardinality in optional features where there can be a set of optional feature but a non-empty subset of the optional feature must be selected to complete the product. Thus,

while we can equate features and use cases, the relationships between use cases and the relationships between features are less straightforward to equate. We define the relationship between use cases and features as follows:

$$\text{Let } \mathcal{U} \text{ be the set of all use cases.} \quad (3.1)$$

$$\text{Let } \mathcal{F} \text{ be the set of all features.} \quad (3.2)$$

$$\text{We define } \mathcal{U} \subseteq \mathcal{F} \quad (3.3)$$

We use the definition above to capture the transformation process from use case to feature. This definition also allows for further feature identification beyond the original scope of the set of use cases used for initialization. This allows for some flexibility as not every new feature identified may need to have an origin use case. This definition captures the injective relationship between use cases and features without forcing it to be bijective.

For a simple system, it may be that no further decomposition of features are required after equating use cases to features. However, for the majority of system development we find that feature decomposition is necessary to create a sufficiently complete feature model and capture all components and configuration possibilities of a system. What the mapping between UCDs and features enables is the identification of the initial features of a system. As more feature decomposition occurs, while we encourage iterations between feature models and UCD, it is not currently enforced. Hence, we can end up with more features in our feature model compared to use cases in our UCD. We can see in figure 3.6 what a possible initial feature model can look like based on the previous UCDs.

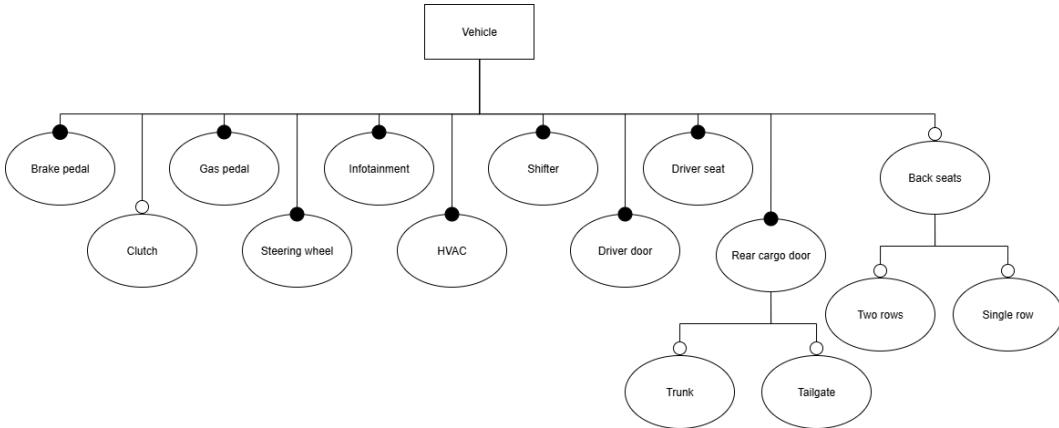


Figure 3.6: Initial attempt at a feature model based on use cases identified in UCD.

As an informal approach to drawing a feature model we use the black circle notation for mandatory features and the white circle notation for optional features. We have defined the clutch as optional, along with both the rear door features and the back seat features. This is where traditional feature modelling takes over. These are optional features of our vehicle but there hasn't been any composition relationship defined yet between them, or the other optional features of our system. We can now define conditions around how the various features should be mixed together to create our system, also known as product, as we transition to the world of PLE. The various versions of the vehicle that we can create based on figure 3.6 are known as product variants. These variants are determined based on what optional features we have in a given product.

At a glance, one product variant might be a vehicle with no back seats, a trunk, and a clutch (a typical coupe style vehicle). Another might be a vehicle with one row for a back seat, no clutch, and a tailgate (A truck for example). We can see what these variants might look like in figure 3.7. It shows just the optional features that become mandatory for the new variants. As previously

mentioned, there are rules that we can define in order to make sure the product variants are semantically and syntactically valid. For example we could have:

$$\text{vehicle.backseats} == \text{false} \rightarrow \text{vehicle.clutch} == \text{true} \quad (3.4)$$

$$\text{vehicle.rearcargodoor.tailgate} == \text{true} \rightarrow \text{vehicle.clutch} == \text{false} \quad (3.5)$$

Where these conditions state that anytime we have no back seats for a vehicle we must include a clutch pedal in the vehicle or anytime there is a tailgate selected there will be no clutch pedal. This translates to all coupe product variants being manual and all truck product variants being automatic. Naturally, there are requirements for each of these features that we need to capture to support development of these product variants. Since the features are mappings from the use cases, and the use cases are supported by the goals we have outlined, the features are also supported by the goals we have outlined. These next need to be refined to more usable requirement specifications.

3.3 Goal Refinement and User Stories

The goals from the goal diagram are used to justify the use cases and features because they can be refined to create our requirements. This refinement process takes shape by finding an answer to how the goals are meant to be satisfied and what they are meant to do. Using the goal of ‘Transport stuff’, we rewrite it as a high-level requirement; vehicle shall enable the transportation of items for stakeholder. We ask the next question; how will the vehicle enable the transportation of items? If there is a goal decomposition, we can also correlate

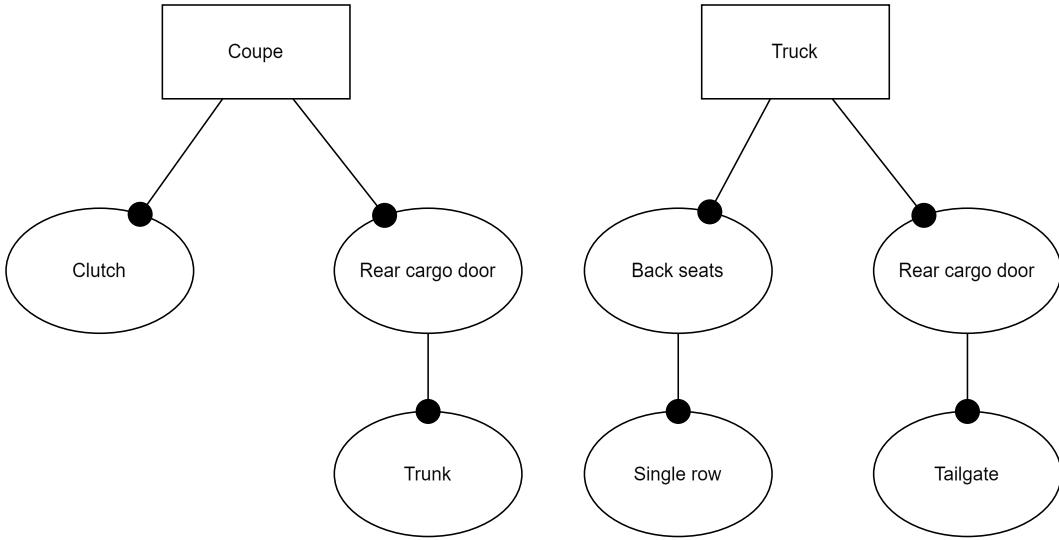


Figure 3.7: Product variants for the initial vehicle feature model. Model is simplified to just the optional features that are mandatory for the variant.

the answer to the question of how to a decomposed goals. How will the vehicle enable the transportation of items; the vehicle shall have a large/medium, small cargo space. What is a large/medium/small cargo space? This question is generally up to the engineers and can be decided arbitrarily or justified with research data.

For writing these user stories we have adapted the use case template from the Meyer requirement handbook [12]. The goal outlines what the user wants to happen. The use case indicates what the user can interact with to achieve a goal. The story connects the dots explaining how the user can use the system to achieve their goal.

While these help to provide requirements from the vehicle perspective, they do not help to determine the requirements from the perspective of the stakeholder. For this, we want to decompose our high-level requirements, identified from our goals, into user stories following Meyer's style of requirements. We thus create the following user story around the high-level requirement of the

vehicle shall enable the transportation of stuff for the stakeholder:

- **Use Case:** Rear cargo door
- **Actor:** Driver, passenger
- **Trigger:** Stakeholder wants to transport stuff
- **Success Scenario:**
 1. Stakeholder identifies access latch to rear cargo area.
 2. Stakeholder uses access latch to open rear cargo door.
 3. Stakeholder is able to lift stuff into rear cargo area.
 4. Stakeholder fits all stuff into rear cargo area.
 5. Stakeholder is able to close rear cargo door.
- **Secondary Scenarios:**
 1. Stakeholder unable to open rear cargo door.
 2. Stakeholder unable to lift stuff into rear cargo area.
 3. Stakeholder unable to fit stuff into rear cargo area.
 4. Stakeholder unable to close rear cargo door.
 5. Stakeholder gets limb or end appendages stuck in door when closing.

Door does not close and stakeholder has suffered harm.
- **Success postcondition:** Stakeholder is safely able to load stuff into rear cargo area.

From this user story we can identify many more requirements around the feature of the rear cargo door and we can identify the goal that this story is

connected to. The points from the user success scenario become the functional requirements for the feature. The secondary scenario outlines other requirements for the feature, both functional and non-functional. Finally the success post condition highlights the satisfaction of the goal if all requirements are met. In summary, from this user story we can identify the following requirements for the rear cargo door and its sub-features:

- Stakeholder shall be able to identify rear cargo door access mechanism.
- Stakeholder shall be able to open rear cargo door.
- Stakeholder shall be able to shall be able to load rear cargo space with stuff.
- Stakeholder shall be able to close rear cargo door.
- Door system shall detect object obstructing door closing path.
- Door shall remain open when object is detected obstructing door path.

3.4 Requirement Modelling and Feature Modelling

With goals, use cases, high-level requirements, and some refined requirements we can begin modelling our requirements. We refer to this environment as our Requirement Canvas. This environment is heavily inspired by the requirement modelling outlined in the SysML specification [11]. We adopted the modelling format and relationsihp between requirements for the requirement canvas. The requirement canvas diverges from its SysML counterpart in its

focus on traceability and implementation details. We wanted to allow for a different specification of requirements within the requirement canvas element. We also have different attributes owned by our requirement types in the requirement canvas. The requirement canvas has several main objectives:

- Provide an environment for modelling requirements in a way that emphasizes traceability between features and requirements. This is meant to be a bi-directional relationship to enable traceability in both directions. This is also meant to automatically propagate changes in both directions.
- Provide users with the ability to specify requirement behaviour in greater detail with a specification language. The language of choice we use at this time is the Gherkin specification language [44].
- Support categorizing requirements. Instead of just having a generic requirement, we want to ability to specify early on whether a requirement is functional or non-functional. We further categorize the non-functional requirements into more specialized categories.

With these in mind, a metamodel was created to provide a specification for how to create models in our newly identified requirement canvas. The metamodel for the requirement canvas is shown in figures 3.9 and 3.10. This metamodel has also been used to develop a domain-specific modelling language named CyclicL. With CyclicL, we aimed to show feasibility of the proposed methodology. This includes the overall objectives of supporting iterative and incremental development and partially automated traceability generation and maintenance.

The requirement canvas portion of the metamodel was created after several iterations of identifying what elements are needed to facilitate the three main objectives for the modelling environment. An important distinction that separates this specification from the requirement model specification for SysML is the requirement types that we have defined. For the requirements in the requirement canvas, we identified four requirement categories; **Functional**, **Qualitative**, **Constraint**, and **Safety**. Functional requirements follow their traditional definition; things that a product or system must do. Qualitative requirements are identified as requirements that affect the way a product or system accomplishes its function. These include the look, feel, usability, humanity, and some performance requirements that are not categorized as functional. Constraint requirements are identified as requirements that add limitations of some type to a product or system. These include operational, environmental, maintainability, support, security, cultural, political, and legal requirements. These requirement definitions can be found in James and Suzanne Robertson Volere requirements [21]. Finally, safety requirements are critical as they identify all requirements that have to do with the safety of users or stakeholders involved in the function of a product or system. We highlight safety requirements as a separate requirement category as our target domain for CyclicL is safety-critical development. This also gives us flexibility to change the profile of the safety requirement type to match safety integrity level ratings based on target domain.

We also added three more classes besides the requirements; **DesignElement**, **Review**, and **TestCase**. The test case and review classes are for verification and validation book-keeping respectively. This is also one of the reasons why we include design elements within the requirement canvas. The test cases are

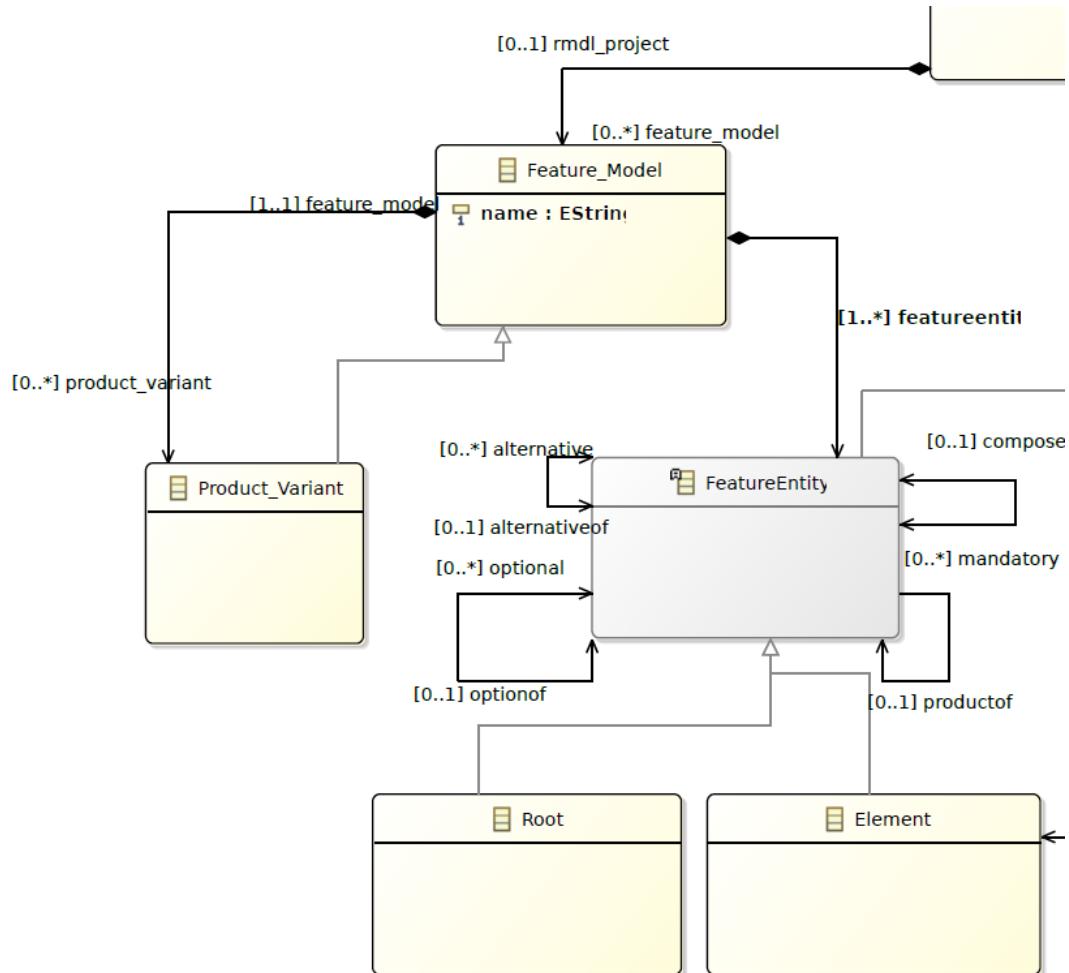


Figure 3.8: CyclicL metamodel. Shows the specification for the feature modelling portion of CyclicL.

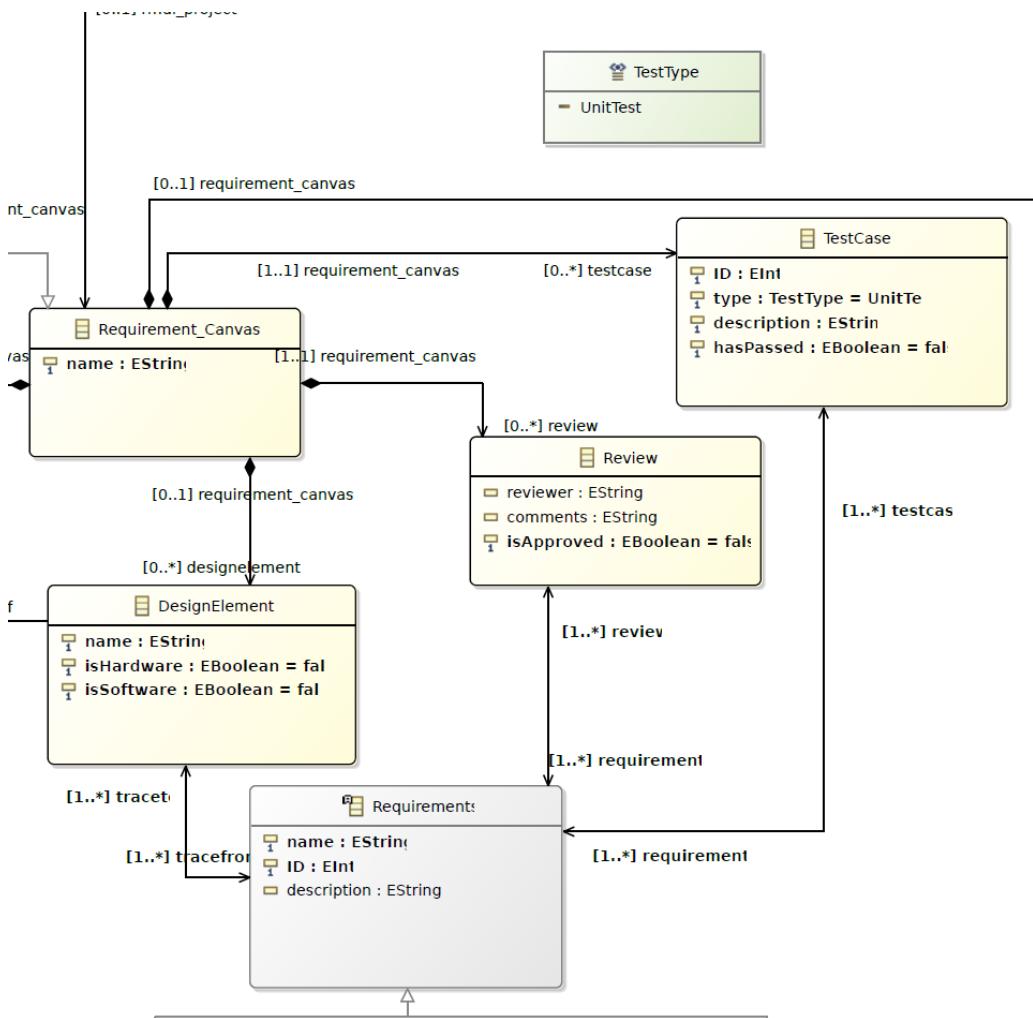


Figure 3.9: CyclicL metamodel. Shows the specification for the first half of the requirement canvas portion of CyclicL.

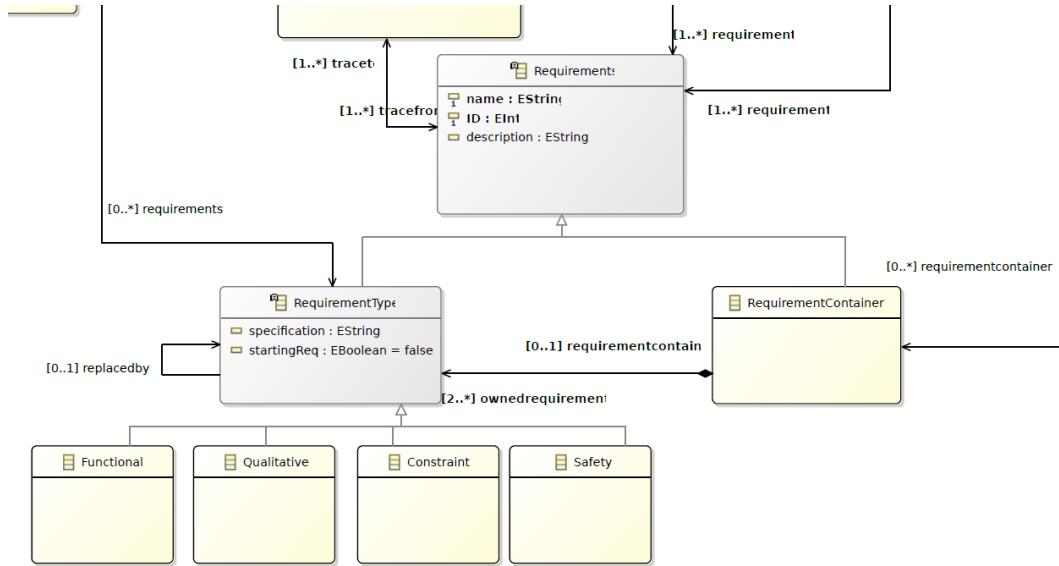


Figure 3.10: CyclicL metamodel. Shows the specification for the second half of the requirement canvas portion of CyclicL.

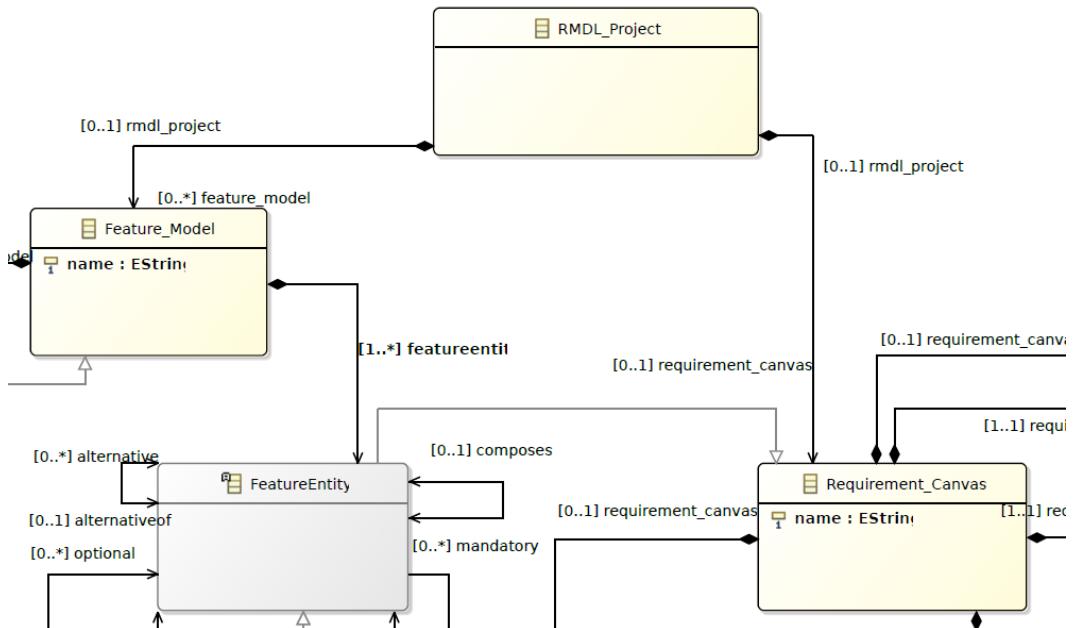


Figure 3.11: CyclicL metamodel. Shows the specification relationship between the requirement canvas and feature modelling portions of the metamodel. Also shows their relation to the model root.

meant to determine if a requirement has been verified against its traced design elements while the review represents if the requirement has been validated. Currently, the verification and validation are expected to happen outside of CyclicL as it was out of scope for initial development. We have an enum `TestType` tied to the test cases. This enum will eventually allow users to categorize the test they may want to apply to their requirements. At this time in development, only `UnitTest` is implemented as a type, but more can be added in future revisions.

The `DesignElement` is perhaps a more controversial addition. There is a risk of polluting the requirements by adding design elements this early in development. The main reason we add them in the metamodel at this stage is once again to enable traceability. Design elements in the requirement canvas are meant to be black boxes to mitigate this potential issue. We have limited the class to determining if a design element is hardware, software, or neither (undetermined when building the requirements or an integrated system). Another reason for having design elements within the requirement canvas is to allow an potential path for future development, specifically design generation based on the requirements specified in CyclicL.

For the feature modelling portion of the metamodel shown in figure 3.8, we tried to remain in-line conceptually with previous work [2, 3, 13]. The `FeatureModel` element is the container for the feature model which is composed of `FeatureEntity` type elements. There is the `Root` element which will form the actual root of the feature model. This is limited to a single `Root` element per model. The root of the model then decomposes to the `Element` type via the three relational associations: `Alternative`, `Mandatory`, and `Optional`. These three relationships are once again bi-directional to enable traceability

in either direction throughout the feature model.

Finally, we have the `Product_Variant` type class defined as an inherited type of `Feature_Model`. This is what allows for users to define their product variants at the highest level and have all the same semantic capabilities as the regular feature model while still allowing us to define more specific semantics around the product variants, such as allowing us to have a zero-to-many multiplicity on how many product variants we can have per feature model.

This specification is loosely based on the work of Peter Höfner et al. [8, 9] which uses set theory as the basis for proving how their feature modelling algebra works. It is also loosely inspired by FeatureIDE [5, 6] a feature modelling plugin tool for Eclipse. The main reason we recreated the specification for feature modelling within CyclicL is to explore the implication of using features to encapsulate requirements, shown in figure 3.11. This is a key step in the process as this allows for what we dub feature-requirement traceability. As this proposed methodology suggests that features be used to encapsulate requirement to support feature-driven development efforts, having a way of representing that relationship in a formal specification such as a metamodel was impactful both for the development of CyclicL and the development of the concept.

The complete metamodel can be found in appendix A.2.

3.5 Feature-Requirement Traceability

Feature-requirement traceability is a main contribution of this thesis. As previous steps have been taken to identify both the features and requirements of the system, this relationship between features and requirements is integral for the

process. This relationship is captured by the inheritance connection between the `Requirement_Canvas` and the `FeatureEntity` classes in the metamodel. By allowing every model element in the feature model to double as a requirement canvas, it allows each feature to contain all the modelled requirements in the requirement canvas. Thus, the individual features are encapsulating the various types of requirements, their associated approvals, and their associated design elements.

The decision to implement the encapsulation as we did was heavily influenced by the desire to tie together two separate modelling approaches and maintain traceability between both modelling environments. This relationship as we have specified allows for several key benefits. First, it provides a mechanism for direct traceability between the requirements and their respective features. Since every requirement will be ‘owned’ by a feature, we can easily trace the relationship between requirements within a feature, between various features, and within the entire system. Secondly, this modelling approach creates minimizes the amount of connections needed between the two modelling environments to a single inheritance relationship while still supporting all our traceability objectives. Thirdly, this allows us to model product variance at the requirement level. Specification of product variance has the potential to be helpful for reducing unexpected changes and capturing knowledge of anticipated variance early in the product lifecycle development. Further, this can be used to support change impact analysis across product variants and their specifications to facilitate product maintenance.

Some anticipated downsides to this extension is that it might not be compatible with existing feature model composition approaches. Since every feature element now doubles as a requirement canvas this can break conventional

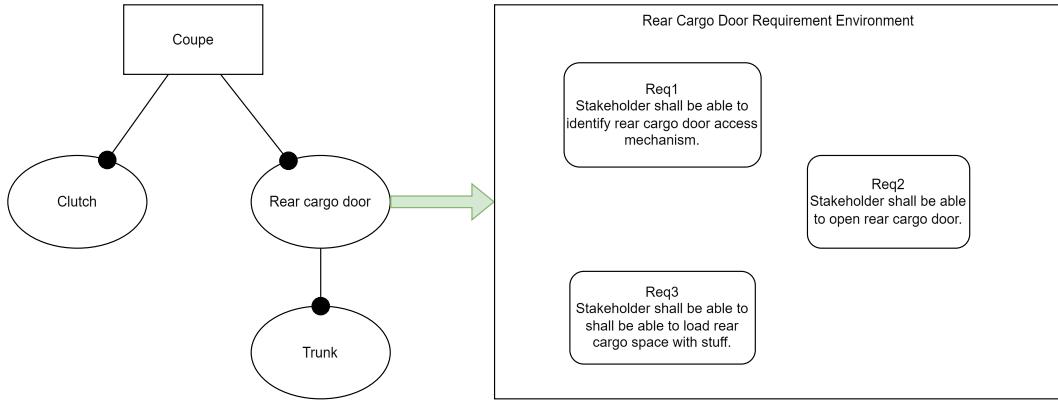


Figure 3.12: Example of how a feature can be opened up to display encapsulated requirements.

feature composition strategies forcing us to create new ones. Another downside is handling requirements that can be related to multiple features. Due to the encapsulation strategy we are proposing, CyclicL does not support cross-cutting requirements across features at this time. There might be ways that users circumvent this which can lead to duplicated or inconsistent requirement across several features.

In figure 3.12 we show the intention of having the features encapsulate the requirements. There is an explicit method for information hiding, and access to the requirements necessitates diving into a feature. This is where we transition from feature modelling to the requirement canvas. Within the canvas we can then define our requirements and specifications for the owning feature.

3.6 Requirement Specification

Between the goal diagrams and use case diagrams we showed how we can carry out requirement refinement via user stories. In the requirement canvas

we intend to express these requirements as a behavioural specification. This is beyond just refinement however, as part of the requirement canvas we also enable requirement specification. Currently we support behavioural specification through an interpreted implementation of Gherkin syntax [44]. The Gherkin style of specification is a structured natural language approach to specifying behaviours. It follows a pseudo predicate logic system using three main keywords; GIVEN, WHEN, and THEN. The GIVEN keyword is used to express behavioural preconditions. The WHEN keyword is used to express events that trigger some new behaviour. These triggers can be either internal or external to the specified system. Finally, the THEN keyword is meant to specify behavioural postconditions.

The reasons for using Gherkin style specifications in the requirement canvas is two-fold. We want to specification environment to be both easy to use and read. This provides a low barrier for entry to specify requirements and reading them. This increased flexibility is meant to both technical and non-technical people to specify system requirements. Secondly, while the flexibility is good, we still wanted some constraints for how specifications can be written. Rather than introducing some arbitrary constraints ourselves, Gherkin provided a nice off the shelf solution. Despite our current implementation not being completely faithful to the original specification, it is enough for a proof of concept.

3.7 Requirement Based Feature Identification

Due to the relationship between features and requirements assumed and defined in this methodology there are alternative ways to identify features. The variation points that exist in this approach are the requirements held in the

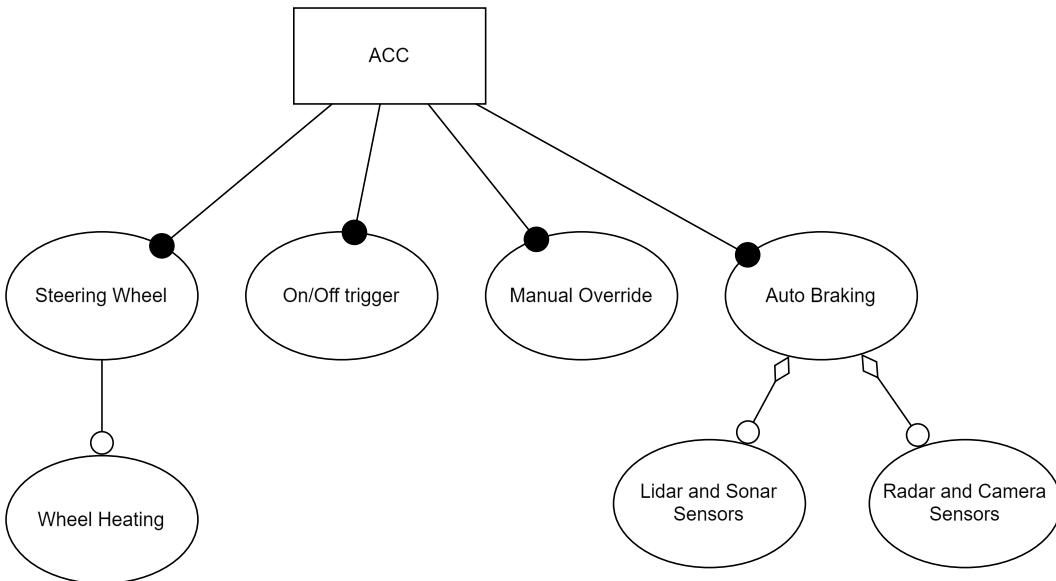


Figure 3.13: Simplified example of an adaptive cruise control feature model.

features. As a result, variant requirements can potentially identify new features previously missed by the UCD. Thus far this methodology has been primarily top-down in the approach to identifying features using the domain analysis. However, it is expected that variant requirements will be identified during the elicitation process. This introduces the possibility for a bottom-up approach to identify features. Variant requirements elicited should still be supported by the goal diagram. Naturally, through this bottom-up portion of the methodology, it is encouraged to iterate on previous steps as engineers incrementally follow the approach.

In figure 3.13, we can see a simplified feature model of an Adaptive Cruise Control (ACC) system. For the auto braking feature there are two alternative features available underneath denoted with the aggregation symbol at the source and open circle at the target. Sensor fusion requirements based on possible variant products (such as radar and camera versus lidar and sonar implementations) generate alternative features under the auto braking feature.

Any requirements that can apply to both are expected to be abstracted up a feature to auto braking. This scoping is meant to reduce cross cutting concerns between features due to requirement dependencies.

In summary, the proposed methodology provides a process to combine MDE and PLE for the purpose of improving requirement elicitation, requirement specification, feature identification, and product planning. It also provided a metamodel to specify a language that can show the feasibility of the process and potential for supporting automated traceability documentation generation and maintenance throughout a product's lifecycle.

Next we will show a tool implementation of CyclicL for the purpose of showing feasibility of the language and potential tool support for the methodology.

Chapter 4

Implementation

In the previous chapter we explored a methodology that proposes how to leverage common points between Product Line Engineering (PLE) and Model-Driven Engineering (MDE) for domain analysis, requirement engineering, and feature modelling. We have also introduced a language, CyclicL, to demonstrate how requirement modelling can be combined with feature modelling to improve traceability via a defined hierarchy between features and requirements.

In this chapter we explore an implementation of CyclicL that shows the capability of tool support for the language. Through this implementation, we attempt to show capability of automatic traceability generation and maintenance from good modelling practice and feasibility of iterative and incremental development using the proposed methodology with tool support.

As a proof of concept, we have implemented a DSL using Sirius [45] and Xtext [46] to show the feasibility of automated traceability between features and requirements. This is a core component of the methodology as we require the means to specify feature models and requirements to capture their inter-relationships to enable automated traceability. By implementing a DSL, we

can enforce syntactic and semantic rules defined from the metamodel to create verifiable models to support incremental and iterative development.

CyclicL is also where we have been exploring potential composition strategies. We anticipate that as a result of designing the language of CyclicL to be focused on feature-requirement traceability there may be some side effects. By implementing the language in a tool, we can explore some of these side effects such as the impact it may have on composition techniques from PLE. Some modifications may be required as a result of the relationship we have introduced between features and requirements.

4.1 Implementation Objectives

We have a couple of key goals we are aiming to satisfy by implementing a tool for CyclicL.

- Enforce syntax and semantic rules of CyclicL when creating models.
- Explore impacts of traceability generation and maintenance when creating and changing models within the tool.
- Enforce automated traceability as a side effect of model creation.
- Explore the impact of feature-requirement traceability on both feature modelling and requirement modelling.

CyclicL has a few objectives to satisfy as a proof of concept. We expect the main users of the tool to be engineers, primarily but not limited to software engineers. To be successful, CyclicL needs to allow users to create feature models so they can define their product families. CyclicL needs to have a

way to define requirements. Since CyclicL is also an implementation of the methodology it also needs to use the relationship we have defined between features and requirements. Finally, since we want to support iterative and incremental development, CyclicL also needs to have some way to enhance the process of generating and maintaining traceability matrices for requirements and features. Finally, the implementation needs to be robust with respect to change to support future changes such as traceability to uses cases and goals or other automated processes such as change impact analysis and product composition.

Based on our objectives, the tool for CyclicL has two main modelling environments that are distinct and connected. The first is the feature modelling half of the tool. The purpose of this modelling environment is to allow the user to build their product family; what are the features and how are they related. The second modelling environment is the requirement canvas. This portion of the tool enables users to specify their requirements, associate requirements to design elements, and assign reviewers and test cases to requirements. This is where the traceability is captured between the feature and various aspects of the requirements.

As discussed in section 3.5, the features encapsulate the requirements via the inheritance relationship from the requirements canvas class to the feature model class. This is our implementation of the hierarchy between features and requirements. Connecting these two modelling environments affords us a variety of traceability options. Scoping the traceability matrix to a single feature we can get a simple requirement traceability matrix. Scope the traceability to the product family and we can get a tabular representation of the feature model. Or, from the same scope, we can get complete traceability of

all the requirements present in the product family. This includes their owning features so we can have a full list of complete traceability for all the features and requirements. We anticipate that this same mechanism can be used to generate traceability based on product instances as well.

Further, our specification allows for another interesting scope. We can have the 'null feature' scope. This is the requirement canvas that isn't owned by any feature model. This is enabled via the composition relationship from the project root class down to the requirement canvas class. Due to cardinality of 0..1, CyclicL affords the end user the option to create a requirement canvas belonging to no feature.

There are a couple of reasons why this can be helpful. Referring back to the scenario where we already have requirements before we start identifying features, having a featureless requirement canvas allows the user to define their requirements right away. This way they have them present within CyclicL from the start and can simply move them into the appropriate feature at a later date. A simple way to make sure that all their requirements have been assigned to features would be to make sure that this requirement canvas is empty when the features have been identified.

Another possible scenario would be new requirements that are identified through product lifecycle development. They may not already have an owning feature identified. Rather than not specifying them, using the featureless requirement canvas allows the end user to fully specify the requirement and use the canvas similarly to a bank. The user can then go back to the bank to withdraw the requirement when an owning feature has been identified.

The feature modelling specification of the tool is loosely inspired by the formal approach from Peter Höfner *et al* [8, 9], with a small extension to account

for requirement encapsulation. We use this specification instead of making a new one to leverage previous proofs and work done on formal specifications for feature modelling. It saved us time and let us begin development much sooner. We consider the relationship between features and requirements within a product family as shown in figure 4.1. By using this definition, we treat each feature as a set of requirements such that every requirement within the feature is unique. As stated in the definition however, it does not account for duplicate requirements in separate feature elements. This is something that will be addressed in future iterations.

Specifying the hierarchy between features are requirements both in a meta-model and using a set theory approach gives us a better understanding for the mechanisms involved between the two entities. The idea is that this will help with specifying and implementing mechanical processes, for example change impact analysis, in the future using a formalized language like the Object Constraint Language (OCL) [47, 48], or another formal approach. This should help with validation of the mechanism and streamline development as we can use one of its derivatives baked into Eclipse like cro:EVLEclipse Validation Language (EVL) or cro:AQLAcceleo Querying Language (AQL).

The requirement canvas portion of CyclicL that was inspired by requirement diagrams from SysML [11], however we did not end up implementing it according to its original specification. The main reason for the divergence was due to our focus on implementing the feature-requirement hierarchy and developing for feature-requirement traceability specifically. As a result, we decided to iterate on requirement diagrams as a means of exploration for traceability and requirement specification leading to the inception of the requirement canvas. In the process of exploring we made several decisions both at the

Let \mathbb{F} be a set of arbitrary elements that we call features. (4.1)

We can call a collection (set) of features a product. (4.2)

The set of all possible products is $\mathbb{P} \stackrel{\text{def}}{=} \mathcal{P}(\mathbb{F})$ (4.3)

A collection of products (an element of $\mathcal{P}(\mathbb{P})$) (4.4)

is called a product line (or product family) (4.5)

This model does not capture feature duplication (4.6)

Let \mathbb{R} be a set of arbitrary elements that we call requirements. (4.7)

We can call a collection (set) of requirements a feature. (4.8)

The set of all possible features is: $\mathbb{F} \stackrel{\text{def}}{=} \mathcal{P}(\mathbb{R})$ (4.9)

This model does not capture requirement duplication. (4.10)

Figure 4.1: Definition of the relationship between features and requirements.

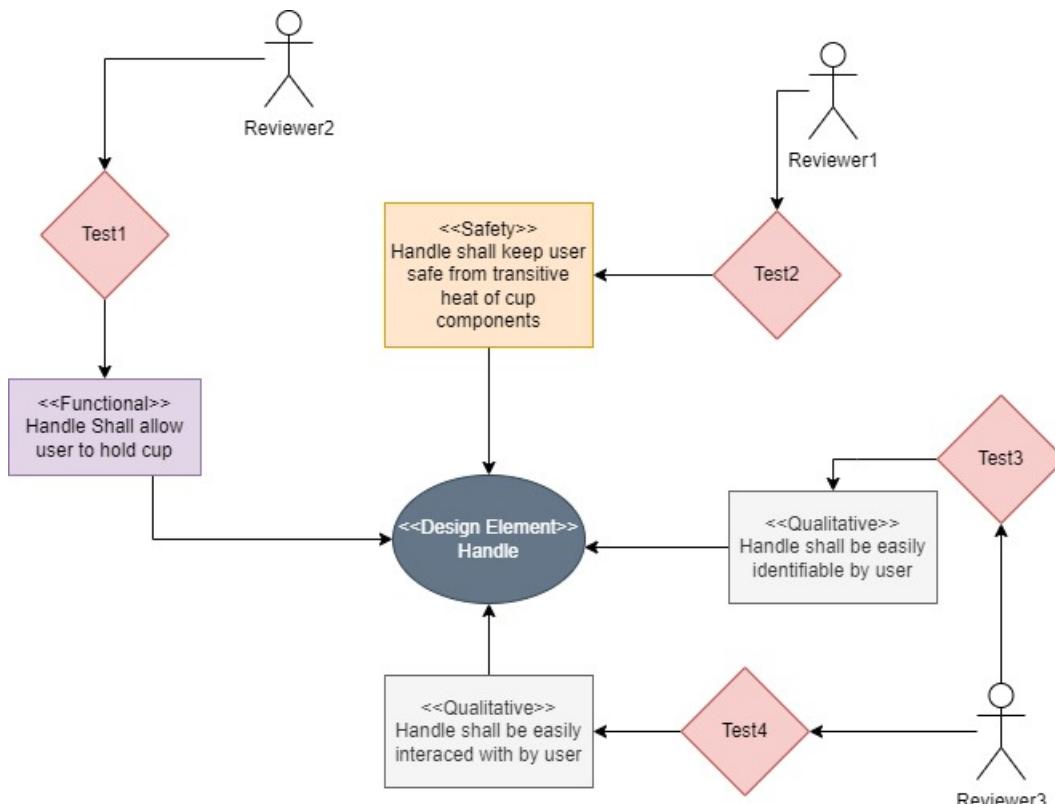


Figure 4.2: Example of a requirement canvas concrete syntax. The example uses a disposable coffee cup with a handle as the focus of the diagram.

requirements and design level for CyclicL that led to its state at the time of writing. This includes the classifications for functional and non-functional requirements, the graphical syntax implemented, and the specification language embedded in the model. We explain these differences in sections 4.2 and 4.3.

Another objective we had during development was reuse. As we were exploring and implementing traceability, and in the spirit of PLE, we were trying to identify as many reusable aspects as possible from previous specifications and implementations of tools as we could. Thus, CyclicL is also inspired by FeatureIDE [5, 6] and GEARS [7] for their abstract and concrete syntaxes. This prevented us from having to reinvent how to build a feature modelling tool so we could focus more efforts on the requirement canvas and the traceability matrices.

4.1.1 Goals and Limitations

In figure 4.3 we capture some anticipated goals for our end-users. In conjunction with our own objectives, we use these to support our high-level requirements for CyclicL. At the time of writing the scope of CyclicL is currently only focused on the traceability portion of the methodology we have scoped user goals to be after they have completed an independent domain analysis. We assume that users will complete the other steps of this methodology (goal diagrams, UCD) using other means prior to or as part of an iterative process when using CyclicL. This is a limitation of the tool for now as we do not currently support modelling or traceability from the domain analysis to the identified features and requirements. We strongly encourage that users complete the prior steps of domain analysis before modelling the features and requirements

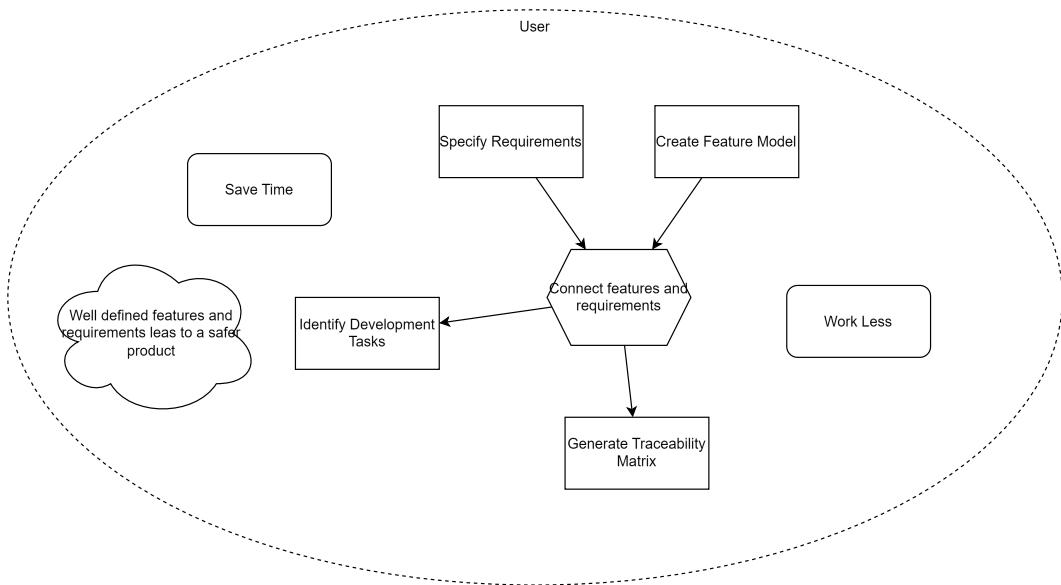


Figure 4.3: Anticipated goals of users for CyclicL.

as it helps the development process to build a better conceptual model and understanding of the intended product.

4.1.2 Requirements

We extracted high-level requirements from previous work in building a product line and requirements for a coffee cup to guide initial development plans for CyclicL [49]. We also have the goal of supporting iterative and incremental development lifecycles. We generally define an iterative change as a change based on feedback from later stages of development or stakeholders. These are usually larger changes that may arise cycling between phases of development. In the context of CyclicL, we want to generate and maintain traceability between features and requirements as users iterate between feature modelling and requirement modelling. Examples of an iterative change can include a new version of a development artifact or a redesign of a development artifact. Incremental changes are changes that are usually smaller in scope, and bound

within a single development phase, such as feature modelling *or* requirement modelling. Examples of an incremental change can be a refactoring change, updating a definition, or changing a connection between components. While we have been using these definitions to guide our development philosophy, this may not be a shared definition by others. This may present a challenge to the adoption or usage of CyclicL. Therefore, we included ease of use and adoption as another goal. Our initial list of requirements was as follows:

- Tool shall provide the user with an environment for defining requirements.
- Tool shall provide the user with an environment for defining features.
- Tool shall allow the user to specify design elements.
- Tool shall allow the user to relate one or more requirements to one or more design elements.
- Tool shall allow the user to use features to capture and own requirements.
- Tool shall automatically create and maintain a traceability matrix.
- Tool shall facilitate iterative development of the requirements canvases (*note:* this remains as ongoing work).
- Tool shall facilitate incremental development of the requirements canvases.
- Tool shall aim to promote familiarity and ease of use to increase rate of adoption.

Along with these requirements, we also use the prior specifications for feature-requirement encapsulation. We treat the requirements encapsulated by a feature as attributes and are considered feature requirements, thus scoped by the feature. As the exploration of requirement modelling started with the requirement diagrams from SysML, we reused as much as possible from the original specification. We then focused more on traceability aspects and integration with feature modelling, diverging from the original specification until we ended up with what we now refer to as our requirement canvas. For a rough idea of what we wanted the requirement canvases to look like we sketched requirements for a disposable coffee cup with a handle, shown in Figure 4.2.

4.2 Abstract Syntax

The metamodel shown in figure A.1 is what we used as a specification for building CyclicL. The abstract syntax can be roughly divided into two main portions; the feature model and the requirement canvas, mirroring our requirement decisions for CyclicL. The requirement canvas portion of the metamodel is not a complete implementation of the SysML specification for requirement diagrams. We decided to implement enough to allow us to model requirements sufficiently to get traceability. To enable features to encapsulate requirements, the `FeatureEntity` class inherits from the `Requirement_Canvas` class. This implementation is what allows the features to encapsulate the requirements. We chose the inheritance relationship as it makes the features share the semantically equivalent to a requirement canvas while also allowing us to define a feature independent of a requirement canvas syntactically. Using another relationship for the implementation, such as a composition relation-

ship, would have added a strict ownership semantic that we did not find added any value and instead added an extra layer of abstraction between features and requirements that ended up cluttering the traceability with useless information. Through this inheritance relationship the features own the requirement directly without any abstraction. Further, the inheritance specification forces a 1-1 relationship between the requirements and the feature meaning every new set of requirements will require a new feature to be encapsulated, reinforcing the convention of a unique feature per unique set of requirements.

There are three defined relationships for features; mandatory, alternative, and optional. These are specified with bidirectional references to allow visibility in either direction from any feature. This decision was made to enhance the traceability available in the modelling environment and simplify development for the traceability matrices. The multiplicities are also defined with 0..1 at the top end to limit how the feature composition will work. This limits how many nodes a leaf element can be connected to, constraining what models can be built and simplifying the feature composition. We also highlight the requirement set owned by the null feature based on the composition relationship from `RMDL_Project` to `Requirement_Canvas`. The justification for this structure is to provide an extra space for requirement specification that can later be refined by cutting and pasting the requirements into a canvas encapsulated by a proper feature. To prevent cluttering instantiated projects with too many disconnected drawing spaces we limit the multiplicity to [0..1].

Next, for the requirements in the requirement canvas, we identified four requirement categories; **Functional**, **Qualitative**, **Constraint**, and **Safety**. Functional requirements follow their traditional definition; things that a product or system must do. Qualitative requirements are identified as requirements

that affect the way a product or system accomplishes its function. These include the look, feel, usability, humanity, and some performance requirements that are not categorized as functional. Constraint requirements are identified as requirements that add limitations of some type to a product or system. These include operational, environmental, maintainability, support, security, cultural, political, and legal requirements. These requirement definitions can be found in James and Suzanne Robertson Volere requirements [21]. We found a benefit for more specific categories beyond just non-functional requirement as it lets the user be more intentional with their specification. We find this especially beneficial for identifying safety requirement separate from the other traditional non-functional requirements. The safety requirement category is critical for our purposes as they identify all requirements that have to do with the safety of users or stakeholders involved in the function of a product or system. We highlight safety requirements as a separate requirement category as we want to target CyclicL towards safety-critical development. It is important to note that requirement categorization relies heavily upon system, or in CyclicL, feature scoping. For example, we hypothesize that in a safety feature, a safety requirement may be considered a functional requirement due to the feature scope. A functional feature, however, may contain safety requirements that are distinctly different from the functional requirements. This distinction relies heavily upon the judgement of the engineer creating the model and the model context.

We also added three more classes besides the requirements; `DesignElement`, `Review`, and `TestCase`. The test case and review classes are for verification and validation book-keeping respectively. We also have an enum `TestType`. This enum allows users to categorize the test users may want to apply to their

requirements. At this time in development, only **UnitTest** is implemented as a type, but more can be added in future revisions. This is also one of the reasons why we include design elements within the requirement canvas. The test cases are meant to determine if a requirement has been verified against its traced design elements while the review represents if the requirement has been validated. Currently, the verification and validation are expected to happen outside of CyclicL as it was out of scope for initial development. Design elements are also meant to be black boxes as we do not want to pollute our requirement environment with design aspects. We have limited it to determining if a design element is hardware, software, or neither (undetermined when building the requirements or an integrated system). Another reason for having design elements within the requirement canvas is to allow an opportunity for future development for design generation based on the requirements specified in CyclicL.

4.3 Concrete Syntax

The design decisions towards the development of the concrete syntax for the requirement canvas follows the ideas from the Physics of Notation [50]. Various colors and shapes were used to maintain a bijective relationship between the concrete syntax and intended semantics. For example, in figure 4.4 all of the requirements have the same shape to show they have commonality as they inherit from the **Requirement** type, but use different colors to differentiate themselves from each other. Similarly, **Test Cases** and **Reviews** are dynamically colored to show when they pass/fail and approved/unapproved respectively. Finally, the **Design Element** symbols dynamically change color

if they are stereotyped as software, hardware, or black-boxed.

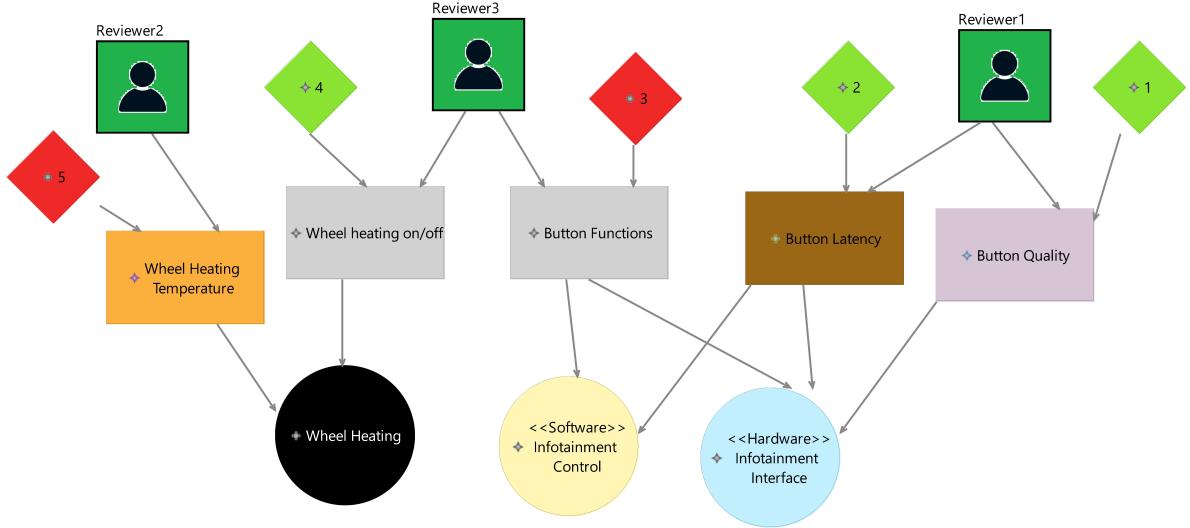


Figure 4.4: An example of the concrete syntax for the requirement canvas using requirements from the automotive domain.

For the feature modelling portion of CyclicL, we attempted to maintain as close to traditional syntax as possible. This was to satisfy the requirement of ease of use and familiarity. Maintaining the original syntax for feature modelling to our best ability during implementation should provide some intuitive familiarity to users familiar with feature modelling while allowing a new user to learn relatively easily using previous work and documentation about PLE. Given some of the default limitations of Sirius there are still some differences.

Figure 4.5 shows an example product family created in CyclicL. The root of the model is shown in the grey box and the features of the model use white circles. The mandatory reference uses an arrow and black diamond at the target to represent the AND relationship. The optional relationship uses circle with a plus at the target to represent OR relationships. The alternative reference uses an arrow and white diamond at the target to represent XOR relationships. As this is a top-down modelling layout, the reference source is towards

the top and the reference target is towards the bottom. We aimed to maintain a syntax that was as similar as possible to feature modelling convention and maintain a bijective relationship between the syntax and semantics.

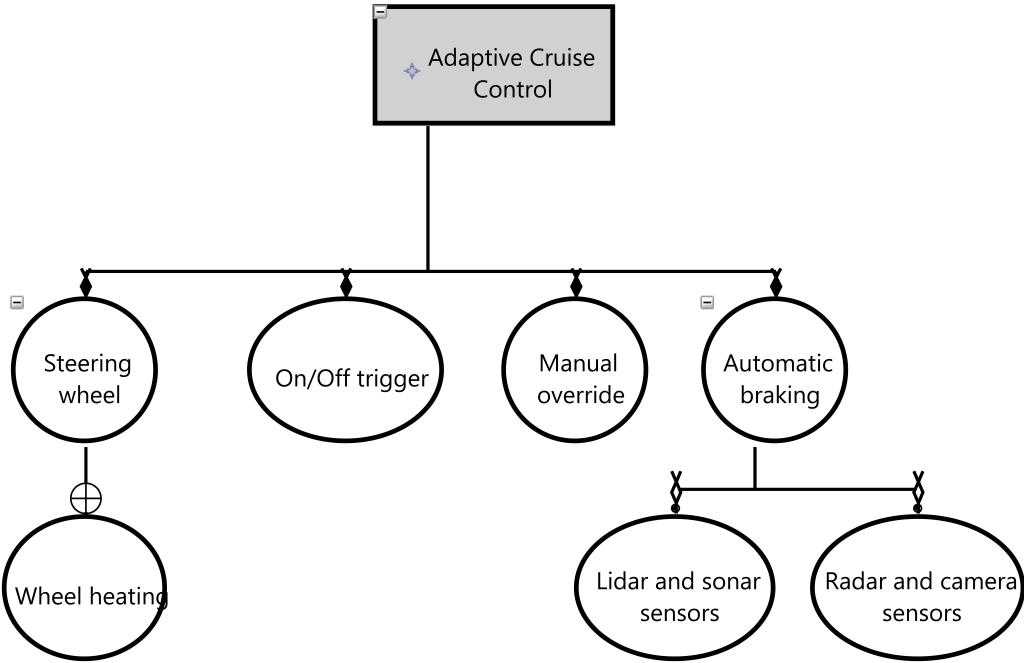


Figure 4.5: An example of the concrete syntax for the feature model using features for an adaptive cruise control system from the automotive domain.

Along with the graphical portion of CyclicL, we also implemented an Xtext-based [46] specification language. The implemented specification language is based on the Gherkin specification language [51, 44]. We chose this as the specification language for our requirements for a few reasons. Gherkin uses a structured natural language for specifications, thus making it relatively approachable for users when defining their requirements. Further, despite using natural language, it uses keywords to provide a predicate logic structure to the specifications. The keyword ‘Given’ is used for preconditions, ‘When’ is used for events, and ‘Then’ is used for postconditions. The combined approach

```

Given{
    Precond FeatureExists: "Steering wheel has heating feature."
    Precond CarIsOn: "Vehicle is on."
}
When{
    Event UserTurnsOnHeating: "User interacts with heating
        interface to turn on/off wheel heating."
}
Then{
    Postcond StateChange: "Wheel heating boolean changes state
        to on or off depending on current state."
}

```

Figure 4.6: Gherkin specification for Wheel Heating on/off functional requirement. The high-level requirement description is: User shall be able to turn wheel heating on/off.

of adding predicate structure to natural language makes it relatively easy to specify requirements and to implement the language within CyclicL. This helps to satisfy the ease of use and adoption requirement as it makes the tool more approachable to learn and implement in a development process. We found that Gherkin struck a nice balance between formal informal approaches to specification that aligns with the philosophy presented in Meyer’s requirement style that inspired some of this methodology.

4.4 Design Decisions

A major design decision for CyclicL was using our own implementation for feature modelling. Other tools such as FeatureIDE already exist and are native to Eclipse. We decided to use our own implementation to remove possible constraints of having to work with an existing tool so that we can focus on the traceability aspects of the tool. Another reason was to allow us some flexibility

in how to implement the new hierarchical relationship between features and requirements. As we were not sure how best to implement the relationship to support. Further, this gave us flexibility to explore how we might want to navigate the created models and traceability matrices.

4.4.1 Composition Implementations

The relationship between features and requirements makes traditional composition techniques for PLE a challenge. Given our convention that a variant requirement necessitates a new feature, the path to composing features and requirements to a product is relatively straightforward at this time. Currently, the approach focuses on the features themselves for the composition, with the assumption that each feature has unique requirements. If a requirement applies to multiple features then it is meant to be abstracted up a level in the tree to a higher feature. This currently reduces cross cutting concerns and dependencies between requirements. This allows users to define the necessary constraints and rules for how the features within a model are allowed to compose to a product (every car needs 4 wheels for example). As of the time of writing more complex analysis is planned for future work, however as a proof of concept, CyclicL has the facilities to support such development in the future. With the `Product_Variant` class in the metamodel [A.1](#) we provide the facilities for a user to manually create instances of their product. Also due to the composition relationship that we use, all the product instances are owned by the feature model they are created in. The inheritance relationship also means that they don't need to be new semantic elements to represent the various features and requirements in the product variants. We can take advantage

of model slicing to create those new variants and instances. Naturally, the inheritance also allows all the traceability mechanisms to apply to the product variants the same way they do to the parent feature model.

Chapter 5

Evaluation

Throughout this thesis we have been using the automotive domain to tell the story of how to use this methodology and CyclicL. This is because within the automotive domain we can find two of the main issues we are addressing with this proposal; how to handle variants within a product family and maintaining traceability throughout the product stacks. For automotive manufacturers there are variants in the customer facing product, evident in the different vehicles that are offered, the different model years between vehicles, and the various trims that are available within a single vehicle model. There are also variants that can exist internal to the manufacturer as they go through development and decompose the vehicle into it's components; engine, transmission, cooling, heating variants that are used across platforms for example. Managing these portfolios, across multiple levels of abstraction necessitates product family traceability in order to keep track of requirements constraints that lead to the various design and implementation decisions.

For the evaluation portion of this thesis, we look towards the medical domain and medical device development. There is a lot of overlap between med-

ical device development and automotive manufacturing. The medical device we will be using for this evaluation is a pacemaker. From the perspective of safety criticality, medical devices such as a pacemaker tend to have a greater impact on individual health and safety which is often reflected in medical regulations. Regional legislation may affect product development and the products themselves may have variants based on the capabilities of target stakeholders, in this case individual patients. For this thesis we keep in mind medical device guidelines and classifications based on Health Canada regulations [52, 53] as we are from a Canadian institution. This means that a pacemaker would be a class 4 medical device.

This evaluation will use an existing requirement document from Boston Scientific A.1 for pacemaker development. The pacemaker requirement document we have chosen is often used as part of the undergraduate curriculum at McMaster University for teaching product life-cycle development and safety-critical development. We will be assessing how this methodology and tool can be used to improve an existing requirement set, create traceability between features and requirement, and potential integration points with a development process.

5.1 Knowledge Capture & Domain Analysis

The first step in this proposal is examining how the domain analysis is conducted. This part of the evaluation examines how we can enhance existing knowledge capture approaches by introducing domain analysis techniques introduced in our methodology. Using the proposed processes, along with CyclicL, we will be assessing for the ability to find gaps in the original pace-

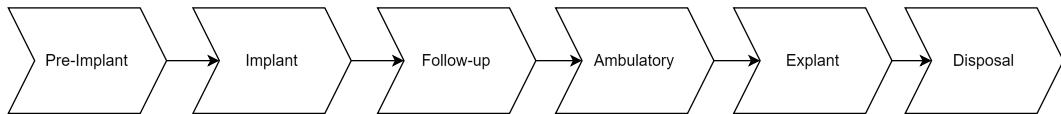


Figure 5.1: Pacemaker lifecycle as outlined by Boston Scientific requirements.

maker requirement document, potential integration points for existing development process, and general enhancements for engineering practices.

5.1.1 Goal Diagrams

The full requirement set by Boston Scientific for a pacemaker is available in the Appendix A.1. Though not explicitly stated as domain analysis, there is some knowledge capture that occurs in chapters 1 and 2 of the pacemaker requirements all captured in natural language. In fact, all the requirements specified by Boston Scientific are captured in natural language. These two chapters introduce the scope of the requirements, identify some of the stakeholders, and define the system and sub-systems. We found it sufficient for the engineers to have an idea of why the system is needed, who they are building it for, where its intended usage environment is, and why it is needed. We now attempt to improve this domain analysis using the goal diagram and UCD approaches outlined in our proposed methodology.

We follow the same pacemaker lifecycle outlined the requirement document shown in figure 5.1. The lifecycle provides context for what the goals of our stakeholders would be throughout the life of the pacemaker. We find that the document explains what is necessary from the pacemaker during each stage of its lifecycle. The document does not however explain what each stage actually is. The stakeholder roles, use cases, or goals are also not defined our outlined for each stage throughout the product lifecycle. This can reduce

the engineers' understanding of the product, and impact confidence in the requirements specified for product development.

The lack of context and knowledge can become a source of uncertainty during development. The extra definitions or knowledge of the stakeholder goals may not always result in a direct impact on the product features or requirements, however the ability to draw this conclusion from a body of knowledge is important to increase confidence in creating more complete and correct sets of features and requirements. To accomplish part of this domain analysis we performed a casual interview with a doctor to provide some perspective and insight to what some stakeholder goals might be outside of the perspective of the engineer responsible for development. Ultimately, this was a benefit as the doctor helped to provide some context missing from the original pacemaker specification.

The first step we took in this domain analysis is to provide some more information around what happens in each stage of the pacemaker lifecycle.

1. **Pre-Implant:** This stage is where the patient is assessed if a pacemaker is required for their benefit. This stage includes risk/benefit discussion, patient risk tolerance, and informed patient consent. The patient has right to refuse care and may deny getting a pacemaker altogether.
2. **Implant:** This stage is the surgery. Straightforward stage, the patient undergoes surgery to implant the pacemaker.
3. **Follow-up:** This stage occurs directly after the pacemaker is implanted. The patient is assessed for surgery safety (correct implantation of the pacemaker, no infection, no other surgical complications etc).

4. **Ambulatory:** This stage is the regular day-to-day life of the patient post-surgery. This include regular check-ups with the patient's medical team. Regular assessment of the pacemaker is performed at this stage. If necessary, pacemaker maintenance or updates may be required. In extreme or unusual cases, pacemaker removal may be required or refused.
5. **Explant:** This stage is where a pacemaker is removed from the patient. Extremely unlikely to be assessed as needed and even less likely to occur during the patients lifetime. However the pacemaker may still be removed after patients death.
6. **Disposal:** This stage is when a pacemaker has reached end of life and is safely disposed.

These definitions are helpful to engineers to get a better understanding of the complete product lifecycle after development. It helps to inform some of the goals that the various stakeholders may have, providing some insight into why the requirements are specified as such. Further, these extra definitions also helped us to identify more stakeholders of the system beyond the original specification. The new list of stakeholders is the following:

- Patient (from Pacemaker requirements)
- Patient Family
- Doctor (from Pacemaker requirements)
- Hospital
- Nurse (from Pacemaker requirements)

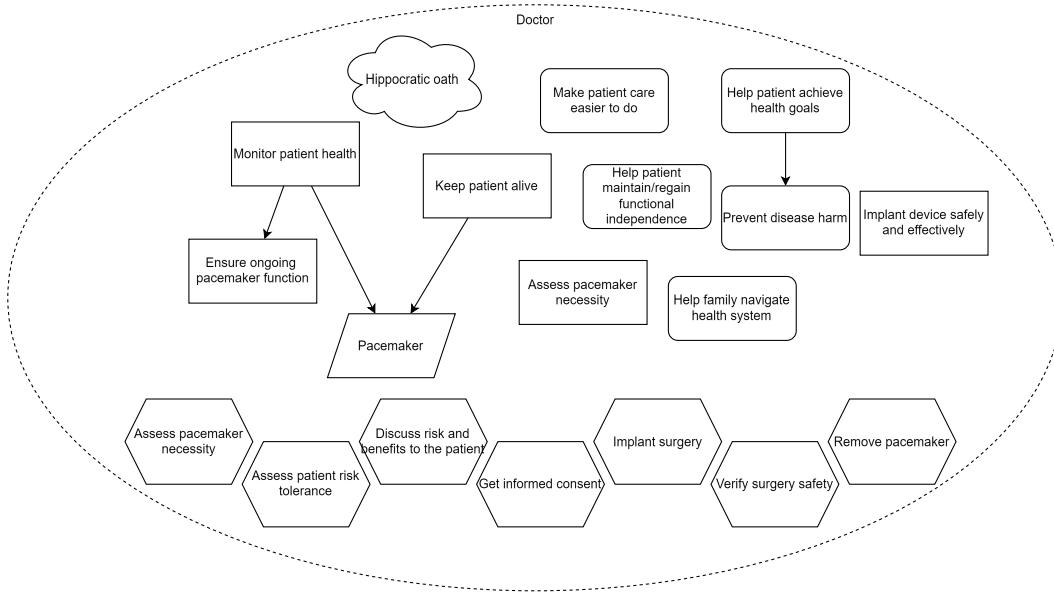


Figure 5.2: Doctor goals in the context of the pacemaker system.

- Technician (from Pacemaker requirements)

The patient family was identified as another stakeholder of the pacemaker system due to some of the goals that were identified for both the patient and doctor stakeholders. One of the goals for doctors is to help the patient family navigate the health care system. Which informed us that the patients family are also stakeholders of the pacemaker system as they are likely to be involved with the patient quality of life. This ties into another goal we identified for the patient which is to regain or maintain a desired quality of life. The goals of the doctor and the patient stakeholders can be found in figures 5.2 and 5.3 respectively.

One important belief that is important to identify is the patient belief **We all die eventually**. This leads to the patient goal of right to refuse care. This is critical to recognize patient autonomy in general. In the specific case around the pacemaker it presents itself twice; a patient can refuse to implant

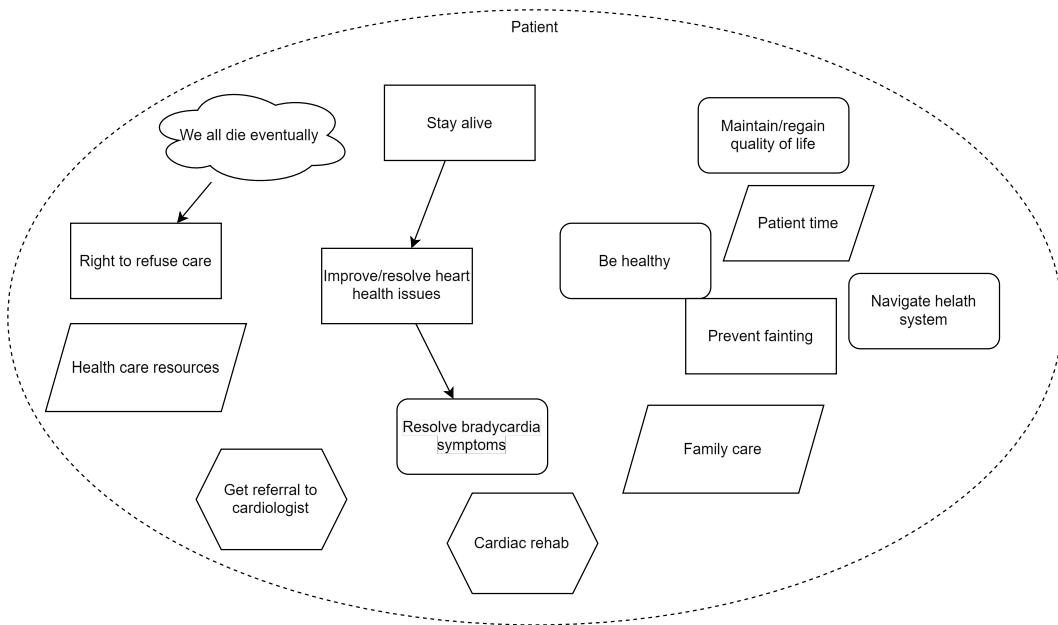


Figure 5.3: Patient goals in the context of the pacemaker system.

the pacemaker and can refuse to explant the pacemaker. Refusal to implant the pacemaker is less common, but informed consent is an important part of the process before a doctor can implant the pacemaker. More commonly, especially with pacemaker failure, the patient can refuse to have the pacemaker explanted. This can be for a multitude of reasons, but often it is simply because the patient is old and does not want surgery. We point this out as a gap in the original specification as this can have requirement and design implications.

We also identify what the goals are for the nurse and technician stakeholders. This helps to clarify what roles they would play in the context of the pacemaker lifecycle. During the ambulatory stage, the patient will be spending a lot of time with the nurse to monitor health and functionality of the pacemaker. The technician may be called if there is a bug detected for troubleshooting, and in the extreme cases, disposal of the pacemaker. We also add what the goals of the hospital could be, though the hospital is mostly

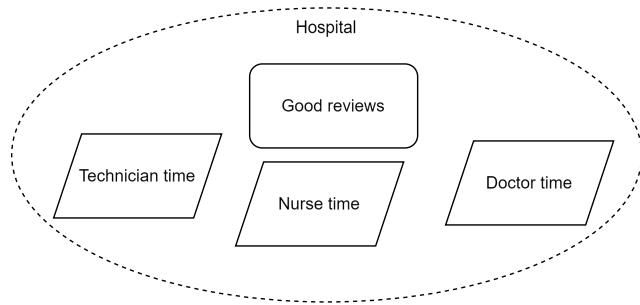


Figure 5.4: Hospital goals in the context of the pacemaker system.

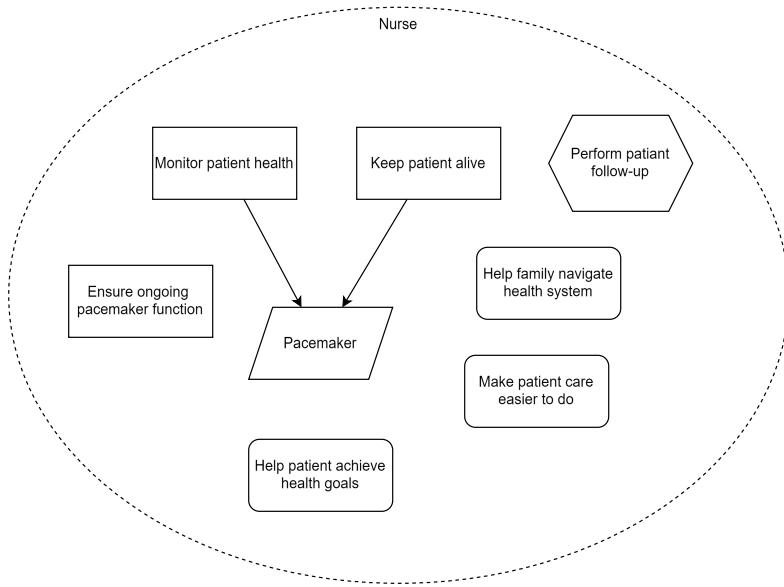


Figure 5.5: Nurse goals in the context of the pacemaker system.

where the resources are sourced. There can be more goals for the hospital in general, but for this context we limited the goals to the scope of a patient with bradycardia and the pacemaker lifecycle.

Along with the hospital, we identified another stakeholder beyond the scope of the original pacemaker requirement; patient family. The reason we include the patient family as a stakeholder is because they are also relevant to the pacemaker lifecycle as a potential resource for the patient. The patient family also will likely share the goal to help care for the patient, performed in ways that the medical team is unable to; help in the day-to-day activities of the

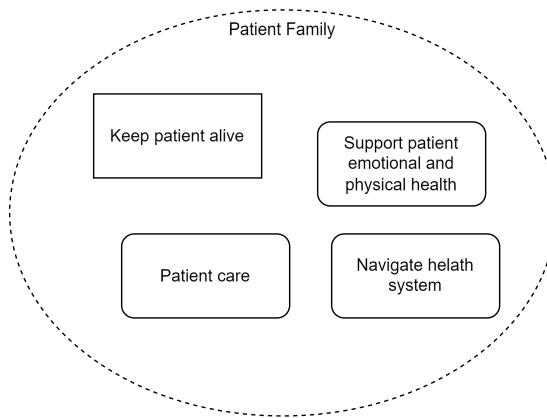


Figure 5.6: Patient family goals in the context of the pacemaker system.

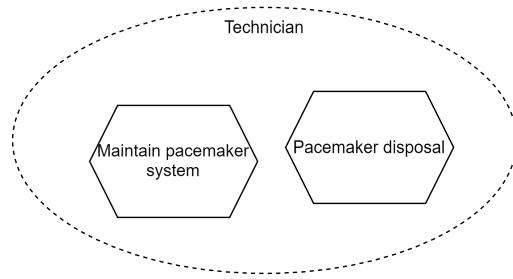


Figure 5.7: Technician goals in the context of the pacemaker system.

patient and in the community.

There are many goals we identified that are shared across stakeholders. We have them as separate bubbles for the sake of displaying in this thesis, but another potential benefit of the goal diagrams is that we can create venn diagrams that explicitly show the overlap between stakeholders and reduce copied goals. We show the combined goals of the doctor and nurse stakeholders in figure 5.8.

Thus far with the goal diagrams we have identified two more stakeholders that were not identified in the original requirement document. We have provided insight into what some of the goals are for each of the stakeholders, along with supporting beliefs, tasks, and resources. We can claim that we have reduced some uncertainty around what goals the stakeholders would

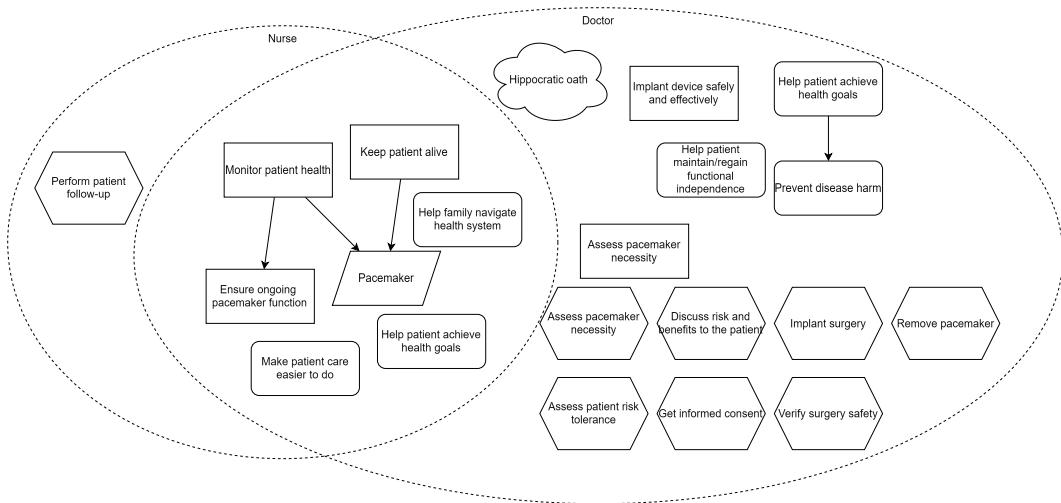


Figure 5.8: Combined nurse and doctor goals.

have in regards to the pacemaker product lifecycle. We also claim that we have provided some extra context around the stages of the pacemaker lifecycle that can help inform or support decisions for the requirements and subsequent development.

5.1.2 Use Case Diagrams

The next step we want to take is identify the use cases for each of these stakeholders, if they have uses, and compare them with the identified uses of the original requirement document. Many of the tasks (from the goal diagrams) that we have identified for the stakeholders do not require the pacemaker system to complete, but the goals do. According to the proposed methodology, we want to identify the use cases that the stakeholders will need the pacemaker system for. These use cases should also be traceable to one or more goals for the stakeholders. This traceability is important to justify the identified use cases.

In chapter 2: System Definition for the pacemaker requirements docu-

ment it defines the various components that compose the pacemaker system; the pacemaker device (also known as the pulse generator or PG), the Device Controller-Monitor and associated software, and the device leads that are implanted in the patient. Those three components become the boundaries of our UCDs. This is an easy integration point between our methodology and the existing documentation. We can save some time by reusing the system and sub-system decompositions from the original document. The document does not explicitly state what the features or use cases are for each of the devices, but there are overview sections that we can use as a starting point for determining what the use cases are.

The PG device overview provides the following information:

- Monitor and regulate patient heart rate.
- Detect and provide therapy for bradycardia conditions.
- Provide programmable, single- and dual-chamber, rate-adaptive pacing, both permanent and temporary.
- In adaptive rate modes, an accelerometer is used to measure the physical activity resulting in a sensor indicated rate for pacing the heart.
- Programming and interrogation via bi-directional telemetry from the DCM. Allows physician to change operating mode of parameters of the device non-invasively after implantation.
- Provide the following historical data:
 - Sensor output data.
 - Atrial and ventricular rate histograms.

- In conjunction with DCM, provide diagnostic features including:
 - Real time telemetry markers
 - EGMs
 - P and R wave measurements
 - Lead impedance
 - Battery status tests

The overview for the DCM contains two sub-systems; a hardware platform and pacemaker application software. It also outlines the role of the DCM. The DCM is the primary implant, pre-discharge EP support, and follow-up device for the pacemaker system. It is capable of being used in the operating room, the doctor's office and the electrophysiology lab. The DCM communicates with the PG using its software and hardware sub-systems. The DCM overview provides the following information:

- Program and interrogate a pacemaker.
- Command delivery of “Pace-Now” pace.
- Acquire and show diagnostics (history) and lead signal measurement information.
- Acquire and show sensor history and trending information.
- Show visible and audible indications of communication between the DCM and PG device, including beeping and LED's for alerting the operator to error conditions.

- Acquire and show multi-channel monitoring including surface electrogram and telemetered signals (e.g. EGM, annotated event markers).
- Print reports and strip charts.
- Monitor battery status.
- Output to external strip chart recorders.
- Manage windows for display of text and graphics.
- Set the date and time.

The lead system overview provides the following information:

- The lead system implanted in the patient allows the device to sense intrinsic activity of the heart's electrical signals and delivers pacing therapy to the patient's heart.
- The leads are connected to the PG via its header. All IS-1 bipolar leads are supported.

Taking into account the various overviews we can begin to create the UCDs. A helpful insight that the overviews provide that has been missed thus far is the way the sub-systems will interact with each other. In the scope of the UCDs, we can model this by treating the various sub-systems as stakeholders in the diagram environment. Interpreting the overview for the PG device we can have the UCD shown in figure 5.9. There are three actors explicitly outlined in the original specifications; the doctor, the patient, and the DCM. In this interpretation we treated sub-bullet points as mandatory use cases of the original use case. This is shown in the ‘Provide diagnostic features’ use

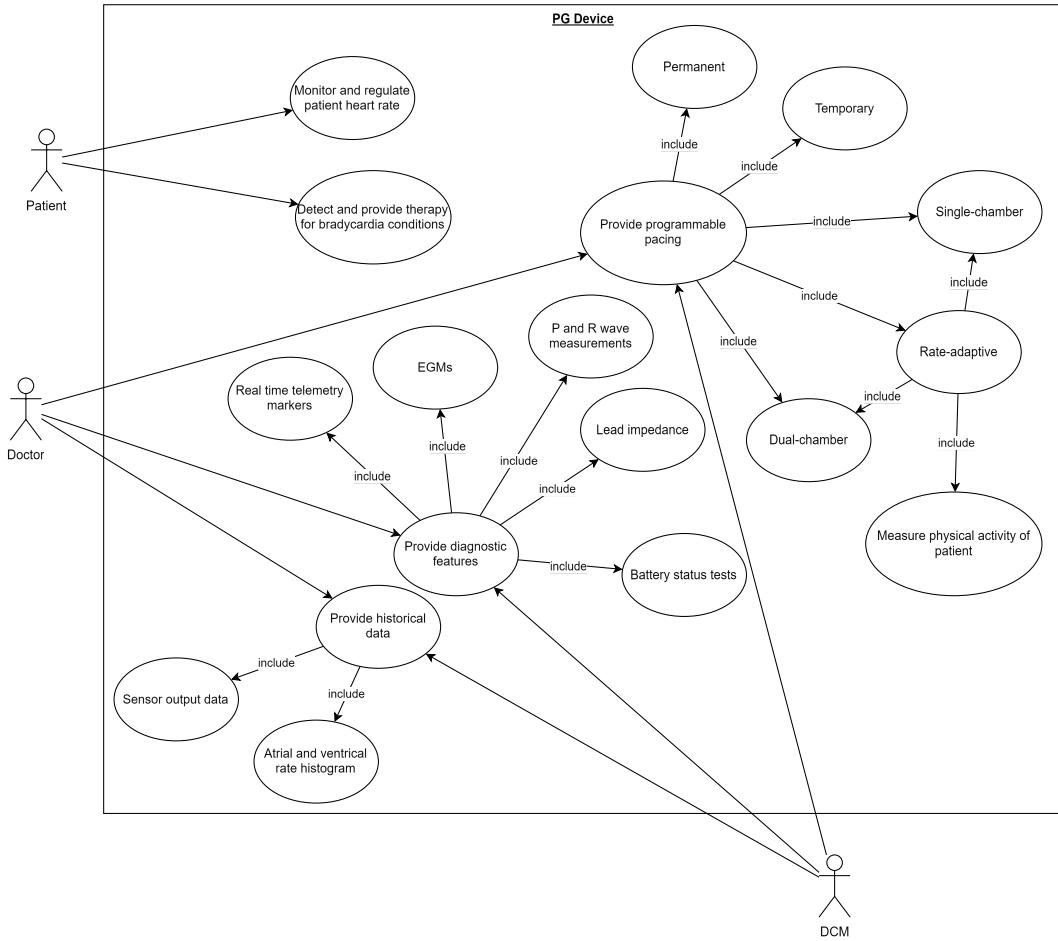


Figure 5.9: Use Case Diagram for the PG device based on the original pacemaker specification.

case and its various included use cases. The same strategy is repeated for the ‘Provide historical data’ use case.

In the original specification the programmable pacing is listed within a single bullet point. However, when we attempted to model that point as a use case we noticed it was a heavily overloaded use case. It inadvertently hid a lot of information behind a single use case. By deciding to break it up and modelling it in a UCD we explicitly state the information that was originally hidden in the specification. We can identify that both single and

dual chamber pacing will reuse by the rate adaptive part of the use case. Though not explicitly mentioned in this section of the original specification, it implies that the pacemaker should allow for both rate-adaptive and non rate-adaptive pacing. This is further supported by the table on page 17 (shown in figure 5.10) of the original specification that outlines the various operating modes for the pacemaker. We took a further liberty with the following ‘an accelerometer is used to measure the physical activity resulting in a sensor indicated rate for pacing the heart’ use case. Since it is still related to rate adaptive pacing, we model it as another included use case along the chain. Thus we can show how the doctor interacts with the ‘Provide programmable pacing’ use case and all of its included use cases along the relationship chain. These connections are not obvious from just the natural language requirements in the original document but becomes much more clear by introducing the UCD as part of the process. It forced us to consider who was interacting with the use cases, and how the use cases identified may have implied knowledge hidden within that can be beneficial to further requirement specification or design. We have identified a possibility for reusable artifacts extremely early in the development lifecycle. We have also refined our use cases with minimal overhead to the original document.

In figure 5.11 we have created another UCD enhanced by the knowledge gained from the goal diagram. We identified an additional actor that has use cases; the nurse. The nurse is never explicitly stated as an actor when using the PG device in the original specification. In our goal diagram we have identified that there is a lot of overlap between the nurse and the doctor stakeholders. Aside from programming the pacemaker, which has been explicitly stated in the specification to be done by the doctor, the nurse also has their unique

Category	I Chambers Paced	II Chambers Sensed	III Response To Sensing	IV (optional) Rate Modulation
Letters	O–None A–Atrium V–Ventricle D–Dual	O–None A–Atrium V–Ventricle D–Dual	O–None T–Triggered I–Inhibited D–Tracked	R–Rate Modulation

Table 2: Bradycardia Operating Modes

Figure 5.10: Intended operation modes for the pacemaker system.

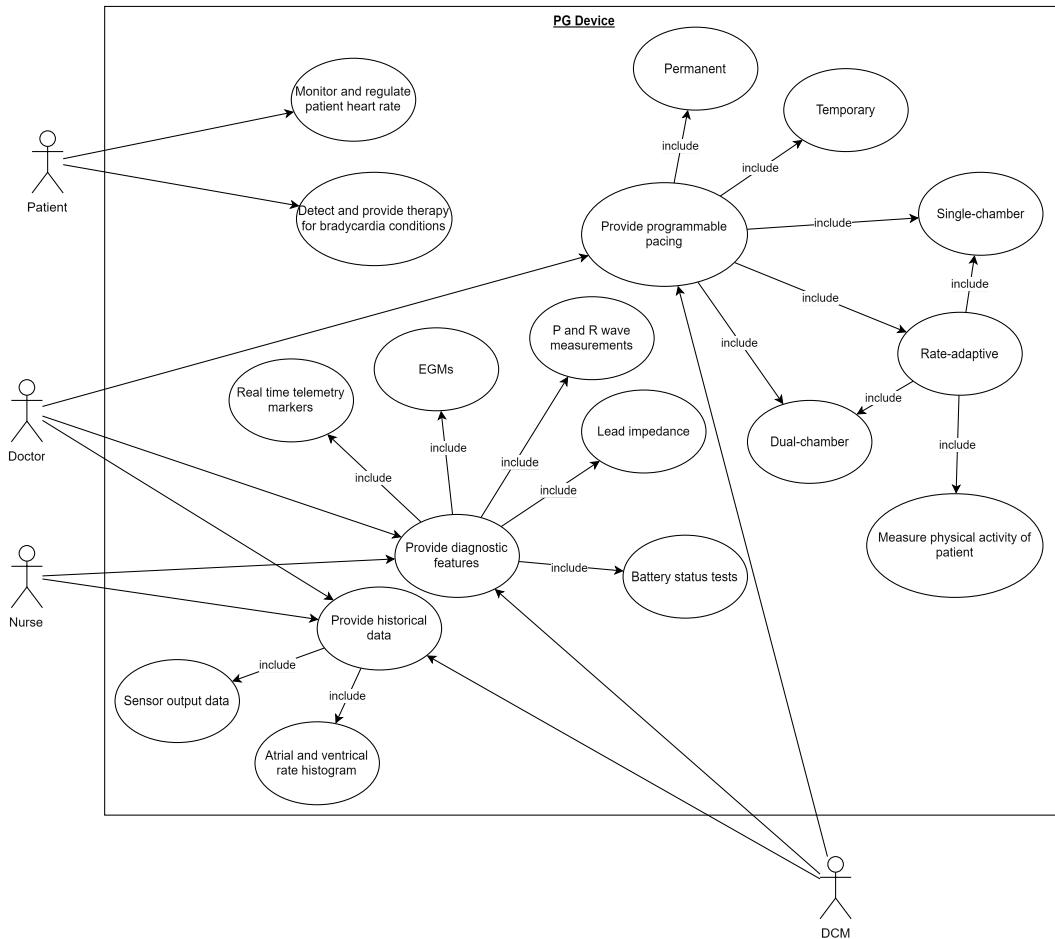


Figure 5.11: Use Case Diagram for the PG device based on the original pacemaker specification enhanced with information from the goal diagram knowledge capture.



Figure 5.12: Use Case Diagram for the DCM based on the original pacemaker specification. This diagram combines the hardware and software sub-systems into a single boundary.

task of patient follow-up that requires they be able to access the historic and diagnostic data from the patient.

Next we see in figure 5.12 a possible interpretation of the information in the DCM overview in a UCD. While the original specification does specify that the DCM sub-system is further decomposed into a hardware platform and software application, it does not make a distinction between the two when specifying the use cases. In this initial modelling attempt, we try to define the use cases for the DCM with its components combined within the system boundary. We also discovered another new actor when creating this UCD: the Strip Chart

Recorder. This actor is related to the singleton use case ‘output to external strip-chart recorders’. This UCD was particularly challenging to create due to all the conjunctive sentences in the specified features. In particular, the ‘Acquire and show diagnostics (history) and lead signal measurement information’ use case is difficult to fully capture who this use case is for. The language used is confusing as by using the word diagnostics the statement seems like this feature is for the Technician. However, because it then specifies history in brackets, similar to what is specified in the ‘Acquire and show sensor history and trending information’ use case, we ultimately decided it is more likely to be used by the doctor and nurse actors. Interestingly, likely due to the overlap between the two stakeholders, the doctor and nurse actors in this model are associated to all the same use cases. As this is supported by their overlapping goals we have a stronger confidence in the correctness of this model.

The DCM system is more challenging to have confidence in the UCD due to the combined nature of the hardware and software sub-systems specified in the requirements. They are heavily coupled which makes identifying what is a software use case and what is a hardware use case difficult as they are currently written. More information may come to light during the refinement however at this point in development it is difficult to gain more insight into how the sub-systems could be decoupled.

The leads sub-system is relatively straightforward to model given the limited use cases that were identified. It identified two actors that will use it, the patient and the PG device. However, we have included the DCM as an actor for this system as well. We justify this due to the fact that the DCM being the interface that allows human actors like the doctor or nurse to program or interrogate the pacemaker.

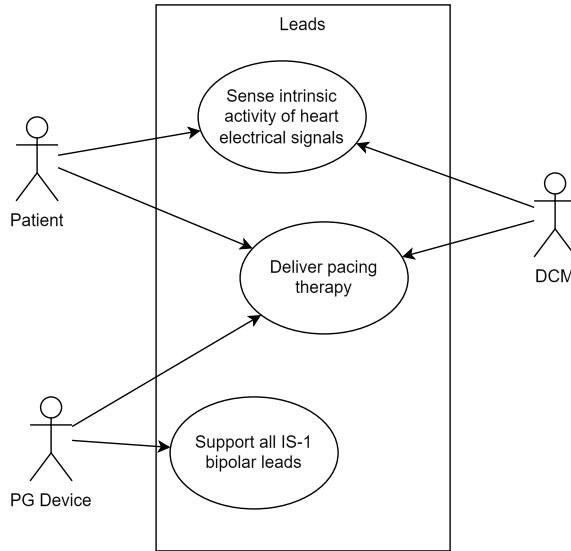


Figure 5.13: Use Case Diagram for the leads based on the original pacemaker specification.

Overall the exercise of creating the UCDs for the pacemaker has given us a lot of insight that would have otherwise been overlooked using just the original specification. We have identified potentially reusable parts of the system in the rate adaptive pacing. We have identified additional actors that were otherwise implied or overlooked in the original specification. We have discovered potential pain points in the high coupling of the DCM sub-systems. Finally we have identified interactions between the various components of the pacemaker system overall. We also noticed during this process that there is a lot of interpretation involved when creating the models. This can introduce issues in consistency as the results are heavily reliant on the knowledge and experience of the engineers executing this process.

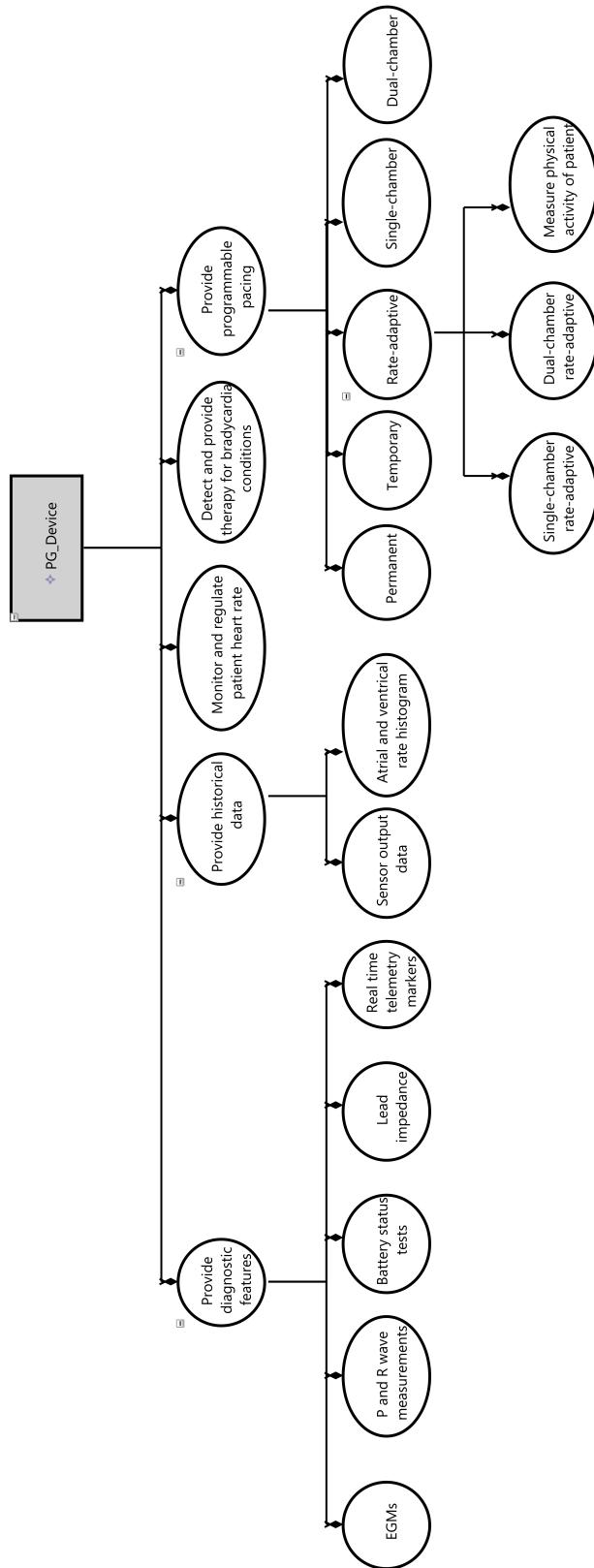


Figure 5.14: Feature model based on the UCD for the PG_Device.
95

5.2 Product Family Development

One of the benefits of this methodology becomes apparent at this stage of development. Leveraging the equivalency definition we have between use cases and features, all of the use cases that we used in our UCDs are also the features that we will use in our feature models. In figure 5.14 we show a potential product family based on the UCD for the PG_Device. We say potential due to the possibility of variability in how engineers can interpret the information from the UCD and goal diagram to create the feature model. For example, single and dual chamber pacing appears twice within the product family due to the necessity to have both rate-adaptive and non-rate-adaptive pacing. There is overlap between the two features, but we put the rate-adaptive versions of the pacing under that feature and the non-rate-adaptive pacing on another level of the model.

Based on the UCD for the PG_Device, every use case is necessary for the system, and we identified the reuse of single and dual pacing between rate-adaptive modes. As a result, there is no potential for variability in the final product since all the features are mandatory to complete the system. This provides a starting point for developing the product. We do not take this model as complete as we expect to learn more about the system through the requirement refinement process. As we refine the model this present opportunities for iteration between the phases of this approach. As more features are identified, there can be benefits in iterating between the feature model and UCD to explore how they relate to the actors and to other features in the system.

5.2.1 Composition

Due to the current definition of the metamodel for CyclicL, feature composition is slightly different. Since we state that all requirements are scoped by their owned features, we do not support cross cutting requirements at this time. When a feature composition is required to define a product based on the feature model, it becomes more of a checklist rather than more traditional composition weaving. This is by design as we want each feature to be self contained with respect to their requirements. This does not exclude dependencies between features however. The validation of feature dependencies can still happen during the composition process if properly implemented. At this time, feature composition is not an automated process in CyclicL. It requires manual intervention by the user to select what features from the product family to add to a product variant they define.

5.3 Requirement Engineering and Refinement

With a prototype product family created from the UCD, we can start to explore the use cases and refining them towards more proper requirements. We do this by taking the use cases and refining them as a user story. We use the following structure when refining a use case to a user story:

- **Use case:** Provide programmable pacing
- **Actor:** Doctor, DCM
- **Precondition:** Pacemaker PG_Device is active and can connect to the DCM.

- **Trigger:** Doctor wants to program pacemaker to mitigate patient bradycardia symptoms via DCM interface.

- **Success Scenario:**

1. Doctor accesses PG_Device via DCM.
2. Doctor verifies they are connected to correct PG_Device.
3. Doctor selects the desired pacing operation mode.
4. Doctor sets the various parameters based on the chosen pacing operation mode (pace pulse characteristics, rate sensing settings, delays etc).
5. Doctor pushes programmed mode and parameters to the PG_Device.
6. Doctor disconnects DCM from PG_Device.
7. Doctor logs off from DCM.

- **Secondary Scenarios:**

- Doctor connects to the wrong PG_Device via DCM.
- Doctor makes a mistake programming the PG_Device via DCM.
- Communication fails or is interrupted during push from DCM to PG_Device.
- Doctor is unable to access the PG_Device via DCM.

- **Success Postcondition:** PG_Device is correctly programmed and pacing the patients heart correctly and safely.

The steps outlined in the success scenario are the functional requirements for the use case. They outline what needs to happen for the actor to accomplish whatever task is outlined. The outcomes in the secondary scenarios

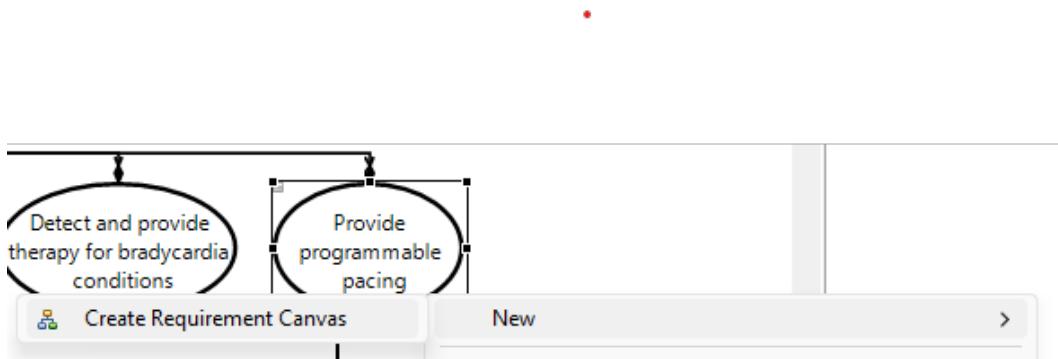


Figure 5.15: When right-clicking on a feature, a context menu appears allowing users to create a requirement canvas that is embedded in the selected feature.

provide alternative results of actors actions. These alternative outcomes can be safety hazards, security hazards, system malfunctions, or user errors. The secondary scenarios outline the justifications for mitigating requirements in order to improve the chance that the actor will arrive to the success postcondition, in this case that the PG_Device is correctly programmed and pacing the patients heart correctly and safely. In the context of our methodology, mitigating requirements are most commonly categorized into safety, constraint, or qualitative requirements. At this time we have not ruled out that a mitigating requirement may lead to a functional requirement, however based on our experiences we support the convention that they are a better fit for the non-functional requirement types.

With these requirements identified, we can begin modelling them in the requirement canvas. In CyclicL, the requirement canvas is embedded in the features of the product family. This is the feature-requirement encapsulation that we outlined in the methodology. In figure 5.15, we show how a user can create an embedded requirement canvas.

In figure 5.16 we show all the requirements created in the model as previously identified to match the refinement of the user story. In figure 5.17,

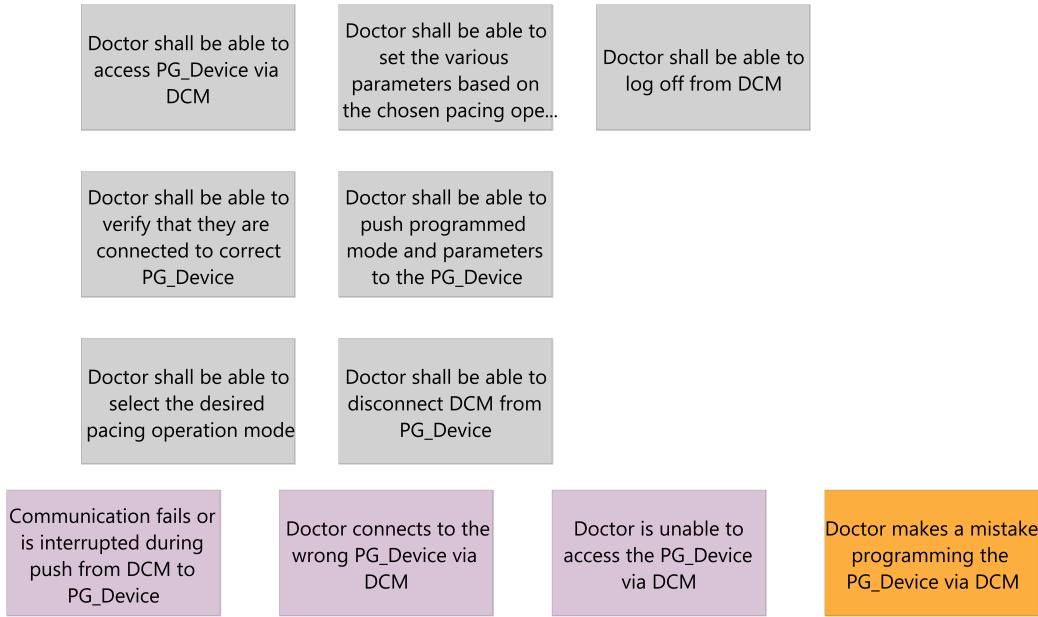


Figure 5.16: Requirements for the ‘Provide programmable pacing’ use case.

figure 5.18, and figure 5.19 we also show the Gherkin specifications for the expected behavior of the first three requirements.

The requirement entities in the canvas have facilities that allow us to specify our requirements in greater details. Due to the user stories that we used to refine the use cases into the requirements necessary to satisfy the successful postcondition, there is an implied ordering that is generated for the functional requirements. We can see this ordering reflected in the Gherkin specifications. The postconditions for one requirement often be reused as the preconditions for the next. The tool does not make this relationship explicit, but the specification does allow for it. This provides some flexibility to the engineers using this methodology for how they want to specify relationships between their requirements.

Another benefit of these ordered requirements is that it can help with planning feature development. Engineers can identify dependencies between

```

Given{
    Precond one: "The pacemaker PG_Device is active and can
                  connect to the DCM"
}
When{
    Event one: "Doctor wants to program pacemaker to mitigate
                patient bradycardia symptoms via DCM interface"
    Event two: "The doctor attempts to access the PG_Device via
                DCM"
}
Then{
    Postcond one: "The doctor is able to connect to a PG_Device"
}

```

Figure 5.17: Gherkin specification for the first requirement of the PG_Device.

```

Given{
    Precond one: "The doctor is able to connect to a PG_Device"
}
When{
    Event one: "The doctor wants to verify they are connected to
                the correct device"
}
Then{
    Postcond one: "The PG_Device shall communicate to doctor via
                  DCM identification to allow doctor to verify device
                  identity"
}

```

Figure 5.18: Gherkin specification for the second requirement of the PG_Device.

```

Given{
    Precond one: "The doctor has verified they are connected to
                  the correct device"
}
When{
    Event one: "The doctor wants to change or select the desired
                pacing mode"
}
Then{
    Postcond one: "The doctor shall be able to select from the
                   available pacing options"
}

```

Figure 5.19: Gherkin specification for the third requirement of the PG_Device.

requirements within a feature more easily and plan what stages of development need to be completed sequentially and which ones can happen in parallel. This lends confidence to the claim of this methodology helping with FDD. The feature becomes well defined by the requirements, each requirement has the opportunity to be well specified by the engineers, and plans can be made to tackle the order of development to complete the feature. This includes the satisfaction of functional, qualitative, constraint, and safety requirements alike.

Another part of the requirement canvas is the ability to assign test cases and reviewers to specified requirements. At this point in development, the purpose of those model elements is primarily bookkeeping. A further benefit of using the Gherkin syntax are the many existing libraries that support it testing. At this time, we do not natively support testing in the tool. We do however, want to be able to keep track of which requirements have been satisfied and which ones have not. Additionally, we also want to keep track of which ones have been reviewed and approved. As part of the requirement engineering process

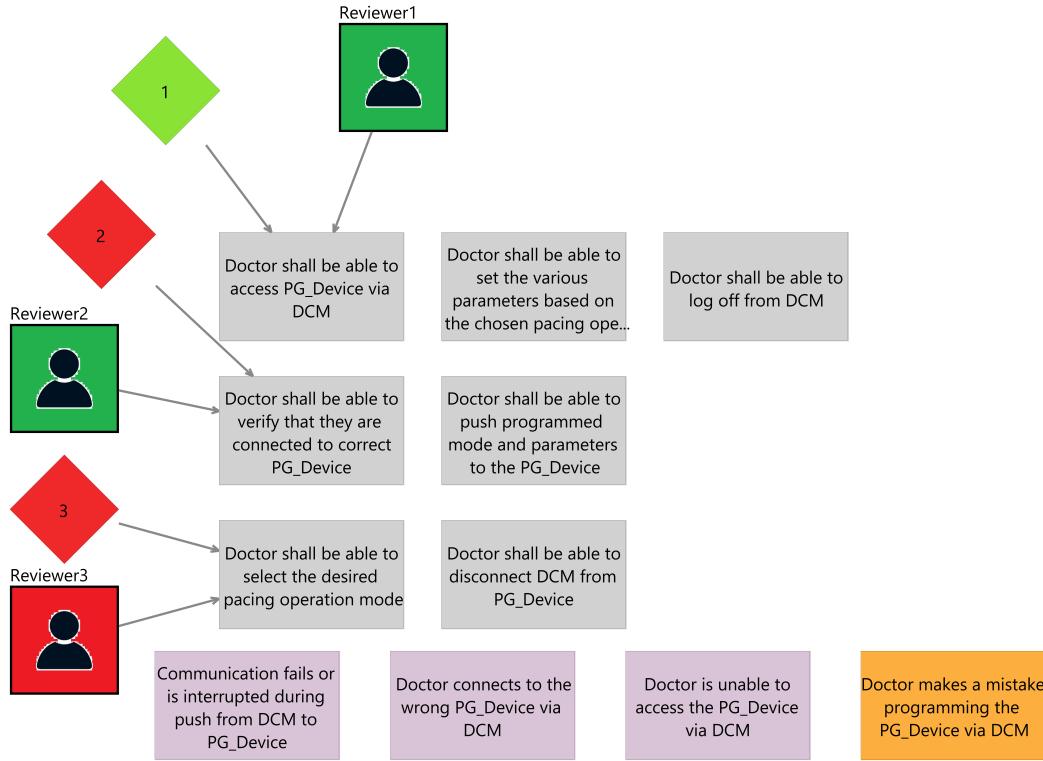


Figure 5.20: Various stages of approval and testing for the requirements.

Requirement ID	Requirement Description	Requirement Rationale	Requirement Specification	Test Case	Design Element	Test Case ID	Test Case Description	Test Case Type	Test Case Passed	Approved	Comments	Review
1	Doctor shall be able to access PG_Device via DCM	Given	Provide programmable pacing	<input type="checkbox"/>	(1)	(1)	(1)	Test	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	[Review]	[Review]
2	Doctor shall be able to verify that they are connected to correct PG_Device	Given	Provide programmable pacing	<input type="checkbox"/>	(2)	(2)	(2)	Test	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	[Review]	[Review]
3	Pacing options are AAF, V	Given	Provide programmable pacing	<input type="checkbox"/>	(3)	(3)	(3)	Test	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	[Review]	[Review]

Figure 5.21: Showing how traceability is automatically maintained throughout requirement approval and testing.

it is important to keep track of which requirements are ready to development, and which ones are still undergoing refinement. We show how this can appear for the pacemaker in figure 5.20. Naturally, once these model elements are added and connected to requirements, they are automatically updated in the traceability matrix as well.

At this stage we have created a functional behavioural specification for the “Provide programmable pacing” use case beyond what is in the original specification. We have justified our requirements using the use cases from the UCD which are in turn supported by our goal diagrams. We have also

identified qualitative and safety requirements that were not identified in the original specification. We have demonstrated flexibility in how we can specify these requirements using the Gherkin syntax. This has shown the benefit of a semi-formal approach as we can get an order to our functional requirements if desired to help with project planning and scoping. This also shows potential integration points with existing FDD development methods. The natural language also makes the specification process relatively easy to learn and approach while allowing meaningful specifications to help development. Finally, we have shown that all of these efforts automatically create a traceability matrix for us with no extra steps or inputs by us during this process execution.

5.4 Iterations and Incrementation

For the purpose of this methodology, iterative changes are thought of as cycles between development phases (features and requirements, requirements and design, design and implementation). The idea for this definition is that there are lessons learned when working through a subsequent development phase that can inform and create cause for repeating a process from a previous phase. In the case of this methodology, if there is a need identified during the requirement phase, it may create a cause for repeating one of the processes in feature modelling phase.

Incremental changes are thought of as changes within a single development phase. Currently, CyclicL supports maintenance of the traceability matrices through incremental changes as we have defined them. As engineers work through the various steps within a development phase, they will be constantly adding, modifying, or removing artifacts until they reach a satisfactory result.

We consider all of these incremental changes until they get enough completed to move onto the next phase of development.

There are three traceability matrices that are supported in CyclicL; project traceability, feature-feature traceability, and feature-requirement traceability. Project traceability and feature-requirement traceability are defined to focus on traceability of the requirement entities. Project traceability is the traceability matrix for the entire project. Feature-requirement traceability is a traceability matrix that is scoped to a single feature. The Feature-feature traceability matrix is a tabular representation of the feature model that is defined within the project.

These are a second syntactic representation of the modelling environments present within the tool. Since they are semantically equivalent, any change to the requirements in the requirement canvas or features in the feature modelling environment are automatically reflected in their respective traceability matrix. The tabular environments, though somewhat limited, are also capable of defining new requirements. The limitations are that test cases and reviewers cannot be created in the tabular environment. This is by design as it forces users to manually create those traces explicit in the modelling environment and reflect on what test cases are being created and who is approving of the requirements.

We currently have limited support for iterative changes according to our definition. While changes that are made to an existing model will naturally be automatically reflected in the traceability matrices, the impact of the change cannot currently be assessed. Based on the metamodel for CyclicL, however, the mechanisms to support change impact analysis are all present. This is due to the nature of traceability that is inherently built into the model and the

way we require explicit connections between elements in order for a model to be what is consider syntactically valid by our current conventions.

CyclicL is also currently only scoped to feature and requirement modelling. It does not currently support UCDs or goal diagrams within itself. As a result, we do not automatically generate and maintain traceability to those phases of development. By having all the modelling environments connected within CyclicL, we will have more confidence in being able to trace iterative changes and change impact more comprehensively through the various stages of development.

Chapter 6

Future Work

While the results of our proposed methodology and tooling are promising, there are some definite short-comings at present. While CyclicL is self-contained with regards to features and requirements, there is no traceability between the UCD and goal diagram to the tool. This presents challenges to comprehensive traceability from requirements through the features to their sources. This also causes a gap in analysis when performing iterative changes as there are no facilities currently that can enable change impact analysis between feature modelling and the UCD. As users learn more about their product we anticipate and encourage them to iterate through the various phases of development in this methodology and we want to support that computationally as well.

Using this methodology to support safety and security assurance case development is another avenue we would like to explore. As this methodology highly emphasizes traceability, there could be opportunities for supporting assurance case development. This possibility is part of the reason for the “Safety” type for the requirement canvas. This would include the traceability to the UCDs and goal diagrams in CyclicL.

The implementation of the tool, CyclicL, is still incomplete as we have only demonstrated the capability of the minimum viable product. There are many functions we will need to add to have a more complete tool. There are currently no constraints in the tool to ensure that all user created models are valid according to the metamodel. Thus, there is no way to validation tools to ensure that the engineering artifacts created are semantically or syntactically meaningful, useful, or computable.

Thus there is potential for users to create semantically meaningless models even if the syntax is shown as correct. We are currently relying on conventions and good engineering to create semantically meaningful models but this is very difficult to maintain at scale within a company or industry. Another lane of development would be in allowing users to create these models textually as well as graphically. Very often adoptability of a tool, especially an MDE tool is difficult as many engineers are unfamiliar with MDE techniques and approaches. Creating a textual environment for creating these models may help with usability and approachability of CyclicL for use in industry. Next, we would want to implement a method of consistency checking within the tool. This would remove a lot of the burden from the engineer to ensure that their features are consistent and their requirement are consistent between features. This is a unique challenge as the requirements are specified using the Gherkin behavioural approach and thus may require us to re-evaluate what type of specification language we implement within the requirement canvases of CyclicL.

Other future work tasks that have been mentioned throughout this thesis include integration with existing test compilers, integration with previous feature modelling tools, and general UI enhancements. As there are many Gherkin

test environments, building a bridge to allow integration with CyclicL would be helpful to verification tracking and traceability. Integration with another feature modelling tool, such as FeatureIDE, would be serve as an exploration for the impact of the feature-requirement hierarchy defined in this thesis on existing feature weaving and composition techniques.

Chapter 7

Conclusion

What comes first, the requirement or the feature? In this thesis we have attempted to outline an answer to this question. incidentally, and perhaps unsurprisingly, the answer is both and it depends. Both realities are possibilities for companies in industry as they startup new product development from scratch or try to reuse previous work for a new model. So we support both of them.

The ultimate answer however, is neither. The stakeholder use cases and goals come first as we conduct a domain analysis and problem space exploration. We have shown that following Meyer’s requirements engineering approach we can leverage both informal and formal MDE techniques to perform a domain and problem space analysis. We found that goal diagrams were an easy to use, informal approach to capturing knowledge of what stakeholders really want to do which helped to inform why they would want to use a product we develop. The UCD was important for establishing how the would use our product. Identifying what interfaces the stakeholders would need from the product to accomplish their goals. Goal diagrams and UCDs formed the core

of our informal modelling techniques for domain analysis and problem space exploration.

Feature modelling and our requirement canvas leveraged more formal approaches to modelling and problem space exploration as with CyclicL we could enforce syntactic and semantic rules when creating the models. Equating features and use cases also helped to establish requirements for our features as we refined the use cases into user stories. With our proposed feature-requirement hierarchy, those requirements are immediately scoped and owned by the features making an easy transition from problem space exploration to requirement specification. We have demonstrated both formal and informal approaches to domain analysis, problem space exploration, and requirement development while using MDE techniques. We have explored several processes for accomplishing these tasks in multiple environments. Finally, we have demonstrated how this methodology can be used to identify gaps in existing documentation and how to improve requirements specification and identification. Thus, we conclude that we have satisfied **RQ.4**

FDD is a well established development strategy. A major contribution of this thesis is the proposed feature-requirement hierarchy that establishes features own their requirements. Another contribution is how we have equated use cases from UCDs to features in PLE. Our methodology refines the use cases into user stories to create our requirements for each use case. Due to having equated use cases and features, those requirements are thus also applicable to the feature. When combined with the proposed feature-requirement hierarchy we end up with well defined requirements for each feature. An added bonus to this we discovered was a natural ordering that formed in the requirements due to the nature of using user stories as a refinement process. These all

contribute to having well defined and scoped features, and potentially aiding in project planning due to the implied ordering of requirements from the user stories. These are all possible ways to enhance FDD development environments, satisfying **RQ.1**.

One of the main contributions of our methodology is the feature-requirement encapsulation. By defining this hierarchy, we have shown how it facilitates traceability between features and requirements. We also removed ambiguity in the relationship between features and requirements. We have demonstrated how this hierarchy enables increased granularity of traceability through our implementation of CyclicL. We were able to define a formal metamodel to capture this hierarchy and implement a tool to show satisfiability of this proposal. Through CyclicL, we exposed the benefit of this hierarchical relationship between features and requirements in the semi-automated maintenance and generation of traceability matrices between features and requirements. This was completed as a by product of following the methodology and naturally using CyclicL to create models and specify our requirements. We thus got the traceability matrices without any manual intervention or extra steps in the engineering process. We conclude that this reduces much of the time and effort that would traditionally have been spent manually creating or maintaining engineering data traceability, satisfying **RQ.2**.

Finally, CyclicL has shown the potential benefits of a PLE tool that is self contained with requirements as part of the tool in the context of iterative and incremental development. We have shown that we can support incremental development of both feature models and requirement models in CyclicL. By leveraging cro:EMFEclipse Modeling Framework (EMF) and Sirius, we were able to show the possibilities enabled by a tool that will maintain traceability

through both requirement changes and product changes without dependencies on external support. While there are still some limitations in the current tool capabilities to support iterative changes, we have demonstrated how future development can continue to address these short-comings. Due to the currently limitations in the tool, we conclude that we have partially satisfied **RQ.3** with CyclicL as a tool that supports iterative and incremental development of traceability in parallel to feature and requirement development. We give it a partial designation as there are still limitations on how CyclicL can support iterative changes under our current definition of a change between development phases.

Overall our methodology has shown demonstrable improvements on an existing requirement document, provided avenues for potential improvements in requirement engineering and feature modelling, and shown potential for integrating with existing development practice. CyclicLand its supporting methodology are ripe for further exploration and continued development to support safety-critical development in further domains and potentially as a more generalized approach to requirement engineering across industries.

Appendices

Appendix A

A.1 Boston Scientific Pacemaker Requirements

PACEMAKER System Specification

Copyright ©2007 Boston Scientific

January 3, 2007

DISCLAIMER: This document is provided for academic purposes only, specifically demonstration of formal methodologies and engineering education.

Boston Scientific disclaims any use of information contained herein for any product whatsoever, disclaims safety, disclaims efficacy, and disclaims fitness for purpose. Any person or entity who uses this document for anything assumes complete responsibility for its use.

Permission to copy is granted, provided the copyright declaration and this disclaimer are included.

Please direct any comments or questions to: **PACEMAKER@sql.mcmaster.ca**

Contents

1	Introduction	5
1.1	Scope	5
1.2	Acronyms	5
2	System Definition	7
2.1	System Overview	7
2.2	System Components	7
2.2.1	Device Overview	8
2.2.2	Device Controller-Monitor (DCM) Overview	8
2.2.3	Lead System Overview	9
2.3	Indications and Contraindications	9
2.4	User Characteristics and Operational Environment	10
2.4.1	Hospitals/Physician Users	10
2.4.2	Device Usage with Other Equipment	10
2.5	System Operational Life Phases	10
2.5.1	Pre-Implant Phase	11
2.5.2	Implant	11
2.5.3	Predischarge Follow-up	12
2.5.4	Routine Follow-up	12
2.5.5	Ambulatory	12
2.5.6	Explant	12
2.5.7	Disposal	12
3	System Requirements	13
3.1	Model Type	13
3.2	Device Controller-Monitor (DCM)	13
3.2.1	DCM User Languages	13
3.2.2	DCM User Interface	13
3.2.3	DCM Utility Functions	14
3.2.4	Printed Reports	14
3.2.5	Strip Chart Recording Support	15
3.2.6	DCM-PG Telemetry	15
3.3	Lead Support	15
3.4	Pacing Pulse	16
3.4.1	Pulse Amplitude	16
3.4.2	Pulse Width	16
3.4.3	Rate Sensing	16
3.4.4	Sensitivity Adjustment	16
3.5	Bradycardia Operating Modes	16
3.5.1	No Response To Sensing (O)	17
3.5.2	Triggered Response To Sensing (T)	17
3.5.3	Inhibited Response To Sensing (I)	17
3.5.4	Tracked Response To Sensing (D)	17
3.6	Bradycardia States	17

3.6.1	Permanent State	17
3.6.2	Temporary Bradycardia Pacing	17
3.6.3	Pace-Now State	18
3.6.4	Magnet State	18
3.6.5	Power-On Reset (POR) State	18
3.7	Magnet Test	19
3.8	Implant Data	19
4	Diagnostics	21
4.1	Measured Data	21
4.2	P and R Wave Measurements	21
4.3	Lead Impedance Measurement	21
4.4	Battery Status	21
4.5	Threshold Test	22
4.6	Bradycardia History	22
4.6.1	Rate Histograms	22
4.6.2	Rate Trending	23
4.6.3	Recording Duration and Time Stamp	24
4.6.4	Sensor Trending	24
4.7	Real-time Electrograms	24
4.7.1	Electrogram Viewing	24
4.8	Real-time Electrogram Event Marker Annotations	25
4.8.1	Atrial Markers: AS AP AT TN	26
4.8.2	Ventricular Markers: VS VP PVC TN	26
4.8.3	Marker Modifiers: () -Hy -Sr ↑↓	26
4.8.4	Augmentation Markers: ATR-Dur ATR-FB ATR-End PVP→	26
4.9	Faults and Error Handling	27
4.9.1	Faults	27
4.9.2	Errors	27
5	Bradycardia Therapy	28
5.1	Lower Rate Limit (LRL)	28
5.2	Upper Rate Limit (URL)	29
5.3	Atrial-Ventricular (AV) Delay	29
5.3.1	Paced AV Delay	29
5.3.2	Sensed AV Delay	29
5.3.3	Dynamic AV Delay	29
5.3.4	Sensed AV Delay Offset	30
5.4	Refractory Periods	30
5.4.1	Ventricular Refractory Period (VRP)	30
5.4.2	Atrial Refractory Period (ARP)	30
5.4.3	Post Ventricular Atrial Refractory Period (PVARP)	30
5.4.4	Extended PVARP	30
5.4.5	Refractory During AV Interval	31
5.5	Noise Response	31
5.6	Atrial Tachycardia Response (ATR)	31

5.6.1	Atrial Tachycardia Detection	31
5.6.2	ATR Duration	31
5.6.3	ATR Fallback	32
5.7	Rate-Adaptive Pacing	32
5.7.1	Maximum Sensor Rate (MSR)	32
5.7.2	Activity Threshold	32
5.7.3	Response Factor	32
5.7.4	Reaction Time	33
5.7.5	Recovery Time	33
5.8	Hysteresis Pacing	33
5.9	Rate Smoothing	33
A	Programmable Parameters	34
B	Measured Parameters	35

List of Tables

1	Model Type and Lead Port	13
2	Bradycardia Operating Modes	17
3	Battery Status and Therapy Availability	22
4	Intervals Associated with Histogrammed Events	23
5	Event Marker Annotations	25
6	Programmable Parameters for Bradycardia Therapy Modes	28
7	Programmable Parameters	34
8	Measured Parameters	35

1 Introduction

This System Specification for PACEMAKER defines the functions and operating characteristics, identifies the system environmental performance parameters, and characterizes anticipated uses of the system.

1.1 Scope

This document identifies the functions that the system must perform and provides a description of these functions and their primary interactions.

1.2 Acronyms

AP	Atrial Pace
AS	Atrial Sense
ARP	Atrial Refractory Period
ATR	Atrial Tachycardia Response
AV	Atrial-to-Ventricular
BOL	Beginning Of (battery) Life
BPM	Beats Per Minute
cc	Cardiac Cycle(s)
CCI	Cardiac Cycle Interval
DCM	Device Controller-Monitor
ECG	Electrocardiogram, external heart signals
EGM	Electrogram, internal heart signals
EOL	End Of (battery) Life
EP	Electrophysiology, electrophysiologist
ERN	Elective Replacement Near
ERT	Elective Replacement Time
HRL	Hysteresis Rate Limit
ICD	Implantable Cardio-Defibrillator
IS-1	Industry Standard lead type 1
LRL	Lower Rate Limit
MSR	Maximum Sensor Rate
NSR	Normal Sinus Rhythm
OR	Operating Room
PG	Pulse Generator
POR	Power-On Reset
PMT	Pacemaker-Mediated Tachycardia
ppm	Pulses Per Minute
PVARP	Post-Ventricular Atrial Refractory Period
PVC	Premature Ventricular Contraction
SIR	Sensor Indicated Rate
SRD	Sustained Rate Duration
URL	Upper Rate Limit
VP	Ventricular Pace

VS Ventricular Sense
VRP Ventricular Refractory Period

2 System Definition

2.1 System Overview

This PACEMAKER System Specification describes the PACEMAKER-specific programming application and pulse generator (PG). The PACEMAKER system supports the following needs of patients that require bradycardia pacing support:

- Implantation
- Ambulatory
- Follow-up
- Explantation

The PACEMAKER system:

- Provides dual chamber, rate adaptive bradycardia pacing support
- Provides historical data on device performance
- Provides user diagnostics through brady analysis functions

The bradycardia analysis functions permit the following pacing measurements and tests to be performed:

- Lead impedance
- Pacing threshold
- P and R wave measurement
- Battery status
- Temporary brady pacing
- Motion sensor trending

2.2 System Components

The PACEMAKER system consists of three major components:

- Device (also called the pulse generator or PG)
- Device Controller-Monitor (DCM) and associated software
- Leads

A standard cardiac “donut” magnet is a minor system component.

2.2.1 Device Overview

The device monitors and regulates a patient's heart rate.

The device detects and provides therapy for bradycardia conditions.

The device provides programmable, single- and dual-chamber, rate-adaptive pacing, both permanent and temporary.

In adaptive rate modes, an accelerometer is used to measure physical activity resulting in a sensor indicated rate for pacing the heart.

The device is programmed and interrogated via bi-directional telemetry from the Device Controller-Monitor (DCM). This allows the physician to change the operating mode or parameters of the device non-invasively after implantation.

The device provides the following history data:

- Sensor output data
- Atrial and ventricular rate histograms.

The device, in conjunction with the DCM, provides diagnostic features including:

- Real time telemetry markers
- EGMs
- P and R wave measurements
- Lead impedance
- Battery status tests

2.2.2 Device Controller-Monitor (DCM) Overview

The Device Controller-Monitor (DCM) is the primary implant, pre-discharge EP support, and follow-up device for the PACEMAKER system. The DCM is capable of being used both in the OR, physician's office, and the EP lab. The DCM communicates with the PG using a communication protocol and supporting hardware. The DCM consists of the following:

- A hardware platform
- PACEMAKER application software

The DCM has the following features:

1. program and interrogate a PACEMAKER
2. command delivery of "Pace-Now" pace
3. acquire and show diagnostics (history) and lead signal measurement information.
4. acquire and show sensor history and trending information.

5. show visible and audible indications of communication between the DCM and device, including beeping and LED's for alerting the operator to error conditions.
6. acquire and show multi-channel monitoring including surface electrocardiogram and telemetered signals (e.g. EGM, annotated event markers)
7. print reports and strip charts.
8. monitor battery status.
9. output to external strip-chart recorders.
10. receive cursor positioning and button inputs.
11. manage windows for display of text and graphics.
12. set the date and time.

2.2.3 Lead System Overview

The lead system implanted in the patient allows the device to sense intrinsic activity of the heart's electrical signals and delivers pacing therapy to the patient's heart.

The leads are connected to the PG via its header. All IS-1 bipolar leads are supported.

2.3 Indications and Contraindications

The PACEMAKER system is indicated for patients exhibiting chronotropic incompetence and who would benefit by increased pacing rates concurrent with physical activity. Generally accepted indications for long-term cardiac pacing include, but are not limited to, sick sinus syndrome; chronic sinus arrhythmias, including sinus bradycardia, sinus arrest, and sinoatrial (S-A) block; second- and third-degree AV block; bradycardia-tachycardia syndrome; bundle branch block; and carotid sinus syndrome.

Patients who demonstrate hemodynamic improvement from atrioventricular synchrony should be considered for one of the dual-chamber or atrial pacing modes. Dual-chamber modes are specifically indicated for treatment of conduction disorders that require restoration of rate and of atrioventricular synchrony, including varying degrees of AV block; low cardiac output or congestive heart failure related to bradycardia; and certain tachyarrhythmias.

Use of the PACEMAKER pulse generator in DDDR and VDDR modes may be contraindicated (1) in patients with chronic atrial tachyarrhythmias (atrial fibrillation or flutter), which may trigger ventricular pacing, or (2) in the presence of slow retrograde conduction that induces pacemaker-mediated tachycardia (PMT) which cannot be controlled by reprogramming selective parameter values. In DDDR, DDIR, and AAIR modes, atrial pacing may be ineffective in the presence of chronic atrial fibrillation or flutter or an atrium that does

not respond to electrical stimulation. In addition, the presence of clinically significant conduction disturbances may contraindicate the use of atrial pacing. Unipolar pacing is contraindicated for patients with an implanted cardioverter defibrillator (ICD) because it may cause unwanted initiation or inhibition of ICD therapy.

2.4 User Characteristics and Operational Environment

Patients with arrhythmias may be users of portions of the system such as the magnet.

2.4.1 Hospitals/Physician Users

The PACEMAKER device and leads are implanted by physicians and hospital staff with varying degrees of experience. Follow-up of the patients is typically performed by nurses or technicians under the supervision of a following physician.

2.4.2 Device Usage with Other Equipment

The device functions with the following equipment:

- The ECG monitors.
- The DCM.

The device meets industry standards for electrical safety. The device functions with the following ancillary equipment present in ORs and EP labs which can be sources of EMI or direct energy interference:

- A fluoroscope.
- Anesthesia machines.
- Patient water blankets.
- Electrosurgical devices.
- External defibrillators.
- Blood pressure monitors.
- Pulse oximeters.

2.5 System Operational Life Phases

The system's life cycle is shown below. Each phase in the cycle is described in more detail following.

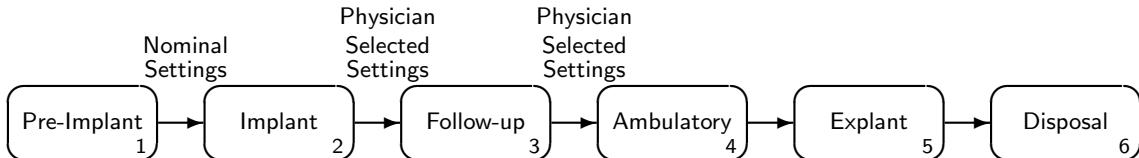


Figure 1: PACEMAKER Life Cycle

2.5.1 Pre-Implant Phase

The system is manufactured using good manufacturing practices as specified by the FDA and other appropriate regulatory bodies. During manufacturing, nominal settings for the PG are set.

2.5.2 Implant

During implant, the PACEMAKER device is placed into the patient. During implant, the DCM is used to:

1. Interrogate the system
2. Review battery status
3. Test the PACEMAKER in the patient
4. Setup the appropriate parameters
5. Program the system before implantation
6. Evaluate ventricular and atrial lead signal amplitudes, impedances, and pacing thresholds.

The procedure to implant a PACEMAKER device and lead system consists of these steps:

1. Checking status of all equipment to be used during implant
2. Implanting the lead system
3. Evaluating lead signals using a pacing stimulus analyzer
4. Programming the nonimplanted system
5. Forming the implantation pocket and tunnel the leads
6. Connecting to the patient leads
7. Testing the system sensing and pacing efficacy
8. Implanting the system

2.5.3 Predischarge Follow-up

During the predischarge follow-up test, the following procedures may be performed via telemetry using the DCM:

1. Interrogating the device and obtaining bradycardia sensing and pacing data
2. Reprogramming to final pre-discharge value
3. Printing the follow-up report for the patient's chart

2.5.4 Routine Follow-up

The programming system is capable of performing the following procedures during the routine follow-up:

1. Interrogating the device
2. Checking the battery status
3. Checking the brady status
4. Performing P and R wave measurements
5. Performing pacing threshold and lead impedance tests
6. Reviewing the activity sensor history and rate histograms
7. Printing a follow-up report
8. If parameter values are changed during the follow-up visit, the new setting is verified by viewing the "Session Net Change" report
9. Clearing the Histograms

2.5.5 Ambulatory

The Pacing/Sensing functions will be available in the Ambulatory stage of the device life cycle.

2.5.6 Explant

Once the device is explanted, it is sanitized and returned to its manufacturer. A fault analysis is performed if applicable. The state of the device, when it is explanted, is a function of any fault that may have occurred and/or the state of the battery at the time of explant.

2.5.7 Disposal

For disposal, the device is sent back to its manufacturer. The device should not be destroyed by incineration because the device contains batteries that can explode when subjected to heat.

3 System Requirements

All detailed development requirements are defined in this section and the next, Bradycardia Therapy. Each requirement is defined by a sentence containing the word shall or by each item in a list of items.

For the Device Controller-Monitor (DCM) only those items that will be supported as part of the application software development for the PACEMAKER will be defined.

Programmable ranges and tolerances are provided in Appendix A.

3.1 Model Type

The PACEMAKER model type shall support single and dual chamber rate adaptive pacing.

Model	Pacemaker Designation	Functionality	Connector
DR1	DR	DDD full function with accelerometer	Dual-in-line, 3.2 mm (IS-1)

Table 1: Model Type and Lead Port

3.2 Device Controller-Monitor (DCM)

3.2.1 DCM User Languages

The application is available in the following languages: English, Danish, Dutch, French, German, Spanish, Italian, and Swedish.

3.2.2 DCM User Interface

The user interface is capable of the following:

1. The user interface shall be capable of utilizing and managing windows for display of text and graphics.
2. The user interface shall be capable of processing user positioning and input buttons.
3. The user interface shall be capable of displaying all programmable parameters for review and modification.
4. The user interface shall be capable of visually indicating when the DCM and the device are communicating.
5. The user interface shall be capable of visually indicating when telemetry is lost due to the device being out of range.

6. The user interface shall be capable of visually indicating when telemetry is lost due to noise.
7. The user interface shall be capable of visually indicating when a different PACEMAKER device is approached than was previously interrogated.

3.2.3 DCM Utility Functions

1. The About function displays the following:
 - Application model number
 - Application software revision number currently in use
 - DCM serial number
 - Institution name
2. The Set Clock function shall set the date and time of the device.
3. The New Patient function shall allow a new device to be interrogated without exiting the software application.
4. The Quit function shall end a telemetry session.

3.2.4 Printed Reports

The following parameter and status reports are available at the user's request:

1. A Bradycardia Parameters Report shall be available.
2. A Temporary Parameters Report shall be available.
3. An Implant Data Report shall be available.
4. A Threshold Test Results Report shall be available.
5. A Measured Data Report shall be available.
6. A Marker Legend Report shall be available.
7. A Session Net Change Report shall be available.
8. A Final Report shall be available. This will consist of the Measured Data, Threshold Test, Trending, Histograms, Implant Data, and Net Change reports.

The following bradycardia diagnostic reports are available at the user's request:

1. A Rate Histogram Report shall be available.
2. A Trending Report shall be available.

Each report shall contain the following header information:

1. Institution name
2. Date and time of report printing
3. Device model and serial number
4. DCM serial number
5. Application model and version number
6. Report name

3.2.5 Strip Chart Recording Support

1. The DCM shall be capable of displaying real time and surface ECG data, which shall be accomplished using the DCM's internal monitor.
2. The system shall be capable of displaying up to three Real-Time traces (2 Telemetered, 1 Surface ECG), along with an annotation for display of event markers, in a scrollable fashion.
3. The DCM shall use the DCM's internal strip chart recorder to provide a means of printing combinations of real time data.
4. The DCM shall be capable of printing real time telemetered data and a surface ECG.
5. The printer shall be capable of simultaneously printing up to three real-time traces, along with full annotation for display of event markers.

3.2.6 DCM-PG Telemetry

The DCM shall either:

- use an inductive telemetry wand to communicate with the pulse generator, maintaining consistent communication over the range of 0 cm to 5 cm between the wand and the pulse generator; or,
- use some other medium, such as RF or ultrasound, that is safe and legal to use, for maintaining consistent telemetry with an implanted medical device.

3.3 Lead Support

1. The Atrial Bipolar Pace/Sensing lead system type shall be supported.
2. The Ventricular Bipolar Pace/Sensing lead system type shall be supported.

3. The system shall operate normally with atrial pace/sense leads between 100 and 2500 ohms.
4. The system shall operate normally with the ventricular pace/sense leads between 100 and 2500 ohms.

Note: all pacing amplitudes and pulse widths in this document are specified using a 750 ohm load.

3.4 Pacing Pulse

The device shall output pulses with programmable voltages and widths (atrial and ventricular) which provide electrical stimulation to the heart for pacing.

3.4.1 Pulse Amplitude

The atrial and ventricular pacing pulse amplitudes shall be independently programmable.

3.4.2 Pulse Width

The atrial and ventricular pacing pulse width shall be independently programmable.

3.4.3 Rate Sensing

Rate sensing shall be accomplished using bipolar electrodes and sensing circuits. All rate detection decisions shall be based on the measured cardiac cycle lengths of the sensed rhythm. Rate shall be evaluated on an interval-by-interval basis.

3.4.4 Sensitivity Adjustment

A means shall be provided for the physician to manually adjust the sensing threshold of the device for both the ventricular and atrial sense channels.

3.5 Bradycardia Operating Modes

The following bradycardia operating modes shall be programmable: Off, DDDR, VDDR, DDIR, DOOR, VOOR, AOOR, VVIR, AAIR, DDD, VDD, DDI, DOO, VOO, AOO, VVI, AAI, VVT and AAT.

OVO, OAO, ODO, and OOO shall be available in temporary operation.

Note: sometimes, “X” is used as a “don’t care” to identify a set of modes; DXXX are the dual-chamber paced modes; OXO are the sensing-only modes; XXT are the triggered modes.

Category	I Chambers Paced	II Chambers Sensed	III Response To Sensing	IV (optional) Rate Modulation
Letters	O–None A–Atrium V–Ventricle D–Dual	O–None A–Atrium V–Ventricle D–Dual	O–None T–Triggered I–Inhibited D–Tracked	R–Rate Modulation

Table 2: Bradycardia Operating Modes

3.5.1 No Response To Sensing (O)

Pacing without sensing is asynchronous pacing. During asynchronous pacing, paces shall be delivered without regard to senses.

3.5.2 Triggered Response To Sensing (T)

During triggered pacing, a sense in a chamber shall trigger an immediate pace in that chamber.

3.5.3 Inhibited Response To Sensing (I)

During inhibited pacing, a sense in a chamber shall inhibit a pending pace in that chamber.

3.5.4 Tracked Response To Sensing (D)

During tracked pacing, an atrial sense shall cause a tracked ventricular pace after a programmed AV delay, unless a ventricular sense was detected beforehand.

3.6 Bradycardia States

The following bradycardia states shall be available: Permanent, Temporary, Pace-Now, Magnet, and Power-On Reset (POR). Operating states shall be mutually exclusive.

3.6.1 Permanent State

The permanent pacing state is the normal state of operation of the device. The normal pacing parameters programmed shall be used in the permanent brady state.

3.6.2 Temporary Bradycardia Pacing

The temporary brady pacing state is independent of other pacing functions. The temporary brady parameters programmed shall be used in the temporary

brady state. The temporary state shall be capable of being used to temporarily test various system parameters or provide patient diagnostic testing. Temporary brady pacing shall be terminated by one of the following: breaking the telemetry link, a Pace-Now pace, or a DCM command to the device to cancel temporary pacing.

3.6.3 Pace-Now State

Commanded emergency bradycardia pacing (Pace-Now) shall be available.

The Pace-Now Pace parameter values are as follows:

1. The mode Pace-Now pace parameter shall have a value of VVI.
2. The lower rate limit Pace-Now pace parameter shall have a value of 65 ppm ± 8 ms.
3. The amplitude Pace-Now pace parameter shall have a value of 5.0 V ± 0.5 V.
4. The pulse width Pace-Now pace parameter shall have a value of 1.00 ms ± 0.02 ms.
5. The ventricular refractory Pace-Now pace parameter shall have a value of 320 ms ± 8 ms.
6. The ventricular sensitivity shall have a value of 1.5 mV.
7. The first Pace-Now pacing pulse shall be issued within two cardiac cycles plus 500 ms from the time of the last user action required to activate the Pace-Now state.
8. Once initiated, Pace-Now pacing shall continue until the DCM changes the device pacing mode.

3.6.4 Magnet State

The Magnet State is used during the Magnet Test.

3.6.5 Power-On Reset (POR) State

A Power-on-reset (POR) state shall be entered when the battery voltage drops so low that PG operation is not predictable. All functions shall be disabled until the battery voltage exceeds the POR trip voltage. Above this trip voltage, the PG enters the POR state which is used to power-up the PG system to a known state and set of parameters.

The POR parameter values are as follows:

1. The mode POR pace parameter shall have a value of VVI.

2. The lower rate limit POR pace parameter shall have a value of 65 ppm ± 8 ms.
3. The amplitude POR pace parameter shall have a value of 5.0 V ± 0.5 V.
4. The pulse width POR pace parameter shall have a value of 0.5 ms ± 0.02 ms.
5. The ventricular refractory POR pace parameter shall have a value of 320 ms ± 8 ms.
6. The ventricular sensitivity shall have a value of 1.5 mV.

3.7 Magnet Test

The magnet can be used to determine the battery status of the device. A standard cardiac donut magnet shall be detected by the device at a distance of 2.5 cm between the center of the labeled surface of the device and the surface of the magnet.

When the magnet is in place, the device shall:

1. Pace asynchronously with a fixed pacing rate. The device mode shall be AOO if previous mode was AXXX, VOO if previous mode was VXXX, DOO if previous mode was DXXX, or OOO if previous mode was OXO modes.
2. At BOL the magnet rate shall be 100 ppm. At ERN the magnet rate shall decrease to 90 ppm. At ERT the magnet rate shall decrease further to 85 ppm. During post-ERT operation the rate interval may gradually decrease as the battery voltage continues to decrease.
3. When the magnet is removed the device shall automatically assume pretest operation.
4. The magnet mode shall have the capability to be programmed OFF, so that it will ignore magnet detection.

3.8 Implant Data

The device shall be capable of storing the following information in device memory:

1. The device shall be capable of storing the device model and serial number, and implant date information.
2. The device shall be capable of storing the lead implant date and polarity information.
3. The device shall be capable of storing pacing thresholds and P and R-wave amplitude information.

4. The device shall be capable of storing the pacing lead impedance information.
5. The device shall be capable of storing the patient's indications for pacing.

4 Diagnostics

The system provides the following diagnostic tools:

- Measured Data diagnostic tools shall be provided.
- Threshold Test diagnostic tools shall be provided.
- Rate Trending and Histograms diagnostic tools shall be provided.
- Real-time data diagnostic tools shall be provided.

4.1 Measured Data

Measured Data tolerances are shown in Appendix B.

4.2 P and R Wave Measurements

The device shall allow for DCM-commanded measurement of P and R waves.

4.3 Lead Impedance Measurement

Lead impedance measurements works as follows:

1. The device shall allow for manual measurement capability.
2. DCM commanded lead impedance shall be made with the device in the temporary state.
3. Lead impedance measurements shall be conducted at a default value of 5.5 V.

4.4 Battery Status

Battery status information includes the following:

1. Monitoring voltage information shall be provided.
2. Battery Status indicator information shall be provided.
3. Last interrogation date information shall be provided.
4. The battery status for the device shall be used to predict the following battery status levels:

Battery Status Level	Status	Functionality
Beginning of Life	BOL	Fully functional
Elective Replacement Near	ERN	Fully functional
Elective Replacement Time	ERT	Non-rate-adaptive single chamber modes only. Temporary programming, automatic threshold testing, measured data, electrograms, and event markers disabled.
Elective Replacement Past	ERP	Same as ERT, except pacing rate gradually decreases as battery voltage decreases.

Table 3: Battery Status and Therapy Availability

4.5 Threshold Test

Auto threshold tests work as follows:

1. An automatic pacing threshold test in AAI, VVI, and DDD modes shall be available on command of the DCM for both pulse width and amplitude measurements.
2. The test starts at a user-selectable amplitude or pulse width. After approximately every fourth cardiac cycle, the DCM automatically steps down the amplitude or pulse width one setting.
3. The user is instructed to terminate the test by removing the telemetry wand or selecting the "Stop" button when loss of capture is observed.
4. The last six test results will be displayed (each chamber) on the screen and printed report.
5. Programmable DDD/AAI back-up pacing with a programmable rate shall be available for atrial testing, and programmable DDD/VVI back-up pacing with programmable rate shall be available for ventricular testing.

4.6 Bradycardia History

To assist adjustment of pacing parameters, bradycardia history is retained in PGs for viewing with DCM.

4.6.1 Rate Histograms

The operator interface of the system shall be able to display histograms of pacing rate and intrinsic rate distributions from a histogram recording period.

Histogram data are recorded as follows:

1. Distributions shall be recorded for all paced atrial events.
2. Distributions shall be recorded for all sensed atrial events.
3. Distributions shall be recorded for all paced ventricular events.
4. Distributions shall be recorded for all sensed ventricular events.
5. The number of premature ventricular contractions (PVCs) and atrial tachycardia response episodes shall be recorded and displayed.
6. The recording period for a rate histogram shall be the time since the rate histograms were last reset to the present.
7. The rate histograms shall be resettable (clearing previously recorded data) via telemetry.
8. All rate histograms shall be cleared simultaneously.
9. The intervals associated with the histogrammed events shall begin and end at the events specified in the following table:

Histogram Event	Beginning Event	Ending Event
Sensed Atrial	Refractory or non-refractory atrial sense or atrial pace	Refractory or non-refractory atrial sense
Paced Atrial	Atrial pace or non-refractory atrial sense	Atrial pace
Sense Ventricular	Non-refractory ventricular sense or ventricular pace	Non-refractory ventricular sense
Paced Ventricular	Ventricular pace or non-refractory ventricular sense	Ventricular pace

Table 4: Intervals Associated with Histogrammed Events

4.6.2 Rate Trending

The system shall be configurable to record and display the following data items separately or concurrently over a programmable duration and storage method:

1. Pacing Rate
2. Sensor Data

4.6.3 Recording Duration and Time Stamp

The recording duration shall be programmed to one of the following options:

1. Fixed: Start recording now and stop when available storage is full (time stamped at beginning).
2. Continuous: Circular buffer keeping the latest information (time stamped at end).

The recording duration shall be time stamped as indicated above.

The system shall display only the programmable durations that are applicable for the current pacing mode.

4.6.4 Sensor Trending

The system shall provide off-line prediction analysis of sensor indicated rate with or without intrinsic rate for the synchronized data collected.

4.7 Real-time Electrograms

Real-time internal electrograms shall be made available from

1. The atrial and ventricular sense/pace leads.
2. A surface electrogram.

The real-time electrogram transmission shall be re-initiated if the telemetry link was broken during the transmission of electrograms and then reestablished.

4.7.1 Electrogram Viewing

The user shall have the option of viewing the electrograms

1. On the screen
2. Through a printed copy

The user shall have the option of selecting which electrograms are viewed and the resolution utilized.

Internal electrogram (EGM) options provided are the following:

1. An atrial internal electrogram option shall be provided.
2. A ventricular internal electrogram option shall be provided.
3. An atrial and ventricular internal electrogram option shall be provided.

For the surface electrocardiogram (ECG), the user shall have the capability to select

1. The gain utilized (0.5X, 1X, or 2X)

2. Whether high pass filtering is on

For the internal electrogram (EGM), the display gain shall

1. Be selectable (0.5X, 1X, or 2X)
2. Apply to all channels

4.8 Real-time Electrogram Event Marker Annotations

The capability shall exist to print and display annotated event marker abbreviations listed below on the real-time electrogram. These markers shall be a combination of intrinsic cardiac and device-related events.

1. Each marker shall show time-or-occurrence, accurate to 1 ms, since the most recent, non-refractory event in that chamber.
2. At most one atrial marker will occur each 1 ms.
3. At most one ventricular marker will occur each 1 ms.
4. At most one augmentation marker will occur each 1 ms.

Marker Abbreviation	Description
AS	Atrial Sensed
AP	Atrial Paced
AT	A-Tachy Sense
VS	Ventricular Sensed
VP	Ventricular paced
PVC	Premature Ventricular Contraction
TN	Noise Indication
()	During Refractory
↑	Rate Smoothing Up
↓	Rate Smoothing Down
-Sr	Motion Sensor Rate
-Hy	Hysteresis Rate Pace
ATR-Dur	ATR Onset Started
ATR-FB	ATR Fallback Started
ATR-End	ATR Fallback Ended
PVP→	PVARP Extension

Table 5: Event Marker Annotations

4.8.1 Atrial Markers: AS AP AT TN

Atrial markers are generated for events in the atrium.

AS Atrial Sense not faster than URL

AP Atrial Pace

AT Atrial Tachycardia, sense faster than URL

TN Noise indication

4.8.2 Ventricular Markers: VS VP PVC TN

Ventricular markers are generated for events in the ventricle.

VS Ventricular Sense

VP Ventricular Pace

PVC Premature Ventricular Contraction

A ventricular sense is deemed to be a premature ventricular contraction if there has been no atrial event since the previous ventricular event.

TN Noise indication

4.8.3 Marker Modifiers: () -Hy -Sr ↑ ↓

Atrial or ventricular markers may have modifiers that change their meaning.

() Sense during refractory

-Hy Hysteresis pace

-Sr Sensor-rate pace

↑ Up-rate smoothing pace

↓ Down-rate smoothing pace

4.8.4 Augmentation Markers: ATR-Dur ATR-FB ATR-End PVP→

During atrial tachycardia response (ATR) and PVARP extension, augmentation markers are generated synchronously with atrial or ventricular markers.

ATR-Dur Atrial tachycardia detected, duration started

ATR-FB ATR fallback started when tachycardia lasts for duration

ATR-End ATR ended because tachycardia ceased, or fallback period expired

PVP→ PVC caused PVARP extension

4.9 Faults and Error Handling

4.9.1 Faults

DCM malfunctions shall be indicated.

4.9.2 Errors

Errors indicated are the following:

1. Parameter interactive limit errors shall be indicated.
2. Printer errors shall be indicated.

5 Bradycardia Therapy

User programmable parameters are provided for controlling the delivery of patient-tailored, bradycardia therapy. These parameters are described in the following subsections; which parameters are meaningful with which pacing mode are listed in Table 6, Programmable Parameters for Bradycardia Therapy Modes.

Parameter	A	V	A	A	V	V	V	D	D	D	A	A	V	V	V	D	D	D	D
	A	V	O	A	O	V	D	O	D	D	O	A	O	V	D	O	D	D	D
	T	T	O	I	O	I	D	O	I	D	O	I	O	I	D	O	I	D	R
Lower Rate Limit	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Upper Rate Limit	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Maximum Sensor Rate												X	X	X	X	X	X	X	X
Fixed AV Delay						X	X	X	X						X	X	X	X	X
Dynamic AV Delay						X			X					X				X	
Sensed AV Delay Offset								X											X
Atrial Amplitude	X		X	X				X	X	X	X	X				X	X	X	X
Ventricular Amplitude		X			X	X	X	X	X				X	X	X	X	X	X	X
Atrial Pulse Width	X		X	X				X	X	X	X	X				X	X	X	X
Ventricular Pulse Width		X			X	X	X	X	X				X	X	X	X	X	X	X
Atrial Sensitivity	X			X				X	X			X					X	X	
Ventricular Sensitivity		X				X	X		X	X			X	X			X	X	
VRP		X			X	X		X	X				X	X			X	X	
ARP	X			X				X	X			X					X	X	
PVARP	X		X					X	X			X					X	X	
PVARP Extension						X			X						X			X	
Hysteresis			X		X				X		X		X					X	
Rate Smoothing		X		X	X				X		X		X	X				X	
ATR Duration						X			X						X			X	
ATRFallback Mode						X			X						X			X	
ATR Fallback Time							X		X						X			X	
Activity Threshold										X	X	X	X	X	X	X	X	X	X
Reaction Time										X	X	X	X	X	X	X	X	X	X
Response Factor										X	X	X	X	X	X	X	X	X	X
Recovery Time										X	X	X	X	X	X	X	X	X	X

Table 6: Programmable Parameters for Bradycardia Therapy Modes

5.1 Lower Rate Limit (LRL)

The Lower Rate Limit (LRL) is the number of generator pace pulses delivered per minute (atrium or ventricle) in the absence of

- Sensed intrinsic activity.
- Sensor-controlled pacing at a higher rate.

The LRL is affected in the following ways:

1. When Rate Hysteresis is disabled, the LRL shall define the longest allowable pacing interval.
2. In DXX or VXX modes, the LRL interval starts at a ventricular sensed or paced event.
3. In AXX modes, the LRL interval starts at an atrial sensed or paced event.

5.2 Upper Rate Limit (URL)

The Upper Rate Limit (URL) is the maximum rate at which the paced ventricular rate will track sensed atrial events. The URL interval is the minimum time between a ventricular event and the next ventricular pace.

5.3 Atrial-Ventricular (AV) Delay

The AV delay shall be the programmable time period from an atrial event (either intrinsic or paced) to a ventricular pace.

In atrial tracking modes, ventricular pacing shall occur in the absence of a sensed ventricular event within the programmed AV delay when the sensed atrial rate is between the programmed LRL and URL.

AV delay shall either be

1. Fixed (absolute time)
2. Dynamic

5.3.1 Paced AV Delay

A paced AV (PAV) delay shall occur when the AV delay is initiated by an atrial pace.

5.3.2 Sensed AV Delay

A sensed AV (SAV) delay shall occur when the AV delay is initiated by an atrial sense.

5.3.3 Dynamic AV Delay

If dynamic, the AV delay shall be determined individually for each new cardiac cycle based on the duration of previous cardiac cycles. The previous cardiac cycle length is multiplied by a factor stored in device memory to create the dynamic AV delay.

The AV delay shall vary between

1. A programmable maximum paced AV delay
2. A programmable minimum paced AV delay

5.3.4 Sensed AV Delay Offset

The Sensed AV Delay Offset option shall shorten the AV delay following a tracked atrial sense.

Depending on which option is functioning, the sensed AV delay offset shall be applied to the following:

1. The fixed AV delay
2. The dynamic AV delay

5.4 Refractory Periods

To avoid false sensing, refractory periods follow events during which senses in the affected chamber are ignored. To show that a sense was ignored due to refractory, its marker is displayed in parentheses.

5.4.1 Ventricular Refractory Period (VRP)

The Ventricular Refractory Period shall be the programmed time interval following a ventricular event during which time ventricular senses shall not inhibit nor trigger pacing.

5.4.2 Atrial Refractory Period (ARP)

For single chamber atrial modes, the Atrial Refractory Period (ARP) shall be the programmed time interval following an atrial event during which time atrial events shall not inhibit nor trigger pacing.

5.4.3 Post Ventricular Atrial Refractory Period (PVARP)

The Post Ventricular Atrial Refractory Period shall be available in modes with ventricular pacing and atrial sensing. The Post Ventricular Atrial Refractory Period shall be the programmable time interval following a ventricular event when an atrial cardiac event shall not 1. Inhibit an atrial pace. 2. Trigger a ventricular pace.

5.4.4 Extended PVARP

The Extended PVARP works as follows:

1. When Extended PVARP is enabled, an occurrence of a premature ventricular contraction (PVC) shall cause the pulse generator to use the Extended PVARP value for the post-ventricular atrial refractory period following the PVC.
2. The PVARP shall always return to its normal programmed value on the subsequent cardiac cycle regardless of PVC and other events. At most one PVARP extension shall occur every two cardiac cycles.

5.4.5 Refractory During AV Interval

The PG shall also be in refractory to atrial senses during the AV interval. In this context, refractory means the pacemaker does not track or inhibit based on the sensed activity.

5.5 Noise Response

In the presence of continuous noise the device response shall be asynchronous pacing.

5.6 Atrial Tachycardia Response (ATR)

The Atrial Tachycardia Response prevents long term pacing of a patient at unacceptably high rates during atrial tachycardia. When Atrial Tachycardia Response is enabled, the pulse generator shall declare an atrial tachycardia if the intrinsic atrial rate exceeds the URL for a sufficient amount of time.

5.6.1 Atrial Tachycardia Detection

The atrial tachycardia (AT) detection algorithm determines onset and cessation of atrial tachycardia.

1. AT onset shall be detected when the intervals between atrial senses are predominately, but not exclusively, faster than URL.
2. AT cessation shall be detected when the intervals between atrial senses are mostly, but not exclusively, faster than URL.
3. The detection period shall be short enough so ATR therapy is not unnecessarily delayed nor continued.
4. The detection period shall be long enough that occasional premature atrial contractions do not cause unnecessary ATR therapy, nor cease necessary ATR therapy upon occasional slow beats.

5.6.2 ATR Duration

ATR Duration works as follows:

1. When atrial tachycardia is detected, the ATR algorithm shall enter an ATR Duration state.
2. When in ATR Duration, the PG shall delay a programmed number of cardiac cycles before entering Fallback.
3. The Duration delay shall be terminated immediately and Fallback shall be avoided if, during the Duration delay, the ATR detection algorithm determines that atrial tachycardia is over.

5.6.3 ATR Fallback

If the atrial tachycardia condition exists after the ATR Duration delay is over, the following shall occur:

1. The PG enters a Fallback state and switches to a VVIR Fallback Mode.
2. The pacing rate is dropped to the lower rate limit. The fallback time is the total time required to drop the rate to the LRL.
3. During Fallback, if the ATR detection algorithm determines that atrial tachycardia is over, the following shall occur:
 - Fallback is terminated immediately
 - The mode is switched back to normal
4. ATR-related mode switches shall always be synchronized to a ventricular paced or sensed event.

5.7 Rate-Adaptive Pacing

The device shall have the ability to adjust the cardiac cycle in response to metabolic need as measured from body motion using an accelerometer.

5.7.1 Maximum Sensor Rate (MSR)

The Maximum Sensor Rate is the maximum pacing rate allowed as a result of sensor control.

The Maximum Sensor Rate shall be

1. Required for rate adaptive modes
2. Independently programmable from the URL

5.7.2 Activity Threshold

The activity threshold is the value the accelerometer sensor output shall exceed before the pacemaker's rate is affected by activity data.

5.7.3 Response Factor

The accelerometer shall determine the pacing rate that occurs at various levels of steady state patient activity.

Based on equivalent patient activity:

1. The highest response factor setting (16) shall allow the greatest incremental change in rate.
2. The lowest response factor setting (1) shall allow a smaller change in rate.

5.7.4 Reaction Time

The accelerometer shall determine the rate of increase of the pacing rate. The reaction time is the time required for an activity to drive the rate from LRL to MSR.

5.7.5 Recovery Time

The accelerometer shall determine the rate of decrease of the pacing rate. The recovery time shall be the time required for the rate to fall from MSR to LRL when activity falls below the activity threshold.

5.8 Hysteresis Pacing

When enabled, hysteresis pacing shall result in a longer period following a sensed event before pacing. This encourages self-pacing during exercise by waiting a little longer to pace after senses, hoping that another sense will inhibit the pace.

To use hysteresis pacing:

1. Hysteresis pacing must be enabled (not Off).
2. The pacing mode must be inhibiting or tracking.
3. The current pacing rate must be faster than the Hysteresis Rate Limit (HRL), which may be slower than the Lower Rate Limit (LRL).
4. When in AAI mode, a single, non-refractory sensed atrial event shall activate hysteresis pacing.
5. When in an inhibiting or tracking mode with ventricular pacing, a single, non-refractory sensed ventricular event shall activate hysteresis pacing.

5.9 Rate Smoothing

Rate Smoothing shall limit the pacing rate change that occurs due to precipitous changes in the intrinsic rate.

Two programmable rate smoothing parameters shall be available to allow the cardiac cycle interval change to be a percentage of the previous cardiac cycle interval:

1. Rate Smoothing Up
2. Rate Smoothing Down

The increase in pacing rate shall not exceed the Rate Smoothing Up percentage.

The decrease in pacing rate shall not exceed the Rate Smoothing Down percentage.

A Programmable Parameters

Parameter	Programmable Values	Increment	Nominal	Tolerance
Mode	Off DDD VDD DDI DOO AOO AAI VOO VVI AAT VVT DDDR VDDR DDIR DOOR AOOR AAIR VOOR VVIR	—	DDD	—
Lower Rate Limit	30-50 ppm 50-90 ppm 90-175 ppm	5 ppm 1 ppm 5 ppm	60 ppm	±8 ms
Upper Rate Limit	50-175 ppm	5 ppm	120 ppm	±8 ms
Maximum Sensor Rate	50-175 ppm	5 ppm	120 ppm	±4ms
Fixed AV Delay	70-300 ms	10 ms	150 ms	±8 ms
Dynamic AV Delay	Off, On	—	Off	—
Minimum Dynamic AV Delay	30-100 ms	10 ms	50 ms	
Sensed AV Delay Offset	Off, -10 to -100 ms	-10 ms	Off	±1 ms
A or V Pulse Amplitude Regulated	Off, 0.5-3.2V 3.5-7.0 V	0.1V 0.5V	3.5V	±12%
A or V Pulse Amplitude Unregulated	Off, 1.25, 2.5, 3.75, 5.0V	—	3.75V	—
A or V Pulse Width	0.05 ms 0.1-1.9 ms	— 0.1 ms	0.4 ms	0.2 ms
A or V Sensitivity	0.25, 0.5, 0.75 1.0-10 mV	— 0.5 mV	A-0.75 mV V-2.5 mV	±20%
Ventricular Refractory Period	150-500 ms	10 ms	320 ms	±8 ms
Atrial Refractory Period	150-500 ms	10 ms	250 ms	±8 ms
PVARP	150-500 ms	10 ms	250 ms	±8 ms
PVARP Extension	Off, 50-400 ms	50 ms	Off	±8 ms
Hysteresis Rate Limit	Off or same choices as LRL	—	Off	±8 ms
Rate Smoothing	Off, 3, 6, 9, 12, 15, 18, 21, 25%	—	Off	±1%
ATR Mode	On, Off	—	Off	—
ATR Duration	10 cardiac cycles 20-80 cc 100-2000 cc	— 20 cc 100 cc	20 cc	±1 cc
ATRFallback Time	1-5 min	1 min	1 min	±1 cc
Ventricular Blanking	30-60 ms	10 ms	40 ms	—
Activity Threshold	V-Low, Low, Med-Low, Med, Med-High, High, V-High	—	Med	—
Reaction Time	10-50 sec	10 sec	30 sec	±3 sec
Response Factor	1-16	1	8	—
Recovery Time	2-16 min	1 min	5 min	±30 sec

Table 7: Programmable Parameters

B Measured Parameters

Parameter	Increment	Tolerance
P and R wave measurements	0.1 mV	$\pm 5\%$ referred to the connector
Lead Impedance	50Ω	100 to 500Ω , \pm greater of 100Ω or 30% 500 to 2000Ω , $\pm 25\%$ 2000 to 2500Ω , $\pm 30\%$
Battery Voltage	0.01 V	$\pm 2\%$

Table 8: Measured Parameters

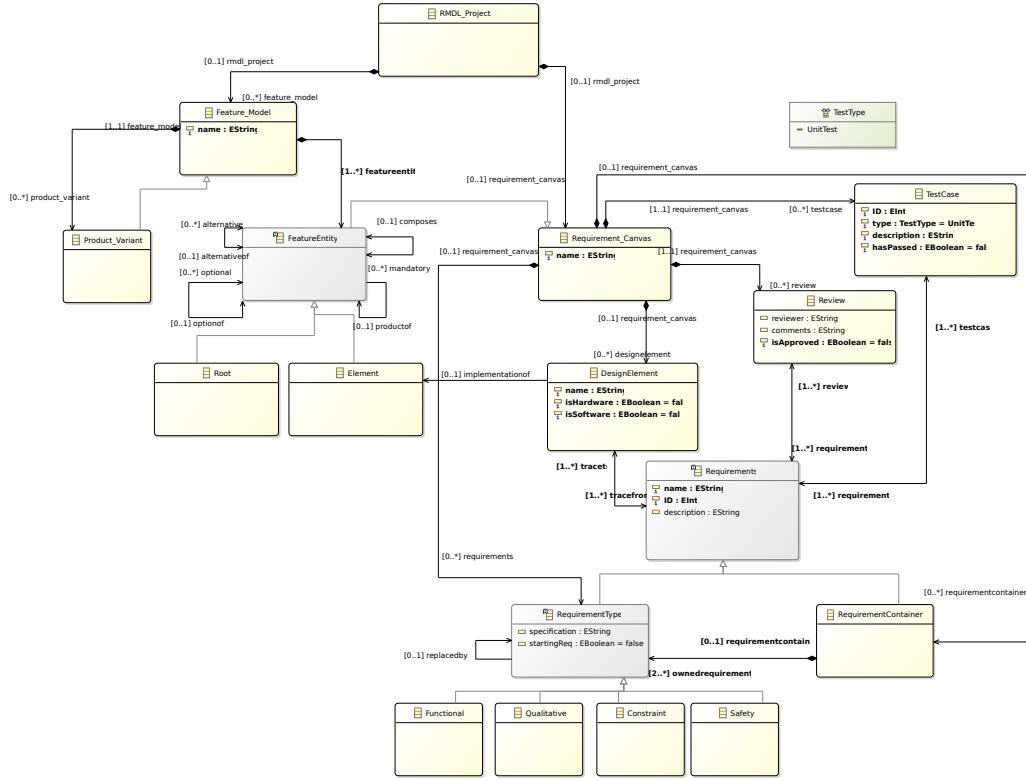


Figure A.1: CyclicL metamodel. Shows the specification for both the requirement canvas and feature modelling portions of CyclicL.

A.2 CyclicL Complete Metamodel

Bibliography

- [1] Steve R Palmer and Mac Felsing. *A practical guide to feature-driven development*. Pearson Education, 2001 (cit. on pp. 1, 27).
- [2] Kyo Kang et al. “Feature-oriented domain analysis (FODA) feasibility study. Software Engineering Institute”. In: *Universitas Carnegie Mellon, Pittsburgh, Pennsylvania* (1990) (cit. on pp. 3, 15, 27, 51).
- [3] Kyo C Kang et al. “FORM: A feature-; oriented reuse method with domain-; specific reference architectures”. In: *Annals of software engineering* 5.1 (1998), pp. 143–168 (cit. on pp. 3, 51).
- [4] A van Lamsweerde. *Requirements engineering: from system goals to UML models to software specifications*. John Wiley & Sons, Ltd, 2009 (cit. on pp. 3, 8, 18, 24, 32).
- [5] Christian Kastner et al. “FeatureIDE: A tool framework for feature-oriented software development”. In: *2009 ieee 31st international conference on software engineering*. IEEE. 2009, pp. 611–614 (cit. on pp. 5, 15, 52, 64).
- [6] Thomas Thüm et al. “FeatureIDE: An extensible framework for feature-oriented software development”. In: *Science of Computer Programming* 79 (2014), pp. 70–85 (cit. on pp. 5, 15, 28, 52, 64).

- [7] Charles W. Krueger. “BigLever software gears and the 3-tiered SPL methodology”. In: *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*. OOPSLA ’07. Montreal, Quebec, Canada: Association for Computing Machinery, 2007, 844–845. ISBN: 9781595938657. URL: <https://doi-org.libaccess.lib.mcmaster.ca/10.1145/1297846.1297918> (cit. on pp. 5, 16, 28, 64).
- [8] Peter Höfner, Ridha Khedri, and Bernhard Möller. “Feature algebra”. In: *International Symposium on Formal Methods*. Springer. 2006, pp. 300–315 (cit. on pp. 5, 16, 52, 61).
- [9] Peter Höfner, Ridha Khedri, and Bernhard Möller. “An algebra of product families”. In: *Software & Systems Modeling* 10 (2011), pp. 161–182 (cit. on pp. 5, 16, 52, 61).
- [10] Government of Canada. *Canadian Motor Vehicle Traffic Collision Statistics: 2022*. <https://tc.canada.ca/en/road-transportation/statistics-data/canadian-motor-vehicle-traffic-collision-statistics-2022>. [Online; accessed 2024-11-27]. 2024 (cit. on p. 7).
- [11] OMG Available Specification. “Omg systems modeling language (omg sysmlTM), v1. 6”. In: *Object Management Group* (2019) (cit. on pp. 8, 19, 39, 45, 62).
- [12] Bertrand Meyer. *Handbook of Requirements and Business Analysis*. Springer, 2022 (cit. on pp. 9, 18, 24, 32, 43).
- [13] Kyo C Kang, Jaejoon Lee, and Patrick Donohoe. “Feature-oriented product line engineering”. In: *IEEE software* 19.4 (2002), pp. 58–65 (cit. on pp. 15, 51).

- [14] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. “Staged configuration using feature models”. In: *Software Product Lines: Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004. Proceedings 3*. Springer. 2004, pp. 266–283 (cit. on p. 16).
- [15] Jose-Miguel Horcas, Mónica Pinto, and Lidia Fuentes. “Software product line engineering: a practical experience”. In: *Proceedings of the 23rd International Systems and Software Product Line Conference- Volume A*. 2019, pp. 164–176 (cit. on p. 16).
- [16] Martin Becker, Rick Rabiser, and Goetz Botterweck. “Not Quite There Yet: Remaining Challenges in Systems and Software Product Line Engineering as Perceived by Industry Practitioners”. In: *Proceedings of the 28th ACM International Systems and Software Product Line Conference*. 2024, pp. 179–190 (cit. on p. 16).
- [17] Damir Bilic et al. “An integrated model-based tool chain for managing variability in complex system design”. In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE. 2019, pp. 288–293 (cit. on p. 17).
- [18] Robbert Jongeling et al. “Co-evolution of simulink models in a model-based product line”. In: *Proceedings of the 23rd ACM/IEEE international conference on model driven engineering languages and systems*. 2020, pp. 263–273 (cit. on p. 17).
- [19] Len Wozniak and Paul Clements. “How automotive engineering is taking product line engineering to the extreme”. In: *Proceedings of the 19th International Conference on Software Product Line*. SPLC ’15. Nashville,

- Tennessee: Association for Computing Machinery, 2015, 327–336. ISBN: 9781450336130. URL: <https://doi.org/10.1145/2791060.2791071> (cit. on p. 17).
- [20] Suzanne Robertson and James Robertson. *Mastering the requirements process: Getting requirements right*. Addison-wesley, 2012 (cit. on p. 19).
 - [21] James Robertson and Suzanne Robertson. “Volere”. In: *Requirements Specification Templates* (2000) (cit. on pp. 19, 47, 69).
 - [22] Ian Sommerville and Pete Sawyer. *Requirements engineering: a good practice guide*. John Wiley & Sons, Inc., 1997 (cit. on p. 19).
 - [23] Ian Sommerville and Pete Sawyer. “Viewpoints: principles, problems and a practical approach to requirements engineering”. In: *Annals of software engineering* 3.1 (1997), pp. 101–130 (cit. on p. 19).
 - [24] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2018 (cit. on pp. 20, 24).
 - [25] Keng Siau and Lihyunn Lee. “Are use case and class diagrams complementary in requirements analysis? An experimental study on use case and class diagrams in UML”. In: *Requirements engineering* 9 (2004), pp. 229–237 (cit. on p. 20).
 - [26] Thomas von der Maßen and Horst Lichter. “Modeling variability by UML use case diagrams”. In: *Proceedings of the International Workshop on Requirements Engineering for product lines*. 2002, pp. 19–25 (cit. on p. 20).
 - [27] Alain Wegmann and Guy Genilloud. “The role of “Roles” in use case diagrams”. In: *International Conference on the Unified Modeling Language*. Springer. 2000, pp. 210–224 (cit. on p. 20).

- [28] Sandy Beidu and Joanne M. Atlee. “Detecting Feature Interactions in FORML Models”. In: *From Software Engineering to Formal Methods and Tools, and Back: Essays Dedicated to Stefania Gnesi on the Occasion of Her 65th Birthday*. Ed. by Maurice H. ter Beek, Alessandro Fantechi, and Laura Semini. Cham: Springer International Publishing, 2019, pp. 220–235. ISBN: 978-3-030-30985-5. URL: https://doi.org/10.1007/978-3-030-30985-5_14 (cit. on p. 20).
- [29] Pourya Shaker, Joanne M. Atlee, and Shige Wang. “A feature-oriented requirements modelling language”. In: *2012 20th IEEE International Requirements Engineering Conference (RE)*. 2012, pp. 151–160 (cit. on p. 20).
- [30] Milena Rota Sena Marques, Eliane Siegert, and Lisane Brisolara. “Integrating UML, MARTE and sysml to improve requirements specification and traceability in the embedded domain”. In: *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*. 2014, pp. 176–181 (cit. on p. 21).
- [31] Jane Cleland-Huang et al. “Best Practices for Automated Traceability”. In: *Computer* 40.6 (2007), pp. 27–35 (cit. on p. 21).
- [32] Ankit Agrawal and Jane Cleland-Huang. “Leveraging Traceability to Integrate Safety Analysis Artifacts into the Software Development Process”. In: *31st IEEE International Requirements Engineering Conference, RE 2023 - Workshops, Hannover, Germany, September 4-5, 2023*. Ed. by Kurt Schneider, Fabiano Dalpiaz, and Jennifer Horkoff. IEEE, 2023, pp. 475–478 (cit. on p. 21).

- [33] Jane Cleland-Huang et al. “Requirements-driven configuration of emergency response missions with small aerial vehicles”. In: *SPLC ’20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*. 2020, 26:1–26:12.
URL: <https://doi.org/10.1145/3382025.3414950> (cit. on p. 21).
- [34] Mehdi Mirakhorli and Jane Cleland-Huang. “Tracing architectural concerns in high assurance systems (NIER track)”. In: *Proceedings of the 33rd International Conference on Software Engineering*. 2011, pp. 908–911 (cit. on p. 21).
- [35] Philipp Heisig et al. “A generic traceability metamodel for enabling unified end-to-end traceability in software product lines”. In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. 2019, pp. 2344–2353 (cit. on p. 21).
- [36] Ryosuke Tsuchiya et al. “Recovering traceability links between requirements and source code in the same series of software products”. In: *Proceedings of the 17th International Software Product Line Conference*. 2013, pp. 121–130 (cit. on p. 21).
- [37] Justin Kelleher. “A reusable traceability framework using patterns”. In: *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*. 2005, pp. 50–55 (cit. on p. 22).
- [38] Jonathan I Maletic, Michael L Collard, and Bonita Simoes. “An XML based approach to support the evolution of model-to-model traceability links”. In: *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*. 2005, pp. 67–72 (cit. on p. 22).

- [39] Hazeline U Asuncion, Arthur U Asuncion, and Richard N Taylor. “Software traceability with topic modeling”. In: *Proceedings of the 32nd ACM/IEEE international conference on Software Engineering- Volume 1*. 2010, pp. 95–104 (cit. on p. 22).
- [40] Government of Canada. *Measured standing height, by age and sex, household population, Canada, 2009 to 2011*. <https://www150.statcan.gc.ca/n1/pub/82-626-x/2013001/t023-eng.htm>. [Online; accessed 2024-11-27]. 2015 (cit. on p. 31).
- [41] Government of Canada. *Measured weight, by age and sex, household population, Canada, 2009 to 2011*. <https://www150.statcan.gc.ca/n1/pub/82-626-x/2013001/t024-eng.htm>. [Online; accessed 2024-11-27]. 2015 (cit. on p. 31).
- [42] Yves Wautelet et al. “Building a rationale diagram for evaluating user story sets”. In: *2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)*. IEEE. 2016, pp. 1–12 (cit. on p. 32).
- [43] Lidia López, Xavier Franch, and Jordi Marco. “Specialization in i* strategic rationale diagrams”. In: *Conceptual Modeling: 31st International Conference ER 2012, Florence, Italy, October 15-18, 2012. Proceedings 31*. Springer. 2012, pp. 267–281 (cit. on p. 32).
- [44] “Gherkin Reference”. In: (2024). URL: <https://cucumber.io/docs/gherkin/reference/> (cit. on pp. 46, 55, 72).
- [45] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. “Sirius: A rapid development of DSM graphical editor”. In: *IEEE 18th Interna-*

- tional Conference on Intelligent Engineering Systems INES 2014.* IEEE. 2014, pp. 233–238 (cit. on p. 58).
- [46] Moritz Eysholdt and Heiko Behrens. “Xtext: implement your language faster than the quick and dirty way”. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. 2010, pp. 307–309 (cit. on pp. 58, 72).
- [47] Mark Richters and Martin Gogolla. “OCL: Syntax, semantics, and tools”. In: *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*. Springer, 2002, pp. 42–68 (cit. on p. 62).
- [48] Jordi Cabot and Martin Gogolla. “Object constraint language (OCL): a definitive guide”. In: *International school on formal methods for the design of computer, communication and software systems*. Springer, 2012, pp. 58–90 (cit. on p. 62).
- [49] Thomas Chiang et al. “Mapping Requirements to Features to Create Traceability in Product Line Models”. In: *Proceedings of the 27th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 2024 (cit. on p. 65).
- [50] Daniel Moody. “The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering”. In: *IEEE Transactions on Software Engineering* 35.6 (2009), pp. 756–779 (cit. on p. 70).
- [51] Kamil Nicieja. *Writing Great Specifications: Using Specification by Example and Gherkin*. Simon and Schuster, 2017 (cit. on p. 72).

- [52] Government of Canada. “GUIDANCE DOCUMENT Guidance on the Risk-based Classification System for NonIn Vitro Diagnostic Devices (non-IVDDs)”. In: (2015). URL: <https://www.canada.ca/en/health-canada/services/drugs-health-products/medical-devices/application-information/guidance-documents/guidance-document-guidance-risk-based-classification-system-non-vitro-diagnostic.html> (cit. on p. 77).
- [53] Government of Canada. “GUIDANCE FOR INDUSTRY Keyword Index to Assist Manufacturers in Verifying the Class of Medical Devices”. In: (2006). URL: <https://www.canada.ca/en/health-canada/services/drugs-health-products/medical-devices/application-information/guidance-documents/guidance-industry-keyword-assist-manufacturers-class-medical-devices.html> (cit. on p. 77).