CyclicL - An Approach to Automated 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

PhD (Jan, 2021)	McMaster University
(Software Engineering)	Hamilton, Ontario

Title:	CyclicL - An Approach to Automated Requirement Traceability in Product Line Engineering
Author:	Thomas Chiang, B.Eng Computer Engineering, M.A.Sc Software Engineering (McMaster University)
Supervisors:	Richard Paige, Alan Wassyng
Number of Pages:	viii, 124

Lay Abstract

The purpose of this paper is to

Abstract

Product Line Engineering (PLE) has become a common engineering practice for managing system complexity across multiple industries. The practice is used for identifying reusable pieces of code, composing software components together, and modelling product variation. There exists a gap however with generating system traceability when using PLE as a technique. Specifically, it is a colossal problem to get traceability from a feature model, the modelling environment for PLE, through requirements to design elements. Further, traceability itself is both a costly and tedious task that is often completed at the end of development and even more difficult to maintain throughout a products life-cycle. With the methodology proposed in this thesis, along with its supporting tool CyclicL, we aim to address the gap that exists between PLE and requirement engineering to push traceability out of a retrospective task to an active portion of development.

Acknowledgments

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

Contents

List of Figures

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

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. But let's change the context to engineering; what came 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 backtrack to 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 features 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? cro:FDDFeature-Driven Development (FDD) is a common practice for agile development styles whereby the engineers

will identify a set of features that will be built together to create a system. 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 behaving very similarly to requirements for the features, if not outright being written as requirements for the feature. Therefore, it's safe to say that before we have requirements, we have features. Features come first and thus we should focus on identifying features before we start to think about what requirements we need for a given system.

There is a slight assumption made in this scenario; are there not already existing requirements before we begin to identify features? For more mature development teams and industries, it is very rare 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 say that requirement come before the features.

In reality, both scenarios are likely to happen, perhaps even within the same company. We may find that we have a vague idea of what the problem is, perhaps even some simple drafts of the problems and what requirements we may have to solve them. However it can very often happen that we know what we want to build before we specify anything of the system as we already have some domain knowledge and can predict what problems we will solve with a given solution. Thus we begin to identify features of our solution and later will go to specify our system what requirements apply to our given features.

This generates a couple of problems for development. The relationship

between requirements and features becomes somewhat ambiguous. In fact, what is a feature? What is a requirement? For this thesis, we will follow the definition of a feature from the original work of Kyo Kuang et al. [1, 2] for their Feature-Oriented Domain Analaysis 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 how we interpret 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 user-interface-able, 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.

What is a requirement? This is well developed and studied topic. To answer, we look to Axel van Lamsweerde [3] 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, iterations between features and requirements do not happen in iso-

lation. 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 is it no longer usable. Which means we create a whole new traceability matrix and thus the cycle continues. Therefore, traceability documentation is a very large problem during iterative and incremental development.

FDD as a development style misses a very key component. How do all the features fit together? This is a problem that is addressed by Product Line Engineering cro:PLEProduct Line Engineering (PLE). Originally conceptualized by Kyo Kuang et al. [1, 2], it has since evolved into a highly researched topic with supporting tools like FeatureIDE [4, 5] which supports feature composition of software artifacts. Or a tool like GEARS [6] 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. [7, 8], 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 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.

What comes first, the feature or the requirements? This is another 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 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.

Finally, the problem of how we accomplish all of these tasks? What is the methodology to help make sure the features we identify are correct and sufficiently complete? What is the methodology 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. This is the final we have identified.

A confounding variable in all of these problems that we have not yet identified PLE is usually a cro:MDEModel-Driven Engineering (MDE) process. As a result, we want to further examine these problems through the lens of MDE; can we solve these problems using MDE techniques and processes?

In summary, these problems culminate into the following research questions:

RQ1: How can we improve requirement elicitation and feature identification

processes in FDD?

RQ2: How can we improve traceability maintenance between features and requirements?

RQ3: How can we improve tool support for iterative and incremental development between features and requirements?

RQ4: How can we leverage MDE techniques for domain analysis, problem space exploration, and requirement specification?

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. One area where these two categories overlap is the automotive domain. Within this 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 to be sold. According to the Canadian government, there were a total of 91533 vehicle collisions reported in 2022, resulting in 1931 total fatalities [9]. It is beyond reasonable doubt that automotive development can be considered a safety-critical industry as well to improve the safety of vehicles to reduce accidents, fatalities, and injuries. Thus, we can see that the automotive industry is one that requirement

PLE to help with managing the product catalog, requires help with managing documentation to develop safety cases (or assurance cases in general), and help with iterative and incremental development as they release new vehicles year after year.

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.

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 RQ1, there are several modelling techniques to use. For the domain analysis and identification of system features, we propose the use of UCD from SysML [10]. As a modelling technique it is quite 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 Axel Van Lamsweerde's requirements engineering textbook [3]. 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 however makes it more difficult to use as a method of reasoning around goals and the possible requirements they can be used to derive. This flexibility does help overall however 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 RQ1. For this we turn to the Handbook of Requirements and Business Analysis by Bertrand Meyer [11]. In his book, Meyer outlines several steps towards the requirement elicitation process, which also apply themselves well 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 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 RQ1, 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. As many engineers and developers are familiar with this type of development we felt it would be easier for them to adapt to this type of relationship as opposed to the other way around. By having features encapsulate requirements, we can get feature-scope requirement traceability as every requirement will be owned by a feature. We will hence forth refer to this as feature-requirement traceability.

This also helps with RQ1 as the encapsulation will help with supporting traceability. By formalizing the relationship between features and requirements, tool development becomes simplified 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.

Chapter 2

Literature Review

2.1 Research Method

There were four main pillars that were used to direct this literature review; MDE, PLE, traceability, and requirement engineering. This thesis had explored different ways to support traceability activities between current PLE techniques and requirement engineering. The MDE aspect of this work comes in the form of both tool support and fundamental descriptions of the methodology. As such when looking for related or previous iterations, we focused on MDE activities that support traceability or/and requirement engineering. The PLE and MDE domains already have a lot of overlap. As such we focused more on tooling and specification for PLE. Thus, the search strings used for this literature review are as follows:

- ("model driven engineering" OR "model based engineering") AND ("product line engineering" OR "feature modelling") AND "tools"
- ("model driven engineering" OR "model based engineering") AND "trace-

ability" 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).

2.2 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. Ties to the MDE is one of the boundaries for the results. There is a combination of formalism and informal approaches to analyzing and solving the problem of traceability between product families and requirements. We also include some fundamental works to support definitions and understandings within the various domains. This is to help support one of our hypothesizes that various definitions across the domains are synonymous.

2.3 Product Line Engineering

PLE has become a well established approach to handling diversity in a companies product portfolio. We scoped the related work to the fundamentals of PLE and some concrete implementations. One of the objectives for this thesis is to add little extensions to the existing theories of product families.

Many tools exist for feature modelling. FeatureIDE [4, 5] by Kästner and Thüm et al. is an extremely well-polished feature modelling plugin tool for

Eclipse that lets a user define their product structure as a feature model, write feature code, and seamlessly integrate said features together to form an executable product. Another tool example for feature modelling and PLE is GEARS [6], which emphasizes its 3-tiered software product line methodology. As a formal method for engineering, feature modelling can also be defined through algebraic methods as shown in the work of Peter Höfner et al. [7, 8] 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. [12]. 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 metamodelling) 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.

2.4 Requirement Engineering and Modelling

Requirement engineering is a critical portion of this research. Part of the contribution of this thesis is the advancement of requirement modelling. As such we look for related works that have come before with efforts to combine

MDE with requirement engineering.

We also have Axel Van Lamsweerde's style of requirements [3]. 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. 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.

We focused on MDE requirement engineering techniques, feature modelling strategies and approaches, and other work related to traceability in product lines. To remain within the scope of requirement engineering, we focused on requirement engineering resources that focus more on general requirement engineering than specific requirement notations. Suzanne and James Robertson [13, 14], 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 [15, 16] 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 requirements diagrams are a helpful modelling approach for requirements engineering, other techniques are also helpful. Use case diagrams from UML [17], 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 [18], but have limited usability beyond communication. They have been used by von der Maßen and Horst Lichter [19] for software product line development, having to customize the metamodel to make it work. Wegmann and Genilloud [20] formalize use cases 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 [21, 22], 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.

The idea of using feature models to encapsulate requirements was partially inspired by the requirement engineering book by Bertrand Meyer [11]. His

style emphasizes a lot of stakeholder analysis for determining their use cases and scoping system requirements around user interactions. Another related requirement engineering approach includes the work from Suzanne and James Robertson [13, 14], which has many helpful definitions for functional and non-functional requirements and their relation to business cases, which are similar to Meyer's use cases. Pete Sawyer and Ian Sommerville [15, 16] have another requirement engineering approach which discusses how system viewpoints affect the scope and definitions of requirements. Finally, Axel Van Lamsweerde's requirement engineering style [3] uses a similar approach to requirement engineering as Meyer's but has a much greater emphasis on the use of formal methods for requirement elicitation and specification. While useful for proving requirement specification, this goes beyond the goal of this paper as we focus primarily on the implementation of CyclicL and the methodology that it supports.

2.5 Traceability

Traceability is a commonly studied topic in academia and industry. For the scope of this thesis, we explore traceability as it is used in PLE and MDE. We further scope related work based on how traceability is used with requirements.

The concept behind this paper shares some similarities to the work done by Marques *et al* [23]. 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.

Dronology, the work of Cleland-Huang and colleagues, is a project that has produced comprehensive work around and including traceability. They cover

automation and tooling for drone development and use traceability artifacts as part of their safety assurance activities [24, 25, 26, 27].

The work of Heisig et al [28] 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. [29]. 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.

Heisig *et al* [28] focused on using a generic traceability metamodel for software product lines. They concluded that using a metamodel facilitated the traceability of various artifacts throughout development. Similarly, the work of Kelleher [30] 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* [31], 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* [32] used machine learning

to generate traceability prospectively and retrospectively; during development and after development.

Chapter 3

Methodology

A main pillar of contribution for this thesis is recognizing overlaps between several very different domains. In the world of MDE, UCDs are often used to capture high-level knowledge for how stakeholders will interact, in this case use, certain parts of a system. While the modelling approach is relatively informal in its semantics and syntax, it does a good enough job of allowing technical and non-technical people alike to describe how they want people to use their products.

In the world of PLE, a critical component comes from the identification of 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" [1]. 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 hardware system it can be the different materials and

components used to build a base model versus and top model trim.

Finally, in FDD the concept of a feature is also heavily used as a means of organization and task setting. Features in this domain are typically 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 equivalency. 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 leverage this equivalency to develop strategies for feature/use case identification, requirement elicitation, and traceability.

Another contribution of this methodology is in the abstraction. FeatureIDE [5] is excellent for reusing software implementations. GEAR [6] is helpful 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.

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. 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.

The overall methodology is heavily inspired by the Goals and Systems books from Meyer's requirement engineering book [11]. While not a complete implementation, it leans heavily on the structures and concepts outlines by

Meyer. The steps of the methodology are as follows:

1. Identify stakeholders, users, and customers.
2. Identify stakeholder, user, and customer goals.
3. Identify stakeholder, user, and customer use cases for a designated system. Each use case should work to satisfy at least one goal.
4. Identify system features based on use cases. These are the high-level features for our system.
5. Refine goals to requirements.
6. Decompose high-level features into feature model.
7. Map features from feature model to goals.
8. Use features to encapsulate requirements.

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 system use cases. They are also used to reason about and refine system requirements. Finally, by equating features and use cases, we use the features to contain our identified requirements to guide development and support traceability efforts.

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 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 a 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. 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. [33] Average range determined between 5th and 95th percentile of male and female population in Canada.
- Assume weight between 48.82kg and 106.60kg. [34] 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. This also ties into the categories we define. The goal of a pedestrian is different than that of a driver. While a stakeholder can be both a driver or a pedestrian, their 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 [11]. The engineer can start informal if needed and can later formalize the model if required to support further analysis.

According to 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 [3]. 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.

For an informal approach, we propose a syntax which loosely follows the syntax from i* Strategic Rational Diagram [35, 36]. The legend is shown in figure 3.1. 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 3.2. 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 place to another. They are also likely to either carry cargo, people, or both

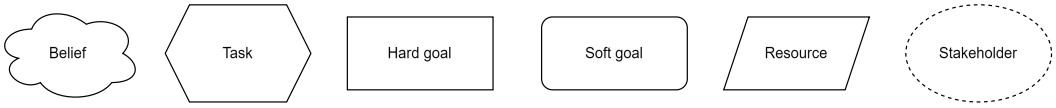

Figure 3.1: Legend of goal diagram elements.

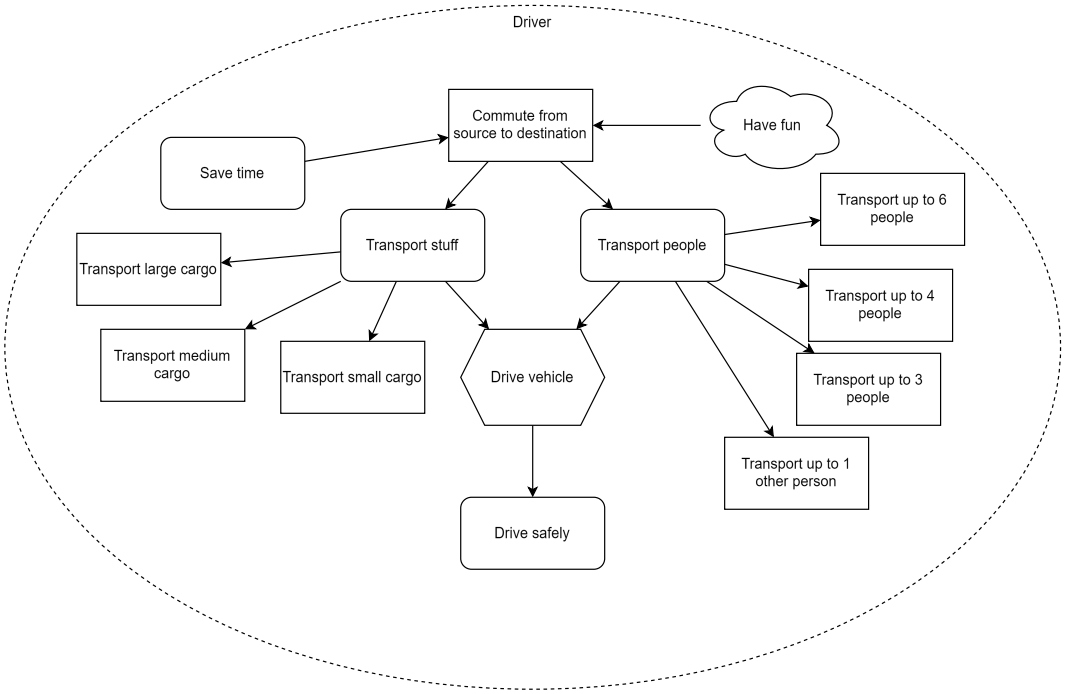

Figure 3.2: Example goal diagram outlining the goals of a driver using a vehicle.

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.

3.2 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 3.3, 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. However 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 hard to detect. As a helpful convention to handle complexity, we consider scoping the system boundary based around the individual goals from the goal diagram. Thus we can re-title the system boundary from figure 3.3 to 'Vehicle - Drive Vehicle'.

This is a simple convention that 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 traceability to the goal diagram for justification. We can create another UCD to specifically target another goal; 'Transport stuff'.

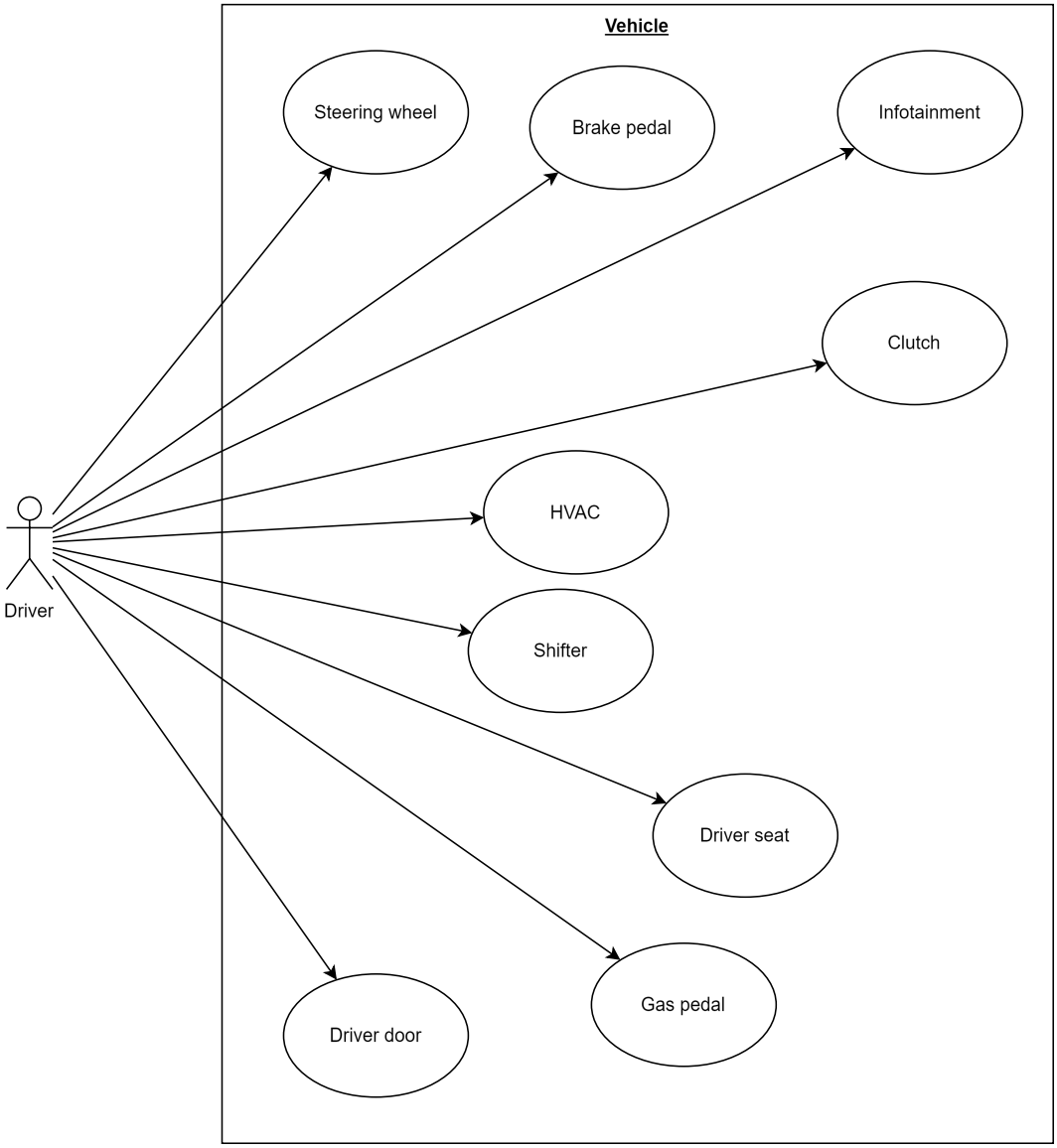

Figure 3.3: A use case diagram outlining what parts of the vehicle a driver will use.

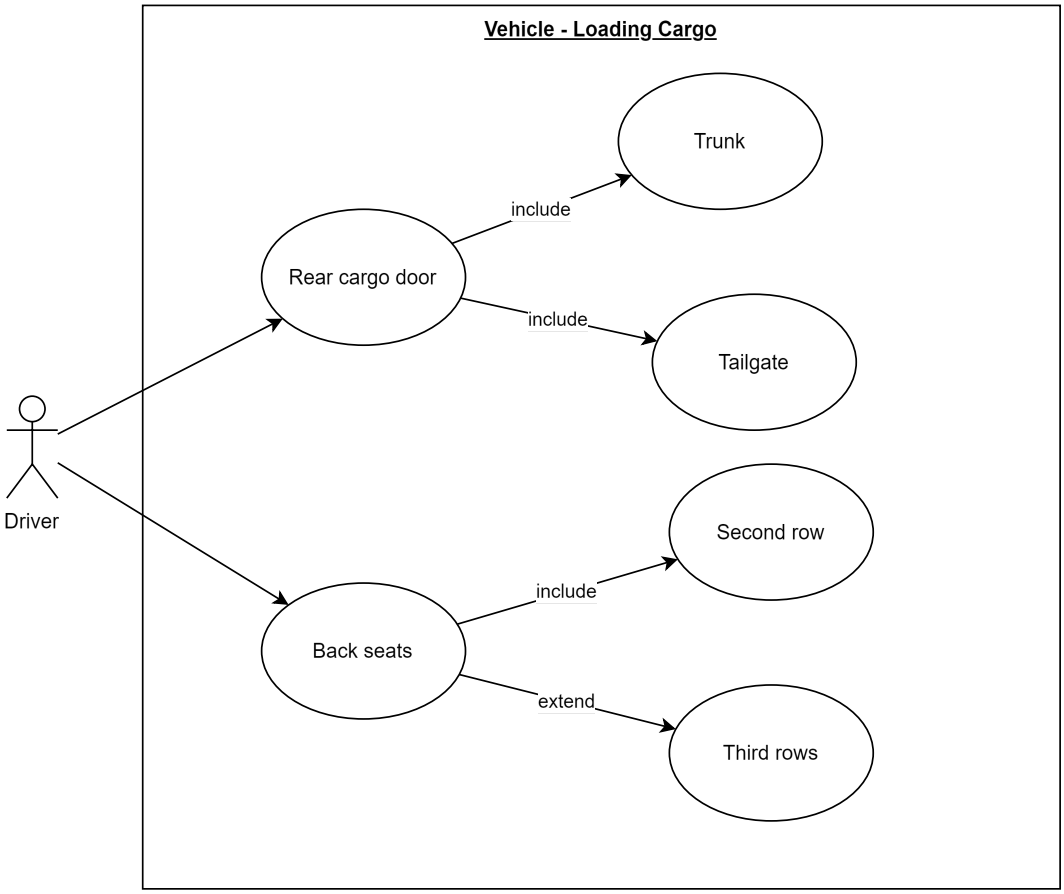

Figure 3.4: UCD scoped by the loading cargo goals

For this convention we recommend maintaining a syntactic match between the goal and the system boundary, however this is not always possible. For the UCD in figure 3.4, 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 equivalently 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.

3.3 Features

With some preliminary work done to identify stakeholders, goals, and use cases, we can begin to identify features of the system. Generally, we want an equivalency 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. This mapping only holds true at the top-level as the UCD is not meant to handle use case decomposition beyond the inclusion and extension relationships as is shown in figure 3.4. We refer to the SysML specification [10] which defines the `include` relationship to point to the use cases that are mandatory for the source use case to be satisfiable. It is also used to point towards use cases that are shared between multiple source use cases. The `extend` relationship is used to point toward optional use cases that

are not mandatory for the source use case to be satisfiable. The distinction between the two can at time be ambiguous. Using figure 3.4 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.

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:

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

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

$$\text{We define } \mathcal{U} \subseteq \mathcal{F} \tag{3.3}$$

We get the definition above based on system complexity. 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 features closest to the root of the feature model, the top-most layer. We can see in figure 3.5 what a possible initial feature model can look like based on the previous UCDs.

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 and compositions 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 3.5 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 3.6. 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:

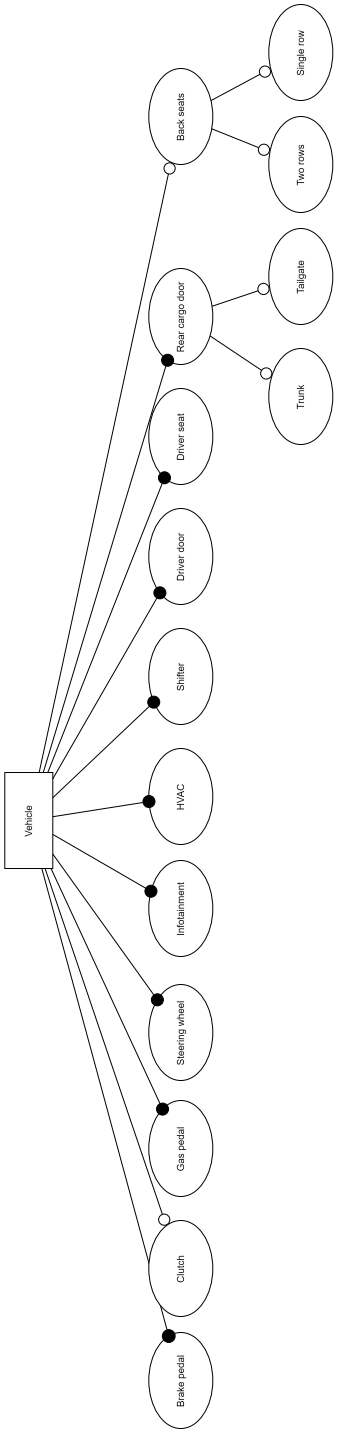

Figure 3.5: Initial attempt at a feature model based on use cases identified in UCD.

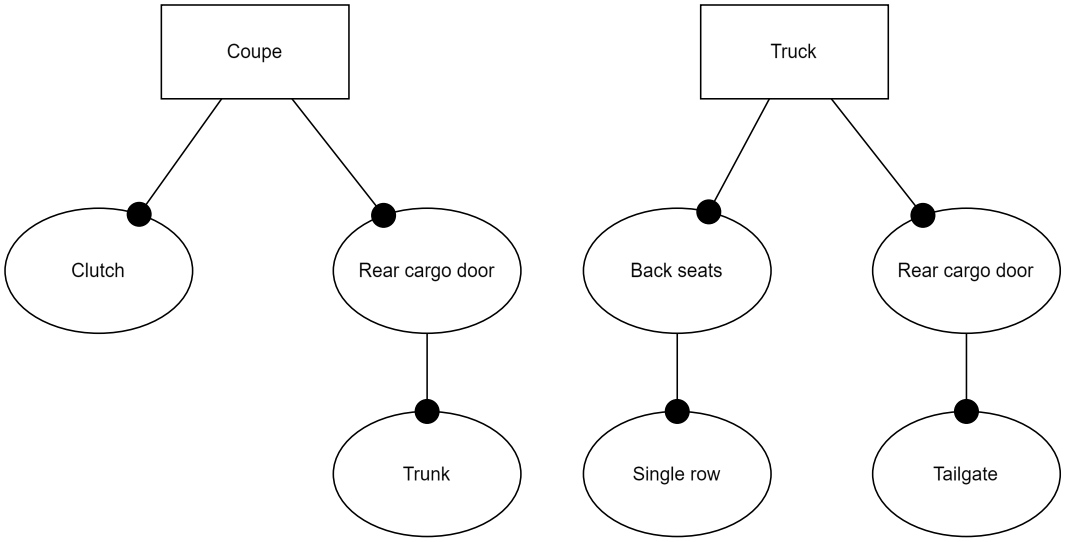

Figure 3.6: Product variants for the initial vehicle feature model. Model is simplified to just the optional features that are mandatory for the variant.

$$\text{vehicle.backseats} == \text{false} \rightarrow \text{vehicle.clutch} == \text{true} \tag{3.4}$$

$$\text{vehicle.rearcargodoor.tailgate} == \text{true} \rightarrow \text{vehicle.clutch} == \text{false} \tag{3.5}$$

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.

3.4 Goal Refinement

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 stuff for stakeholder. We ask the next question; how will the vehicle enable the transportation of stuff? If there is a goal decomposition, we can also correlate the answer to the question of how to a decomposed goals. How will the vehicle enable the transportation of stuff; 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 arbitrarily or by data research. For this example we go with an arbitrary decision. Large cargo space shall be in the range of 140-150 cubic feet, medium cargo space shall be in the range of 120-139 cubic feet, small cargo space shall be less than 119 cubic feet.

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 in chapter S4) from his system book; user stories [11]. 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 elicit 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.

3.5 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 [10]. The requirement canvas diverges from its SysML counterpart in its focus on traceability and implementation details. The requirement canvas has several main objectives:

- Provide an environment for modelling requirements in a way that emphasizes traceability.
- Provide an environment specifying requirement in more detail after elicitation.

- Provide opportunities for automated maintenance and traceability reports. (This include change impact analysis)

With these in mind a metamodel was created to capture these objectives and provide a specification for how to model our requirements found in figure 3.7. This metamodel has also been used to develop a domain-specific modelling language named CyclicL. With CyclicL we aimed to show satisfiability of the proposed methodology in parallel to the continued development of the methodology. This includes the main 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

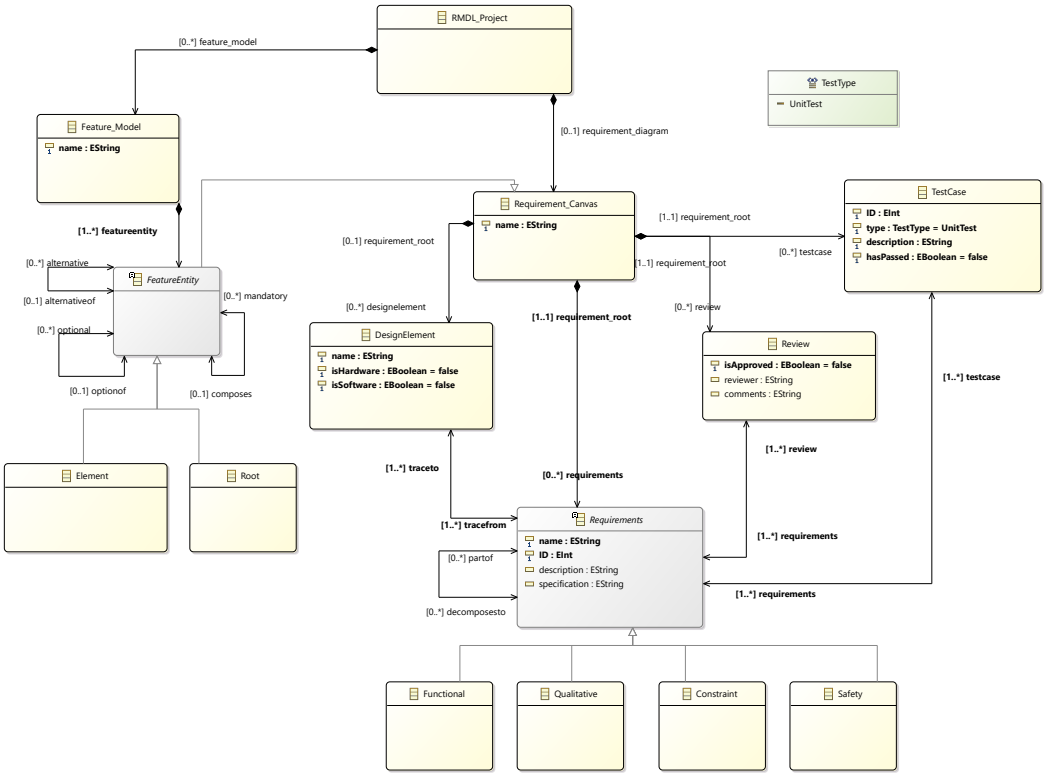

Figure 3.7: CyclicL metamodel. Shows the specification for both the requirment canvas and feature modelling portions of CyclicL.

Suzanne Robertson Volere requirements [14]. 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. It is important to note that requirement categorization relies heavily upon system, or in CyclicL, feature scoping. For 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 capabilities of the engineer creating the models.

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 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. 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.

For the feature modelling portion of the metamodel, the specification is based on the work of Peter Höfner et al. [7, 8] which uses set theory as the basis for proving how their feature modelling algebra works. It is also loosely inspired by FeatureIDE [4, 5] an extremely well-polished 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. This is a key step in the process as this allows for what we dub feature-requirement traceability. As this proposed method-

ology 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.

3.6 Feature-Requirement Traceability

The feature-requirement traceability is the main purpose for this methodology. 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 model elements requirement for the requirement canvas thus encapsulating the various types of requirements, their associated approvals, and associated design elements.

This relationship allows for several key benefits. First, it allows 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 is more reflective of the relationship that features and requirement have in FDD environments allowing more compliance between tolls that support FDD. Thirdly, this allows us to model product variance at the requirement level. Allowing for specification of product variance is helpful for reducing unexpected changes

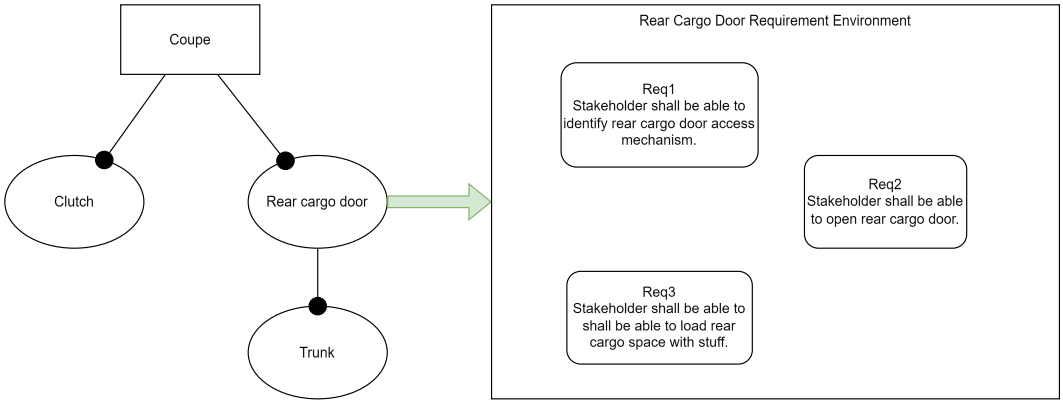

Figure 3.8: Example of how a feature can be opened up to display encapsulated requirements.

and capturing knowledge of anticipated variance in product early in the product lifecycle development. Further, this will support change impact analysis across product variants to support 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 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, cross-cutting requirements across features is not supported at this time. There might be ways that users circumvent this which can lead to duplicated or inconsistent requirement across several features.

In figure 3.8 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.

3.7 Requirement Specification

Between the goal diagrams and use case diagrams we showed how we can do requirement refinement. In the requirement canvas we intend to express these requirements as well. This is beyond just refinement however, as part of the requirement canvas we also enable requirement specification. Currently we support behavioral specification through a loose implementation of Gherkin syntax [37]. Gherkin style of specification is a structured natural language approach to specifying behaviors. It follows a pseudo predicate logic system using three main keywords; GIVEN, WHEN, and THEN. The GIVEN keyword is used to express behavioral preconditions. The WHEN keyword is used to express events that trigger some new behavior. These triggers can be either internal or external to the specified system. Finally, the THEN keyword is meant to specify behavioral 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 to specification, it is enough for a proof of concept.

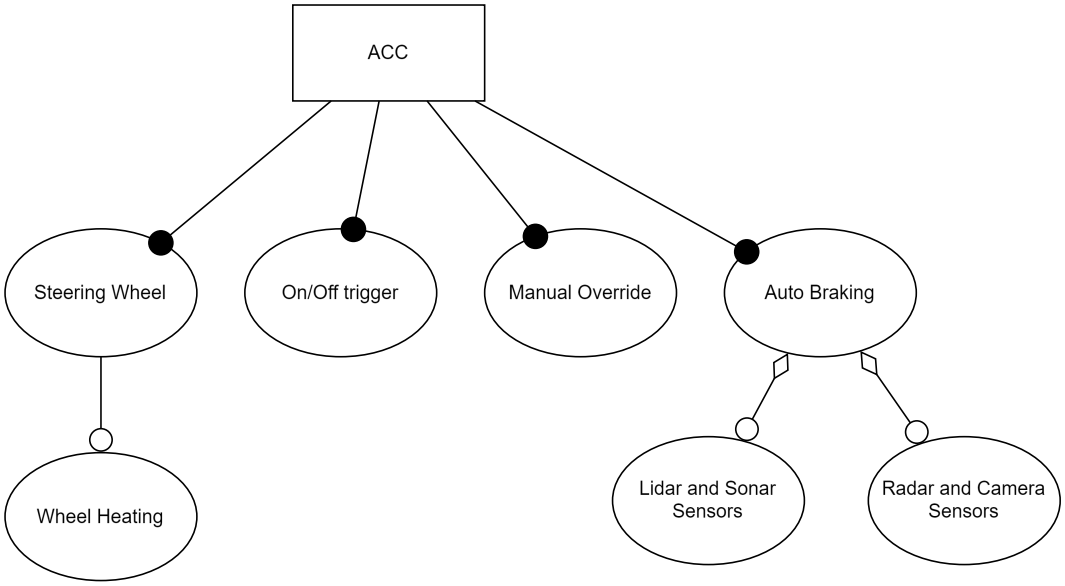

Figure 3.9: Simplified example of an adaptive cruise control feature model.

3.8 Requirement Based Feature Identification

Due to the relationship between features and requirements outlined in this methodology there are alternative ways to identify features. The variation points that exist in this approach are the requirements held in the 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 3.9, 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.

Chapter 4

Implementation

As a proof of concept, we have implemented a DSL to show the feasibility of automated traceability between features and requirements. Supported by the entire methodology, the tool CyclicL is a model-based tool to support to creation of feature models, requirement models, and leveraging the relationship between them to support automated traceability generation and maintenance throughout a products life-cycle.

CyclicL is also where we have been exploring potential composition strategies. We expected that traditional composition techniques from PLE may require some modifications as a result of the relationship we have introduced between features and requirements.

4.1 Implementation Objectives

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.

Based on our objectives, 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 define their product in terms of features. The second modelling environment is the requirement canvas environment. This environment enables the user to specify their requirements, associate requirements to design elements, and assign reviewers and test cases to requirements. This is also where the majority of the traceability in CyclicL is defined. These two modelling environments are connected through the features in the feature modelling portion. We use features to encapsulate the requirements. Each feature owns its requirements, allowing for a strong hierarchy between features and requirements. By connecting the two modelling environments we end up with a couple of different forms of traceability. Using the requirement canvases alone we can get a simple requirement traceability matrix. Using just the feature modelling environment we can generate feature traceability. And when we combine both environments we get a feature-requirement traceability matrix that shows what features own what requirements, and how they trace and relate to other requirements and features in the system.

For the feature modelling portion of the tool, our specification is based on

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.

the same algebraic specification from Peter Höfner *et al* [7, 8], 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 need to be handled in the implementation. This definition does allow us to scope and understand what we want a feature to do; namely encapsulate requirements.

Our main reason for using requirement diagrams as a starting point to capture requirements is the semi-formal specification in SysML for requirement diagrams. We found enough semantics in the specification to enable

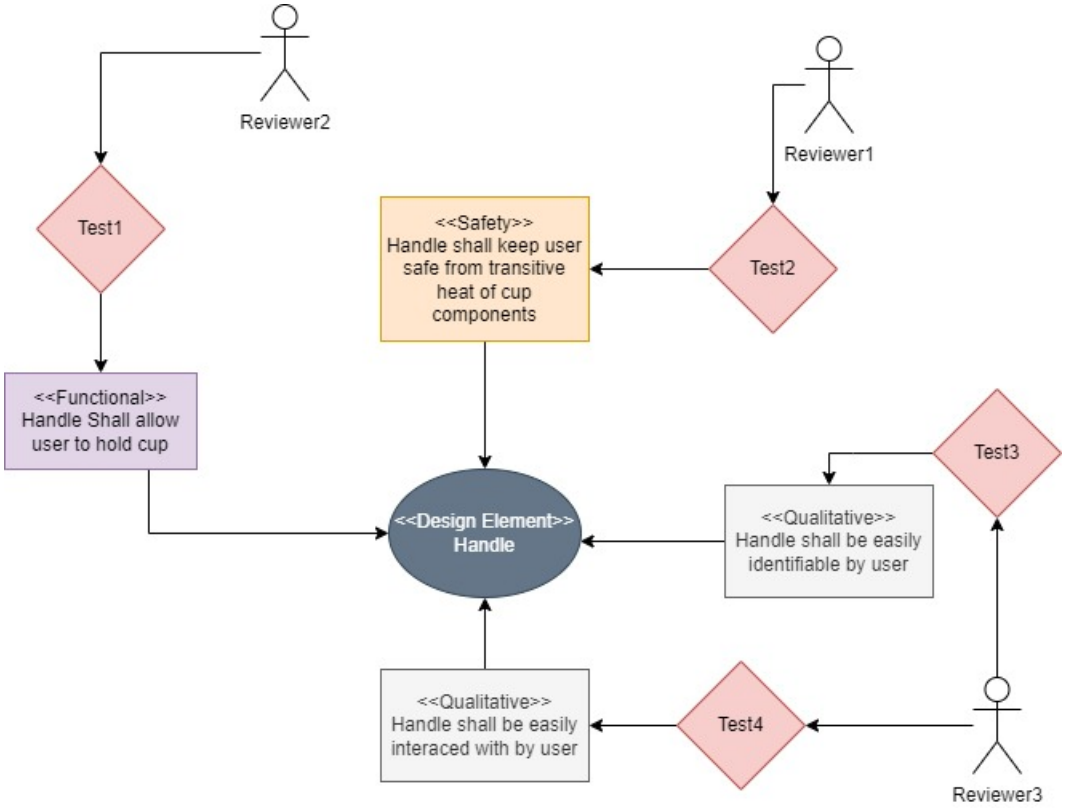

Figure 4.2: Example of a requirement canvas concrete syntax. The example uses a disposable coffee cup with a handle as the focus of the diagram.

traceability automation using the connections between elements. Furthermore, requirement diagrams are a generally under-explored means of requirement engineering based on our literature review, especially when compared to feature modelling and product line engineering. This led to us deciding to iterate on requirement diagrams as a means of exploration for traceability, supporting our previous decision to reuse as much as possible from previous PLE work. In this exploration, we diverged significantly from the original specification for requirement diagrams. In order to capture this divergence, we henceforth refer to the requirement modelling environment as the requirement canvas. We will explore more of the differences in sections 4.2 and 4.3.

An existing tool for PLE and feature modelling such as FeatureIDE [4,

5] allows for product generation by weaving software defined in the features together. GEARS [6] has a three-tiered approach to PLE that can help with software management. However, neither of those tools has an explicit focus on traceability between features and requirements. The requirements are housed separately from the feature model and necessitate user intervention to pull requirements from another source, if at all since the tools can function fine without requirements. However, in CyclicL the requirements are necessary to build out the full traceability that the tool is focused on generating and maintaining, functionality that is lacking or not the primary focus in other PLE tools. Thus, we position CyclicL as a development tool that automates traceability generation and maintenance between PLE and requirement engineering by purpose-building it for feature model and requirement canvas creation. By housing everything in one location it prevents a situation where the feature model and requirements can be out of sync with each other. We propose that having both within a single tool facilitates auxiliary development activities such as change impact analysis and automated maintenance of traceability documentation.

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. As the scope of the tool 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 have their supporting documentation when using CyclicL. This is a limitation

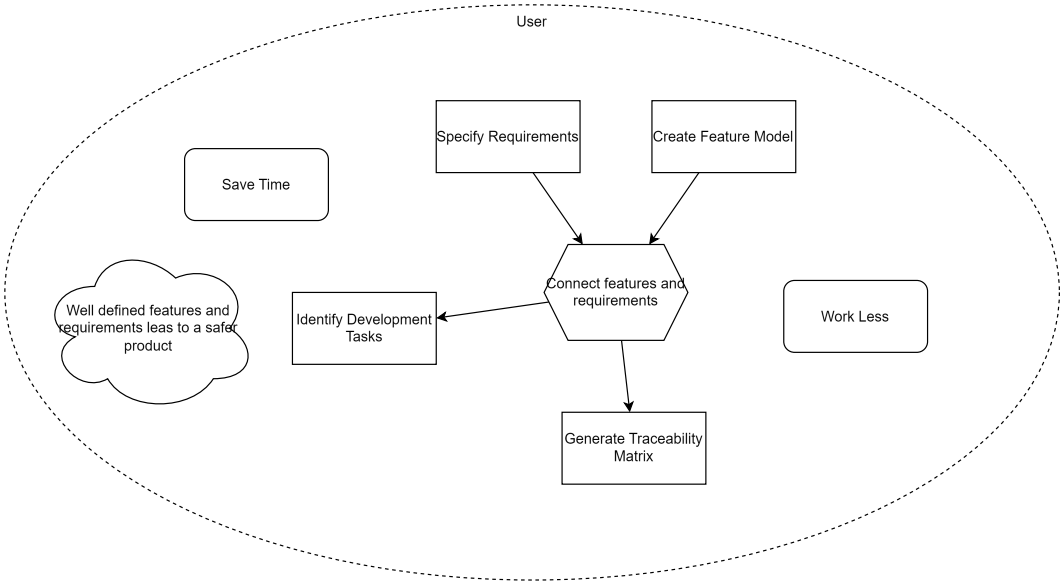

Figure 4.3: Anticipated goals of users for CyclicL.

of the tool for now as we do not currently support traceability from the domain analysis to the identified features and requirements. We also have no way to encourage or enforce a domain analysis purely through the use of CyclicL. Thus, we cannot confirm validity of models created in CyclicL at this time based on justifications from supporting documentation.

4.1.2 Requirements

We extracted high-level requirements from the previous work in building a product line and requirements for a coffee cup to guide initial development plans for CyclicL [38]. We also included the goal of supporting iterative and incremental development life-cycles. We define an iterative change as a change based on feedback from later stages of development or stakeholders. These are usually larger changes relative to an incremental change. 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. Examples of an incremental change can be a refactoring change, updating a definition, or changing a connection between components. 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.

Under these specifications, we allow for requirement encapsulation. We treat the requirements encapsulated by a feature as attributes and are considered feature requirements, thus scoped by the feature. For the structure to define the requirements within a feature, we decided to use requirement diagrams from the SysML specification [10] as inspiration for creating 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 3.7 is what was used 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. Thus, every feature in the feature model can become a requirement canvas, allowing us to capture the requirements for each feature as attributes of said feature. 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. The requirement canvas also has a containment relationship with the model root, `RMDL_Project`. This is carried over from earlier development and was left in the tool, for now, to allow

users to define a requirement canvas for the entire system they are specifying without being tied to specific features. The justification for this structure is to provide a kind of doodle space for requirement specification that can later be refined by copying and pasting some model elements into a requirement canvas encapsulated by a feature. However, 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 [14]. 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. It is important to note that requirement categorization relies heavily upon system, or in CyclicL, feature scoping. For 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 differ-

ent from the functional requirements. This distinction relies heavily upon the capabilities of the engineer creating the models.

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 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. 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.

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 [39]. 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.

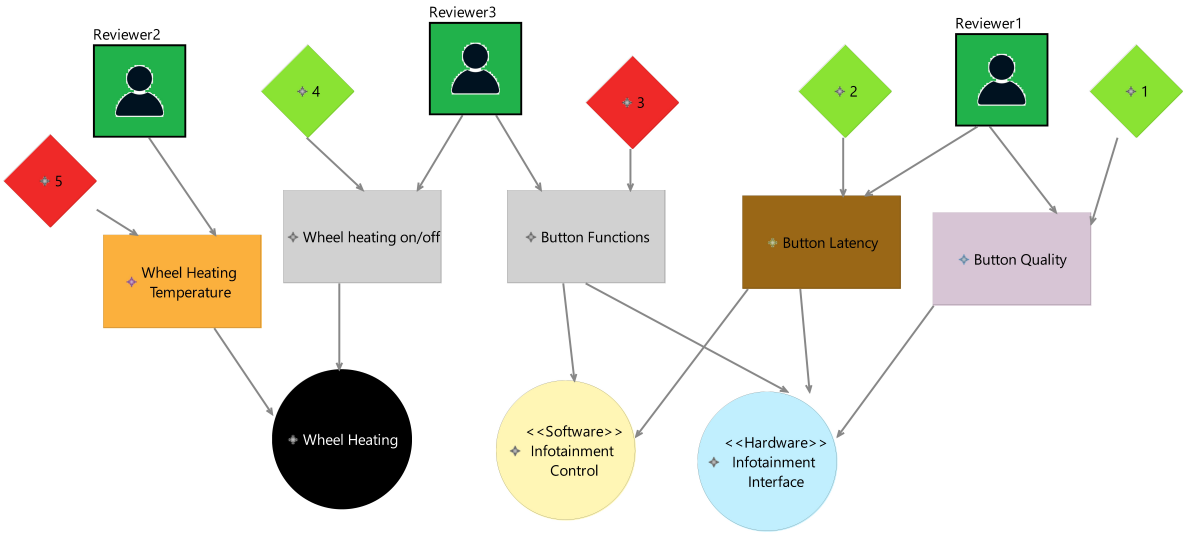

Figure 4.4: An example of the concrete syntax for the requirement canvas using requirements from the automotive domain.

For the feature modelling portion of CyclicL, we maintained as close to traditional syntax as possible. This is due to the objective of maintaining the original specification for feature modelling to our best ability during implementation. 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 the black diamond at the source and

an arrow at the target. The optional relationship uses no decorator at the source and a triangle at the target. This is to represent OR relationships. The alternative reference uses a white diamond at the source and a triangle at the target. This is 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. The optional and alternative references share the diamond at the target to show that the target is optional. In contrast, the white diamond at the source differentiates the two types of references. Thus while they have some common semantics in the options, the alternative has more syntax to represent its extended semantics.

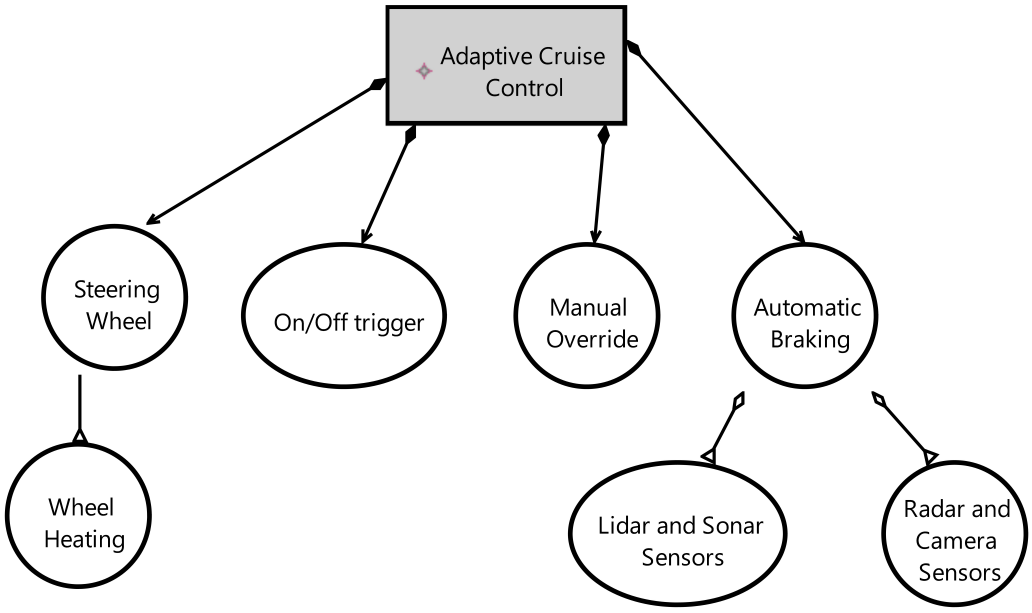

Figure 4.5: An example of the concrete syntax for the feature model using features for an adaptive cruise control system from the automotive domain.

Along with the graphical portion of CyclicL, we also implemented an Xtext-based [40] specification language. The implemented specification language is based on the Gherkin specification language [41, 37]. We chose this as

```
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.

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 of adding predicate structure to natural language makes it relatively easy to specify requirements and to implement the language within CyclicL.

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 compositional 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 the convention that a requirements variant necessitates a new feature, 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 for a proof of concept this is currently feasible in CyclicL.

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

medical device development and automotive manufacturing. Specifically, we are looking at a requirement set from Boston Scientific for a pacemaker **??**. This requirement set is often used as part of the undergraduate curriculum at McMaster University for teaching product life-cycle development and safety-critical development.

From the perspective of safety criticality, medical devices and a pacemaker especially, tend to have a greater impact on individual health and safety. 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 focus on medical device guidelines and classifications based on Health Canada legislation [42].

5.1 Knowledge Capture & Domain Analysis

Method works for automotive requirements. Will show using pacemaker. Do not show generality. Just show that it works in two domains. Would be interesting to explore generality.

- Want to show benefit of methodology for capturing knowledge prior to requirement elicitation process. Benefit around requirement justification, communication, and stakeholder confirmation (last two are difficult to show, but easy to claim). Future work for proving this claim. Maybe make assurance case for this claim?

- Want to show uncertainty and incompleteness management/control/mitigation of requirements using the scenario breakdowns of the requirement refinements. Might want to argue more about potential to better manage

incompleteness and uncertainty. Can provide clarity on requirements, requirement refinement, and requirement decomposition. Evaluates characteristic of methodology rather than empirical property.

- Want to show cost savings of modelling requirements compared to in other tools when it comes to requirement traceability
- Want to show benefits of using product family to contain requirements.
- Want to show benefits of tying development strategies of FDD with product family.
- Want to show how scenario breakdown for refinement helps with specification using Gherkin.

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 do a good job of introducing the scope of the requirements, stakeholder identification, and system definitions. Enough 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 have found however that we can improve upon this domain analysis with the usage of goal diagrams and use case diagrams introduced by our proposed methodology.

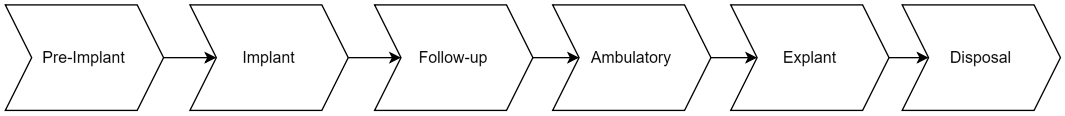

Figure 5.1: Pacemaker lifecycle as outlined by Boston Scientific requirements.

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. The document does not however explain what each stage actually is or what the stakeholder roles, use cases, or goals might be in each stage. While perhaps not directly relevant to the requirements of the pacemaker, the lack of knowledge capture provides uncertainty. As such, the engineer doe not know what information might be missing and therefore would not know if the missing information is relevant to the requirements. As 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.

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 extra definitions are both informed by some of the stakeholder goals, and help expose some other goals that are not as obvious. We identify as stakeholders of the pacemaker system. Compared to the original pacemaker document we have identified more stakeholders of the pacemaker system.

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

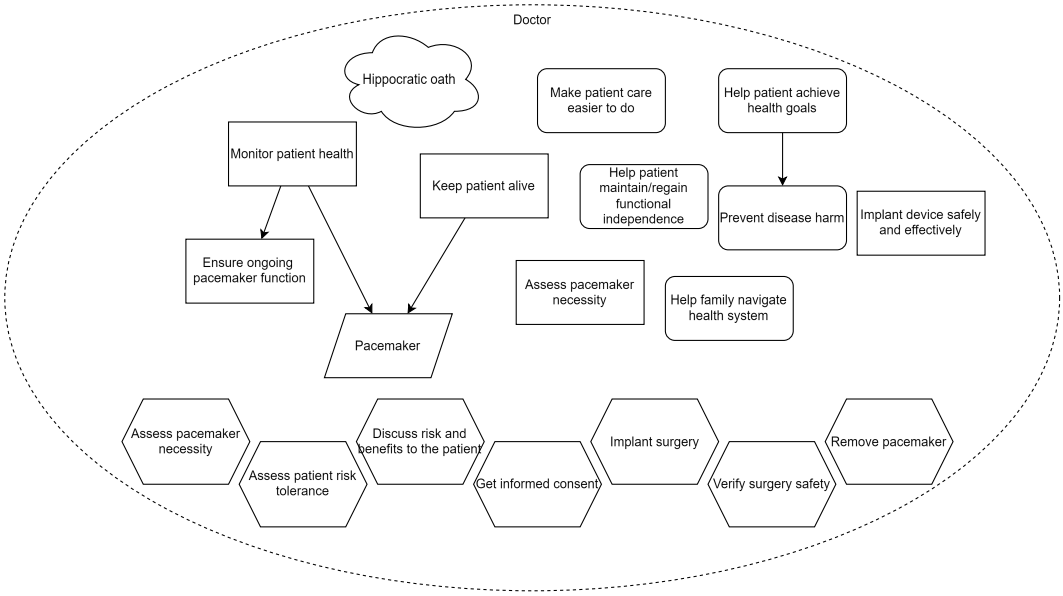

Figure 5.2: Doctor goals in the context of the pacemaker system.

- Nurse (from Pacemaker requirements)
- 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

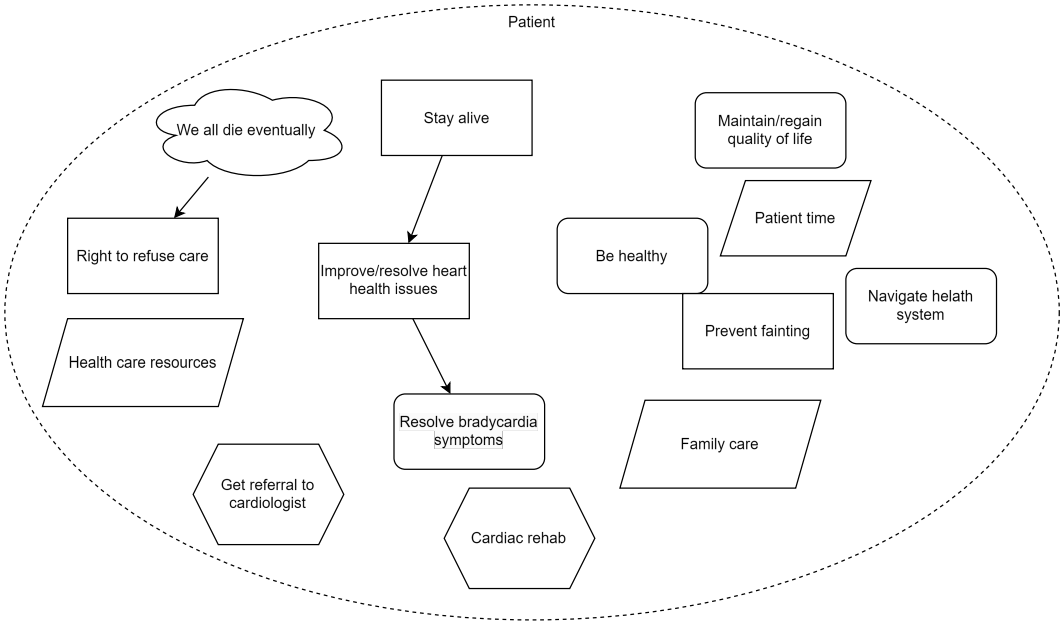

Figure 5.3: Patient goals in the context of the pacemaker system.

around the pacemaker it presents itself twice; a patient can refuse to implant 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 also identify what the goals are for the nurse and technician stakeholder. 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 clarify what the goals of the hospital could be as well, though the hospital is mostly where

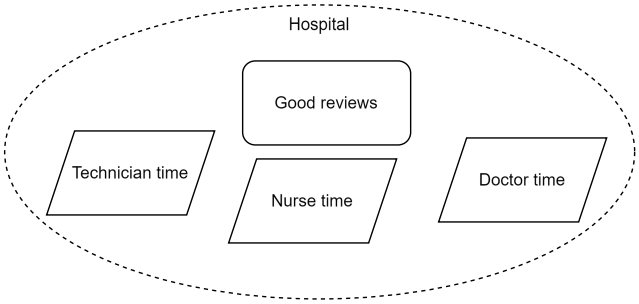

Figure 5.4: Hospital goals in the context of the pacemaker system.

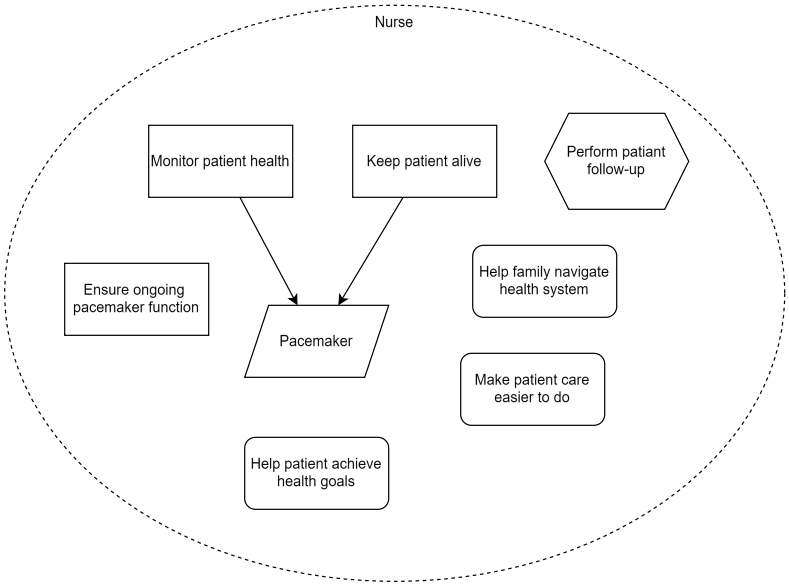

Figure 5.5: Nurse goals in the context of the pacemaker system.

the resources are found. There can be more goals for the hospital in general, but the goals are limited in the scope of a patient with bradycardia and the pacemaker lifecycle.

Along with the hospital, we found 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 have a goal to help care for the patient in ways that the medical team is unable to; help in the day-to-day activities of the patient and in the

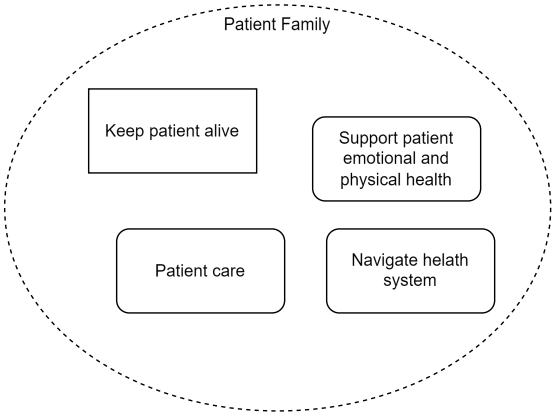

Figure 5.6: Patient family goals in the context of the pacemaker system.

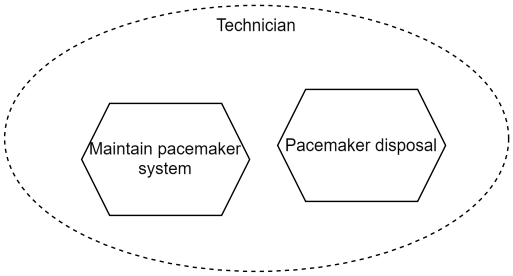

Figure 5.7: Technician goals in the context of the pacemaker system.

community.

There are many goals 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 have in regards to the pacemaker product lifecycle. We also claim that we

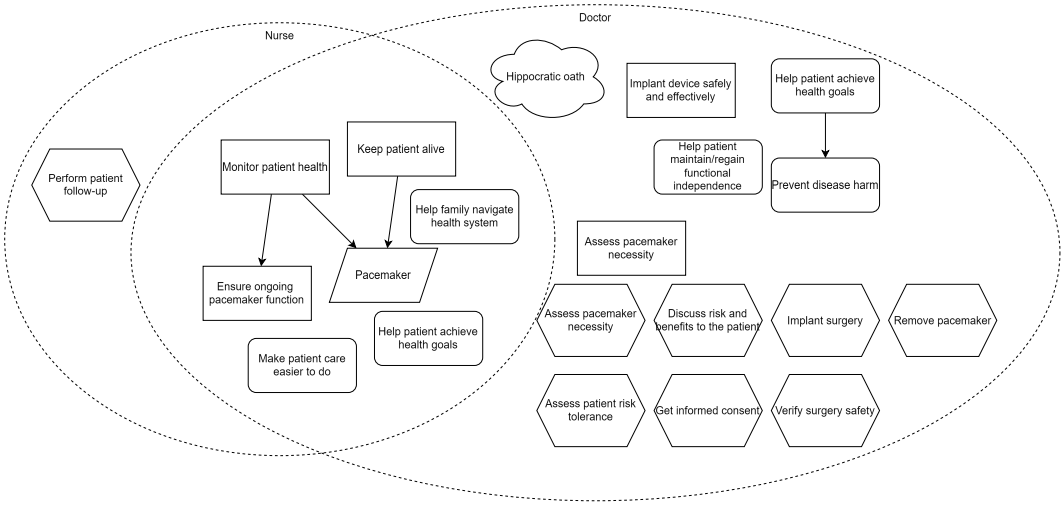

Figure 5.8: Combined nurse and doctor goals.

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 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 document it defines the various components that compose the pacemaker system; the pacemaker device (AKA 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. 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 PD 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 bia 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. 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 electrogardiogram 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 data 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. Further, two of the use cases are broken down into smaller, mandatory use cases to define the full scope of the usage. This is modelled using the various `include` relationships to capture The mandatory use cases. We also capture the reuse of the Rate-adaptive use case for both single and dual-chamber pacing.

In figure 5.10 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 however. 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 task of patient follow-up that requires they be able to access the historic and diagnostic data from the patient.

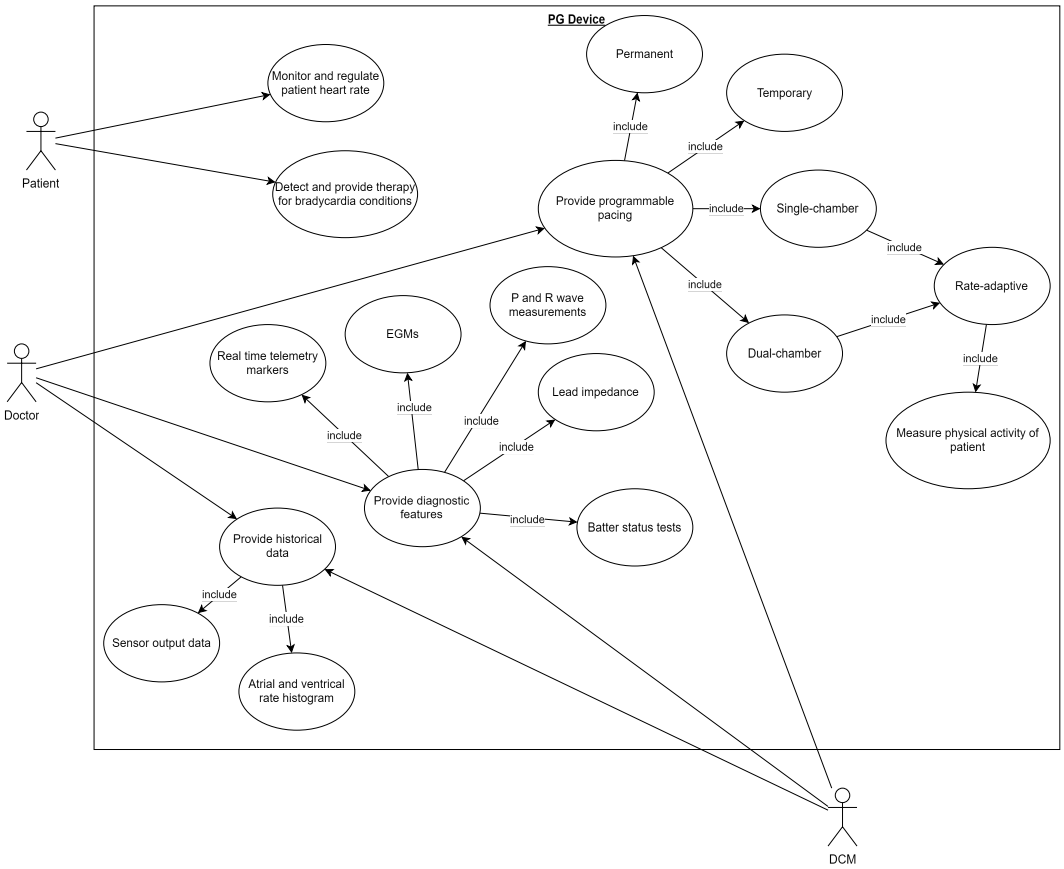

Figure 5.9: Use Case Diagram for the PG device based on the original pacemaker specification.

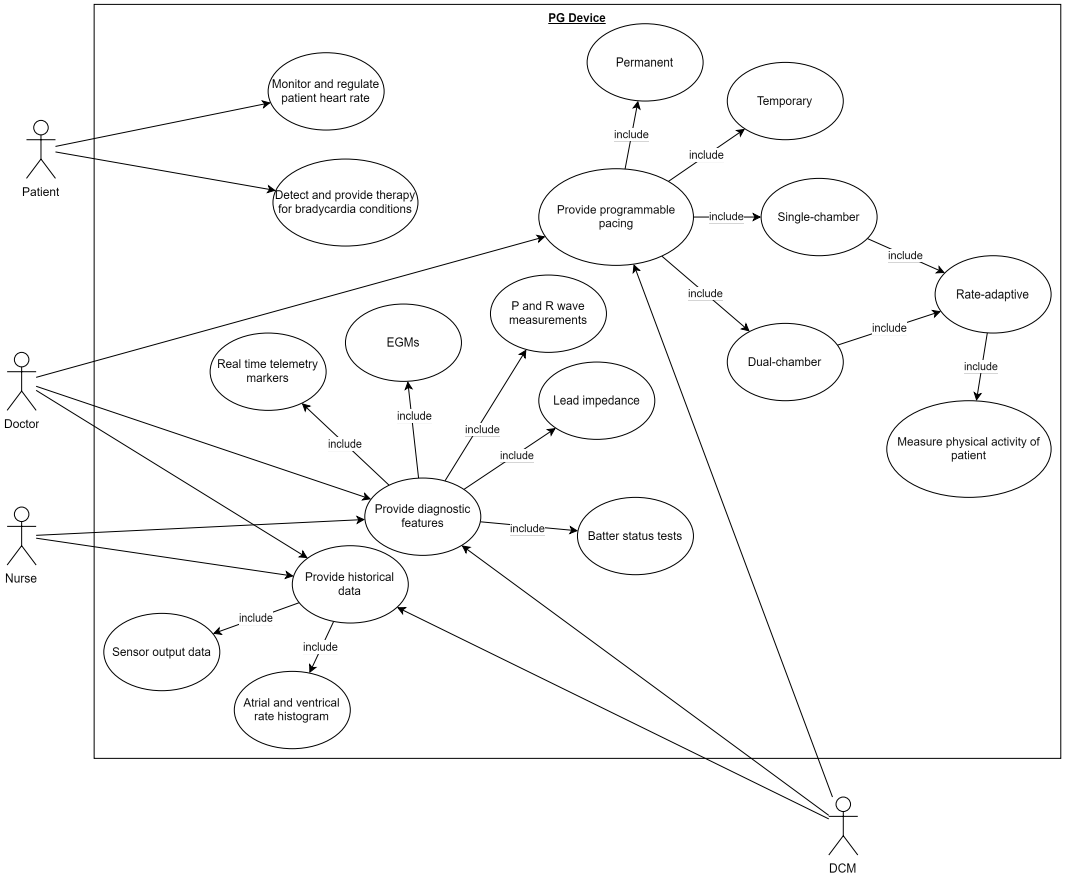

Figure 5.10: Use Case Diagram for the PG device based on the original pacemaker specification enhanced with information from the goal diagram knowledge capture.

5.1.3 Coherence in Medical Device Development

5.1.4 Relevance in Medical Device Development

5.1.5 Impact in Medical Device Development

5.1.6 Efficiency in Medical Device Development

5.1.7 Effectiveness in Medical Device 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 another challenge of traceability and justification. It requires the engineer to be aware of the links between the domain and problem space analysis and the feature and requirement models in the tool. This implicit knowledge will eventually need to be documented somewhere to support both future development changes and assurance for safety-critical development. As a result, one potential path is creating a bridge between the domain analysis modelling and the current tool to extend the traceability all the way up to the concept phase of development. While we expect UCD and goal diagrams to remain mostly static once they are created, the direct traceability could be helpful documentation for onboarding new employees to help them get familiar with why certain decisions were made, support change impact analysis justifications, and aid the development of assurance case by having complete end-to-end traceability between the problem

space and the solution space.

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 syntax constraints in the tool to ensure that all user created models are valid according to the metamodel. 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. Most importantly, 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 behavioral approach and thus may require us to re-evaluate what type of specification language we implement within the requirement canvases of CyclicL.

Chapter 7

Conclusion

What comes first, the requirement or the feature? In this thesis we have attempted to outline an answer to this question. Neither, it is the customer use cases and the user goals that come first. Those allow us to identify features and elicit requirements in parallel. 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 can identify who our customers/users/stakeholders are, what we expect them to do with our system, and why they would want to use our system to satisfy their goals. We have shown to we can map the identified use cases to features of our system. We have shown how we can derive high-level requirement from our goal diagram and how we can refine and decompose those requirements. Thus we have shown satisfaction of RQ1.

Thanks to this process contribution, we are also able to identify features and elicit requirements in parallel. The next step we need to do is decide which features own which requirements. This is first done at a high-level as we map which features satisfy a user goal or goals. Then we can refine

those goals into requirements that are owned by the feature, all scoped by the feature to ensure relevance. We have shown an added bonus to this approach using the feature decomposition in the feature model. This implies a natural decomposition in the requirements as well as they are scoped by the features that own them. Overall, there is a much clearer path to specifying features in a FDD environment with the supporting methodology and tool. This satisfies RQ1.

One of the main highlights of our methodology is the feature-requirement encapsulation. By defining this hierarchy, we have shown how it facilitates traceability between features and requirements. We can show what features own the requirements, dependency between features, and by extension, dependencies external 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, satisfying RQ1.

Finally, CyclicL has shown the potential of a PLE tool that is self contained with requirements as part of the tool. We have shown that we can support incremental and iterative 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 and architectural changes without dependencies on external support. While there are still some limitations in the current tool capabilities, we have demonstrated how

future development can continue to address these short-comings. Therefore, we believe that we have satisfied RQ1 and the potential of a tool that supports iterative and incremental development of traceability in parallel to feature and requirement development.

Appendices

Appendix A

A.1 Boston Scientific Pacemaker Requirements

PACEMAKER System Specification

January 3, 2007

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

Contents

List of Tables

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 tachyrhythmias.

Use of the PACEMAKER pulse generator in DDDR and VDDR modes may be contraindicated (1) in patients with chronic atrial tachyrhythmias (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.
- Electrocautery 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.

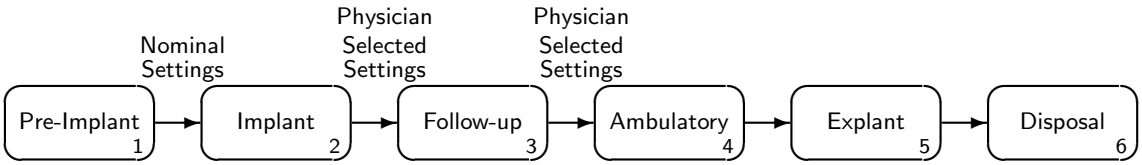

Figure 1: PACEMAKER Life Cycle

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	DDDR 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.

	I	II	III	IV (optional)
Category	**Chambers Paced**	**Chambers Sensed**	**Response To Sensing**	**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	AAT	VVT	AOO	AAI	VOO	VVI	VDD	DOO	DDI	DDD	AOOR	AAIR	VOOR	VVIR	VDDR	DOOR	DDIR	DDDR
Lower Rate Limit	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
Maximum Sensor Rate											X	X	X	X	X	X	X	X
Fixed AV Delay							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
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
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
ATR Fallback Mode							X			X					X			X
ATR Fallback Time							X			X					X			X
Activity Threshold											X	X	X	X	X	X	X	X
Reaction Time											X	X	X	X	X	X	X	X
Response Factor											X	X	X	X	X	X	X	X
Recovery Time											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
ATR Fallback 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	±5% referred to the connector
Lead Impedance	50Ω	100 to 500Ω, ±greater of 100Ω or 30% 500 to 2000Ω, ±25% 2000 to 2500Ω, ±30%
Battery Voltage	0.01 V	±2%

Table 8: Measured Parameters

Bibliography

[1] 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, 4, 19).

[2] 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, 4).

[3] A van Lamsweerde. *Requirements engineering: from system goals to UML models to software specifications*. John Wiley & Sons, Ltd, 2009 (cit. on pp. 3, 7, 14, 16, 24).

[4] 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. 4, 12, 40, 49).

[5] 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. 4, 12, 20, 40, 50).

[6] 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. 4, 13, 20, 50).

[7] Peter Höfner, Ridha Khedri, and Bernhard Möller. "Feature algebra". In: *International Symposium on Formal Methods*. Springer. 2006, pp. 300–315 (cit. on pp. 4, 13, 40, 48).

[8] Peter Höfner, Ridha Khedri, and Bernhard Möller. "An algebra of product families". In: *Software & Systems Modeling* 10 (2011), pp. 161–182 (cit. on pp. 4, 13, 40, 48).

[9] 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. 6).

[10] OMG Available Specification. "Omg systems modeling language (omg sysml™), v1. 6". In: *Object Management Group* (2019) (cit. on pp. 7, 30, 37, 52).

[11] Bertrand Meyer. *Handbook of Requirements and Business Analysis*. Springer, 2022 (cit. on pp. 8, 15, 20, 24, 35).

[12] 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. 13).

[13] Suzanne Robertson and James Robertson. *Mastering the requirements process: Getting requirements right*. Addison-wesley, 2012 (cit. on pp. 14, 16).

[14] James Robertson and Suzanne Robertson. "Volere". In: *Requirements Specification Templates* (2000) (cit. on pp. 14, 16, 39, 54).

[15] Ian Sommerville and Pete Sawyer. *Requirements engineering: a good practice guide*. John Wiley & Sons, Inc., 1997 (cit. on pp. 15, 16).

[16] 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 pp. 15, 16).

[17] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2018 (cit. on p. 15).

[18] 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. 15).

[19] 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. 15).

[20] 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. 15).

[21] 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. 15).

[22] 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. 15).

[23] 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. 16).

[24] Jane Cleland-Huang et al. "Best Practices for Automated Traceability". In: *Computer* 40.6 (2007), pp. 27–35 (cit. on p. 17).

[25] 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. 17).

[26] 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. 17).

[27] 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. 17).

[28] 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. 17).

[29] 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. 17).

[30] 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. 17).

[31] 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. 17).

[32] 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. 17).

[33] 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. 23).

[34] 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. 23).

[35] 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. 24).

[36] 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. 24).

[37] "Gherkin Reference". In: (2024). URL: https://cucumber.io/docs/gherkin/reference/ (cit. on pp. 43, 57).

[38] Thomas Chiang et al. "Mapping Requirements to Features to Create Traceability in Product Line Models". In: *Proceedings of the 27th Inter-*

national Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. 2024 (cit. on p. 51).

[39] 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. 55).

[40] 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 p. 57).

[41] Kamil Nicieja. *Writing Great Specifications: Using Specification by Example and Gherkin*. Simon and Schuster, 2017 (cit. on p. 57).

[42] 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. 61).