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

© Copyright by Thomas Chiang, Jan, 2021

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, 61

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 three main pillars that were used to direct this literature review; MDE, PLE, traceability, and requirement engineering. The scope for traceability is limited to connections between PLE, feature models, and requirements. Since another focus for this research is using MDE techniques and tooling we also scope all results to include some sort of MDE. 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 "traceability" AND "tools"
- "traceability" AND ("product line engineering" OR "feature modelling") AND "tools"

- "traceability" AND ("requirement modelling" OR "requirement diagrams") AND "tools"

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

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

2.2 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.

2.3 Product Line Engineering

2.4 Requirement Engineering and Modelling

2.5 Traceability

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.

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 sys-

tem. 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. [12] Average range determined between 5th and 95th percentile of male and female population in Canada.

- Assume weight between 48.82kg and 106.60kg. [13] 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

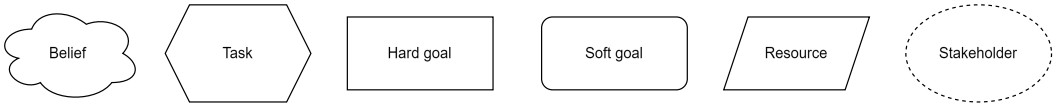

Figure 3.1: Legend of goal diagram elements.

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 [14, 15]. 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 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

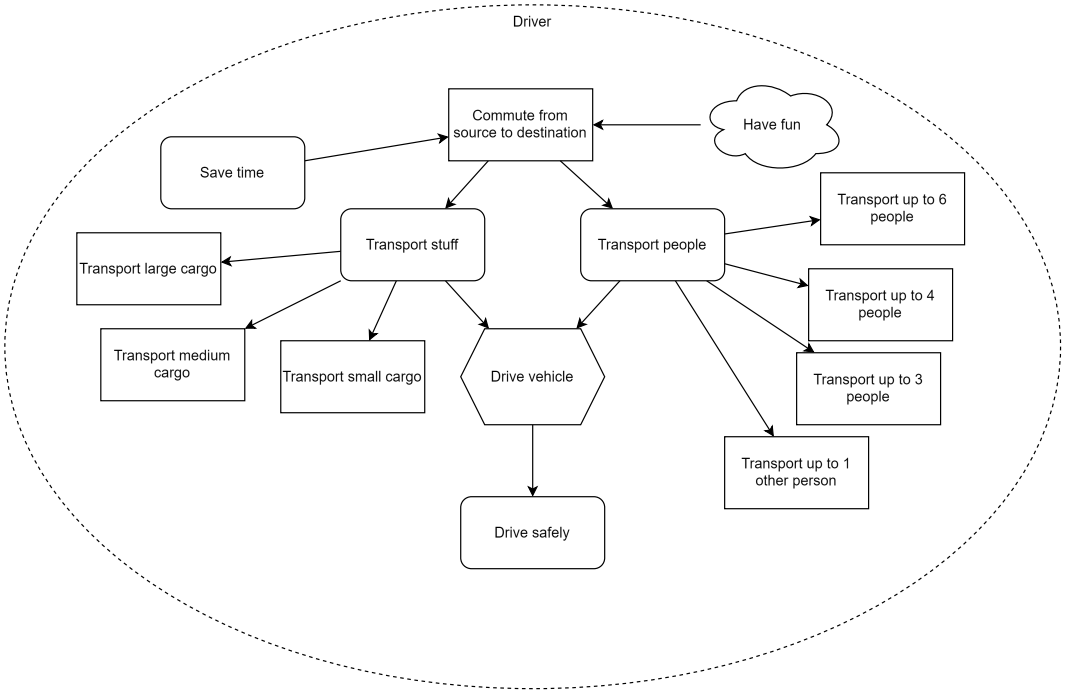

Figure 3.2: Example goal diagram outlining the goals of a driver using a vehicle.

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

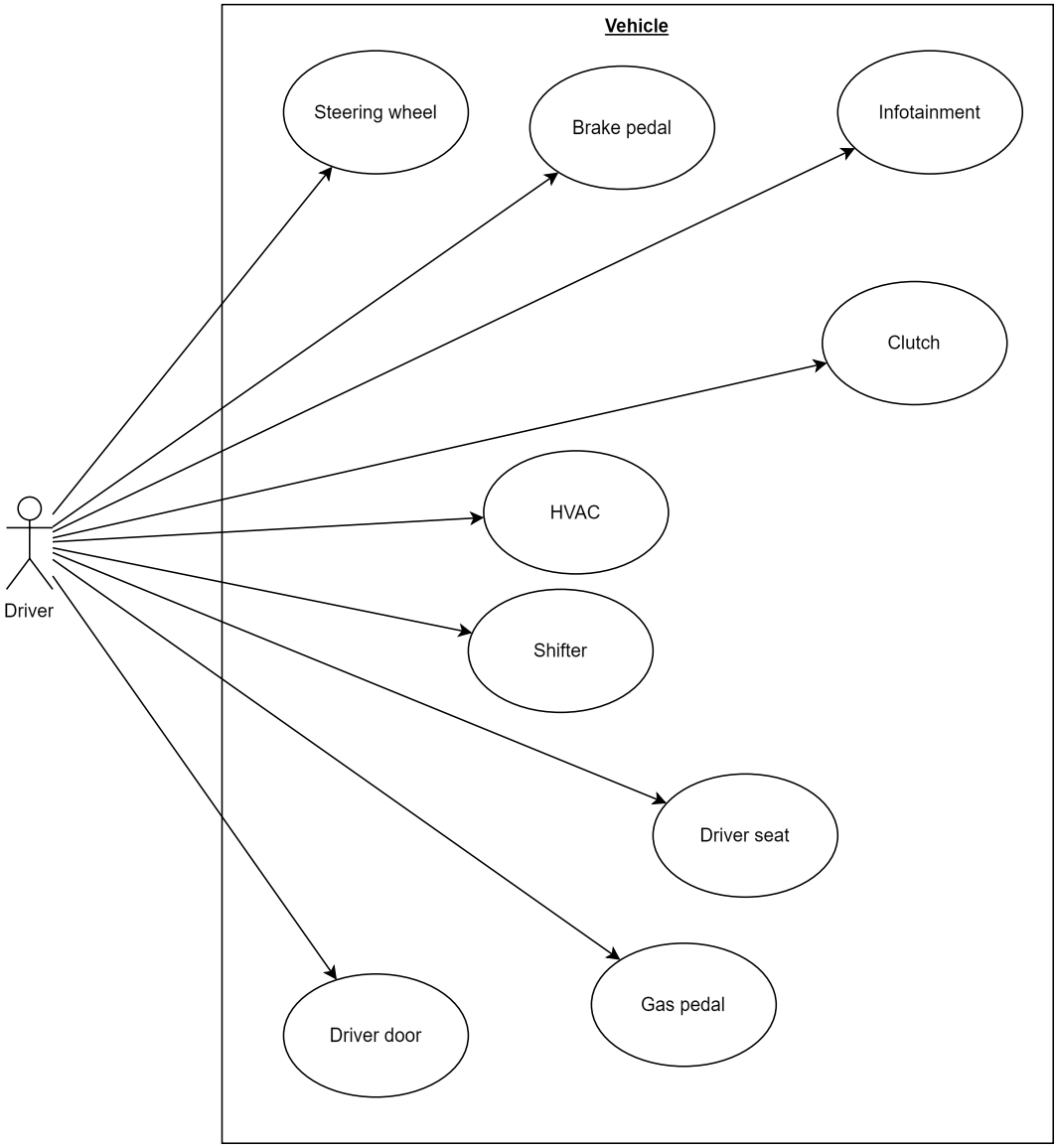

Figure 3.3: A use case diagram outlining what parts of the vehicle a driver will use.

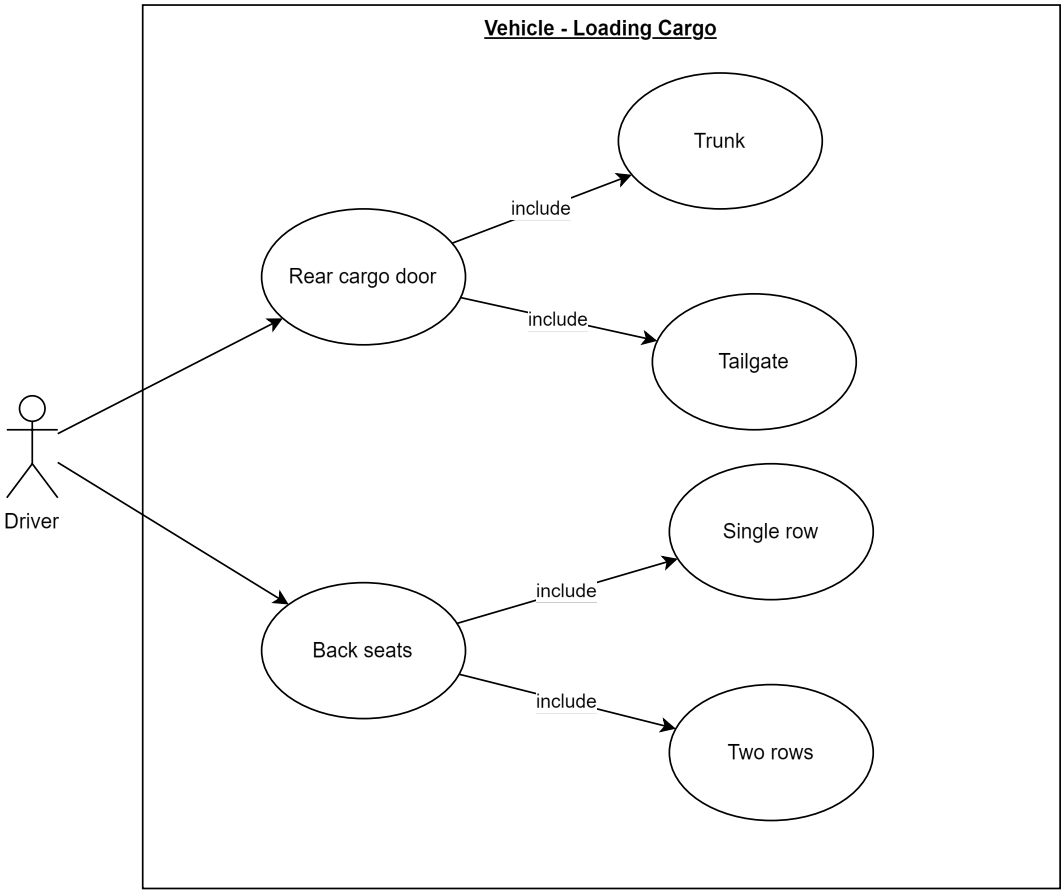

Figure 3.4: UCD scoped by the loading cargo goals

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'. 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. These relationships can sometimes show a use case composition, and by extension a a feature composition, but we do not have any semantics around mandatory and optional features.

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

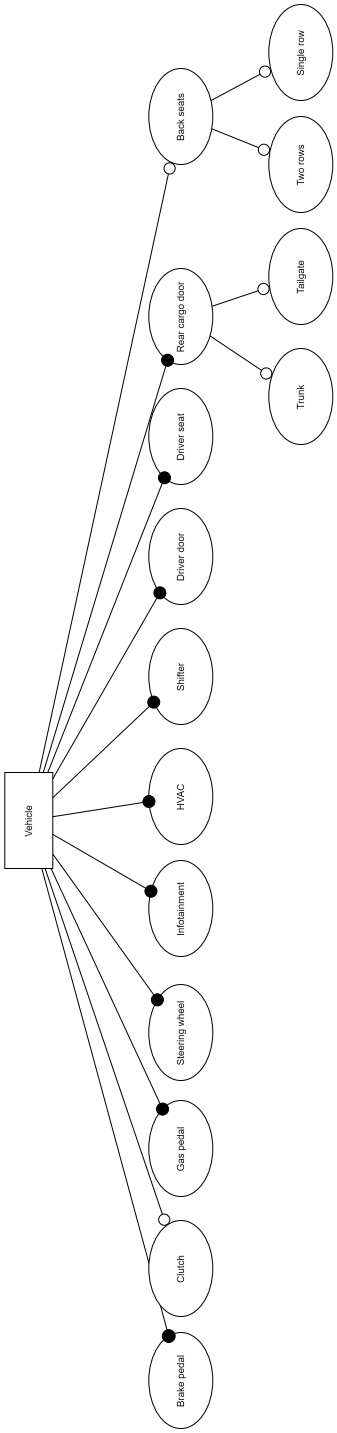

Figure 3.5: Initial attempt at a feature model based on use cases identified in UCD.

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:

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

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

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

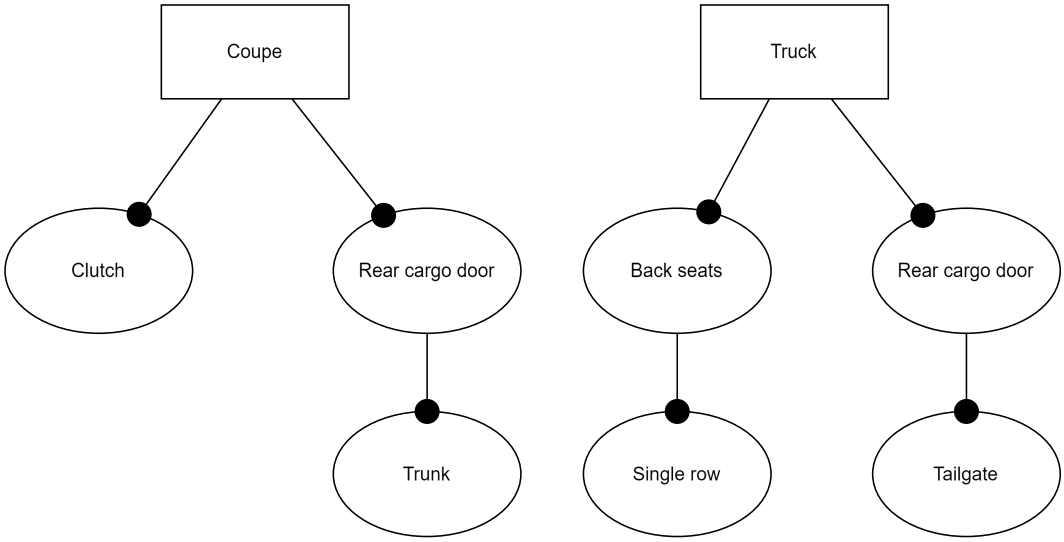

Figure 3.6: Product variants for the initial vehicle feature model. Model is simplified to just the optional features that are mandatory for the variant.

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 Suzanne Robertson Volere requirements [16]. 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

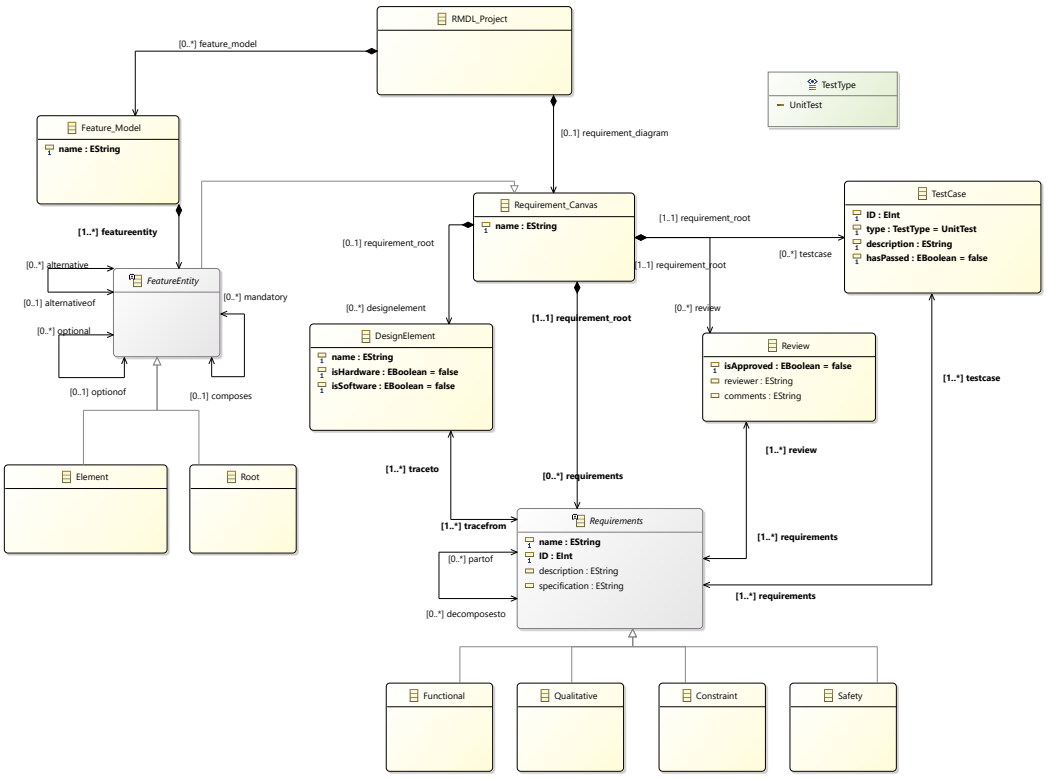

Figure 3.7: CyclicL metamodel. Shows the specification for both the requirment canvas and feature modelling portions of CyclicL.

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 methodology suggests that features be used to encapsulate requirement to support feature-driven development efforts, having a way of representing that relationship in a formal specification such as a metamodel was impactful both for the development of CyclicL and the development of the concept.

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 require-

ments 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 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

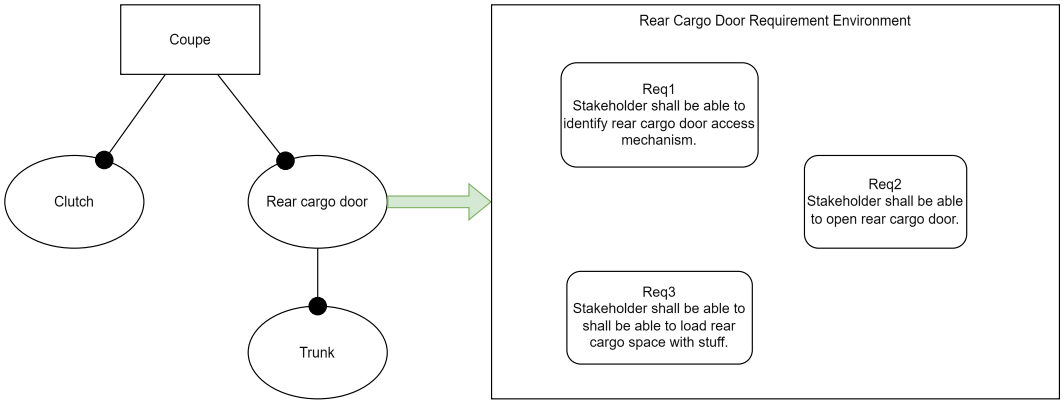

Figure 3.8: Example of how a feature can be opened up to display encapsulated requirements.

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 [17]. 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.

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

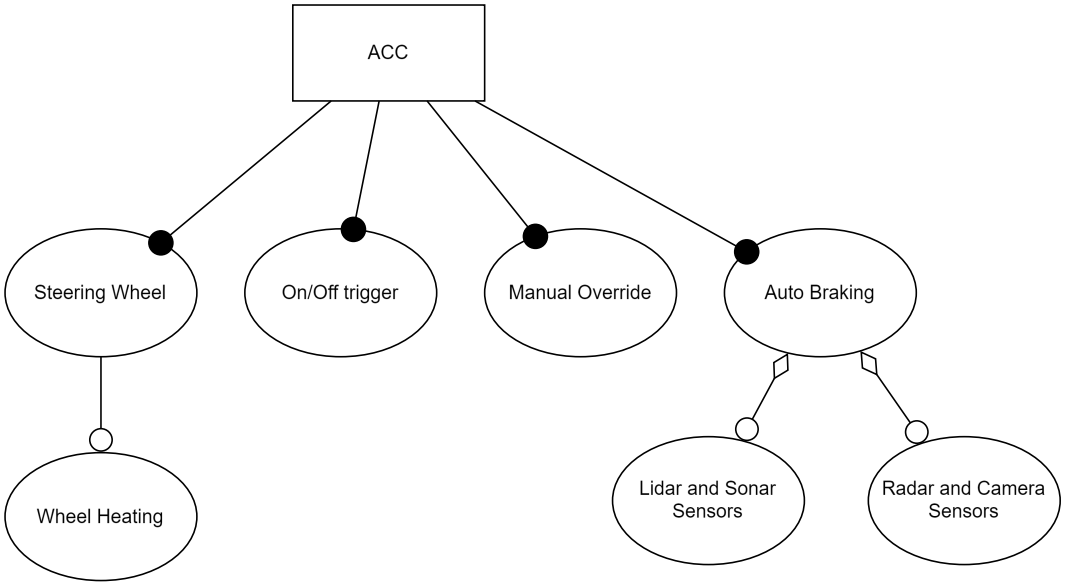

Figure 3.9: Simplified example of an adaptive cruise control feature model.

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 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 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 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 [18]. 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

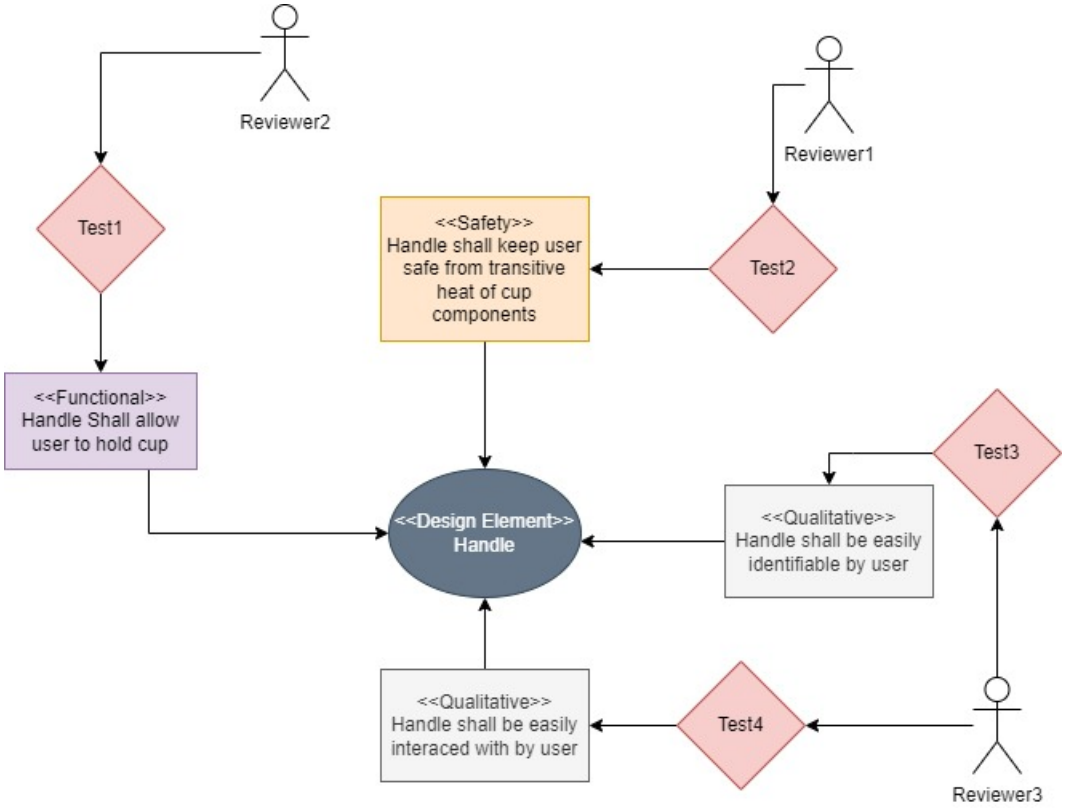

Figure 4.1: Example of a requirement canvas concrete syntax. The example uses a disposable coffee cup with a handle as the focus of the diagram.

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 4.1.

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 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. Thus, our design decision to use features to encapsulate requirements seemed the easiest way to remain faithful

to feature modelling and still give us flexibility with our approach to requirement diagrams. In this process, we diverged significantly from the original specification for requirement diagrams while pursuing traceability. As a result, we named our new modelling approach to requirements within CyclicL as requirement canvases.

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.

4.1.1 Goals and Limitations

4.1.2 Requirements

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 fea-

ture 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 [16]. 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.

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 [19]. Various colors and shapes were used to maintain a bijective relationship between the concrete syntax and intended semantics. For example, in figure 4.2 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.

For the feature modelling portion of CyclicL, we maintained as close to

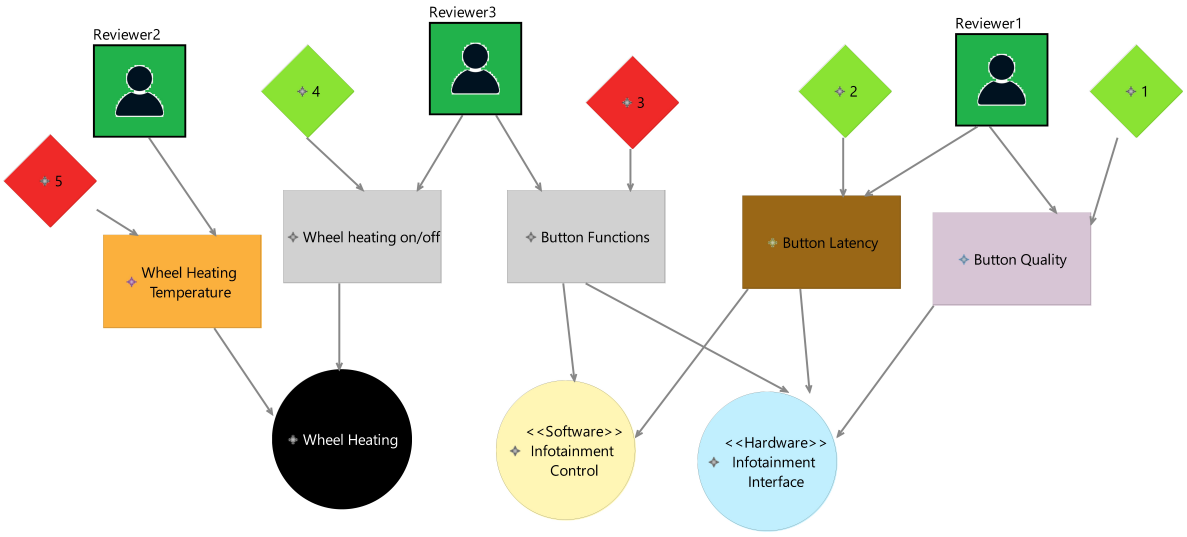

Figure 4.2: An example of the concrete syntax for the requirement canvas using requirements from the automotive domain.

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 4.3 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.

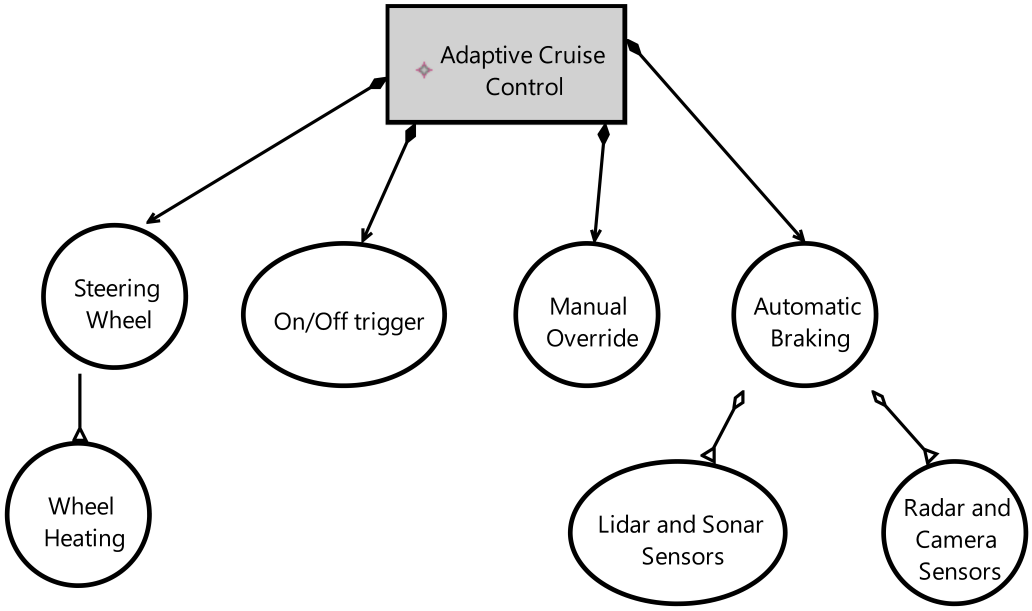

Figure 4.3: An example of the concrete syntax for the feature model using features for an adaptive cruise control system from the automotive domain.

Along with the graphical portion of CyclicL, we also implemented an Xtext-based [20] specification language. The implemented specification language is based on the Gherkin specification language [21, 17]. We chose this as the specification language for our requirements for a few reasons. Gherkin uses a structured natural language for specifications, thus making it relatively approachable for users when defining their requirements. Further, despite using natural language, it uses keywords to provide a predicate logic structure to the specifications. The keyword 'Given' is used for preconditions, 'When' is used for events, and 'Then' is used for postconditions. The combined approach of adding predicate structure to natural language makes it relatively easy to specify requirements and to implement the language within CyclicL.

```
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 4.4: 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.

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 require-

ments 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

5.1 Automotive

5.1.1 Coherence in Automotive Development

5.1.2 Relevance in Automotive Development

5.1.3 Impact in Automotive Development

5.1.4 Efficiency in Automotive Development

5.1.5 Effectiveness in Automotive Development

5.2 Medical Device

5.2.1 Coherence in Medical Device Development

5.2.2 Relevance in Medical Device Development

5.2.3 Impact in Medical Device Development

5.2.4 Efficiency in Medical Device Development

5.2.5 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

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, 13).

[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, 18).

[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, 33, 42).

[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, 33, 42).

[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, 42).

[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, 33, 39).

[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, 33, 39).

[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, 29, 41).

[11] Bertrand Meyer. *Handbook of Requirements and Business Analysis*. Springer, 2022 (cit. on pp. 8, 14, 17, 28).

[12] 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. 17).

[13] 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. 17).

[14] 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. 18).

[15] 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. 18).

[16] James Robertson and Suzanne Robertson. "Volere". In: *Requirements Specification Templates* (2000) (cit. on pp. 31, 44).

[17] "Gherkin Reference". In: (2024). URL: https://cucumber.io/docs/gherkin/reference/ (cit. on pp. 35, 47).

[18] Thomas Chiang et al. "Mapping Requirements to Features to Create Traceability in Product Line Models". In: *Proceedings of the 27th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 2024 (cit. on p. 39).

[19] 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. 45).

[20] 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. 47).

[21] Kamil Nicieja. *Writing Great Specifications: Using Specification by Example and Gherkin*. Simon and Schuster, 2017 (cit. on p. 47).