



TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

HANNU NIEMINEN
MIGRATING A CONTAINER TERMINAL EQUIPMENT CONTROL
SYSTEM TO A SOFTWARE PRODUCT LINE USING EXTRAC-
TIVE AND PHASED APPROACH
Master of Science Thesis

Examiner: prof. Jose L. Martinez
Lastra
Examiner and topic approved on
10.1.2018

ABSTRACT

HANNU NIEMINEN: Migrating a container terminal equipment control system to a software product line using extractive and phased approach

Tampere University of Technology

Master of Science Thesis, 44 pages, 0 Appendix pages

May 2018

Master's Degree Programme in Automation Engineering

Major: Factory Automation and Industrial Informatics

Examiner: Professor Jose Martinez Lastra

Keywords: software product line, software product line, container terminal, container handling, container crane

Software reuse has been researched almost as long as software has existed. Still many companies do not invest in good software reuse methods until they already have several similar products that are being separately maintained. Migrating these software products to utilize reusable components and maintaining them centrally can be a daunting task, and there are only a few tools to help in the process and they haven't yet seen widespread adoption.

Software product lines are one method for planned reuse that have been proven efficient tools in reducing time-to-market and costs, when implemented correctly. However, it can be difficult and requires commitment and monetary investment from the organization.

In this thesis, the steps for migrating a part of a large application into a software product line are explained. The goal of this thesis is to lay out the steps in a clear manner and to provide an example on how an incremental migration can bring some the advantages of software product lines with low-risk. The component being migrated is a part of control systems for container handling equipment from a maritime container terminal.

The thesis consists of three main parts. Information collection and research, designing the software product line and implementing the software product line to develop an example module. Future development directions are also considered and a brief research was made to compare suitable communication technologies for separating the designed software product line module into a standalone application from the larger control system entity.

The work done for this thesis was successful in the sense that applying the designed software product line components to build the pilot application was easy, but the real successfulness can only be measured in the future once the created designs and components have been reused multiple times.

TIIVISTELMÄ

HANNU NIEMINEN: Kontinkäsittelylaitteiden ohjausjärjestelmien siirtäminen ohjelmistotuotantolinjaan käyttäen poimivaa ja asteittaista lähestymistapaa

Tampereen teknillinen yliopisto

Diplomityö, 44 sivua, 0 liitesivua

Toukokuu 2018

Automaatiotekniikan diplomi-insinöörin tutkinto-ohjelma

Pääaine: Factory Automation and Industrial Informatics

Tarkastaja: professori Jose Martinez Lastra

Avainsanat: ohjelmistotuotantolinja, ohjelmistojen uudelleenkäyttö, konttisatama, konttinosturi

Ohjelmistokomponenttien uudelleenkäyttöä on tutkittu lähes yhtä pitkään kuin tietokoneet ovat olleet olemassa. Silti moni yritys ei panosta parempaan tapaan uudelleenkäyttää ohjelmistokomponentteja ennen kuin niillä on monta keskenään samankaltaista ohjelmistotuotetta, joita ylläpidetään erikseen. Näiden samankaltaisten ohjelmien muuttaminen käyttämään uudelleen käytettäviä ja keskitetysti ylläpidettyjä komponentteja voi olla suuri haaste ja työkaluja, jotka helpottaisivat prosessia on vähän ja ne eivät ole laajasti käytössä.

Ohjelmistotuotantolinjat ovat yksi tapa toteuttaa suunniteltu ohjelmistojen uudelleenkäyttö. Ne ovat hyvin toteutettuna todistetusti tehokkaita työkaluja nopeuttamaan ohjelmistokehitystä ja vähentämään kuluja. Niiden toteutus ei kuitenkaan ole helppoa ja vaatii sitoutumista sekä erityisesti alkuvaiheessa rahallista panostusta ja organisaation sisällä uudelleen järjestäytymistä.

Tässä diplomityössä käydään läpi askeleet suuren ohjelmakomponentin muuttamiseen ohjelmistotuotantolinjan mukaiseksi. Työssä pyritään antamaan selkeät ohjeet esimerkkiprojektin kautta, kuinka siirtymä erikseen ylläpidetyistä komponenteista ohjelmistotuotantolinjaan voidaan toteuttaa. Erityisesti keskitytään matalan riskin lähestymistapaan, jossa siirtymää toteutetaan pala kerrallaan, sen sijaan että kaikki muutettaisiin yhtä aikaa. Työssä käsiteltävä ohjelmistokomponentti on osa konttisatamissa käytettävien kontinkäsittelylaitteiden ohjausjärjestelmää.

Työ koostui kolmesta osasta. Tiedon keruusta ja järjestelystä, ohjelmistotuotantolinjan suunnittelusta, sekä yhden pilottikomponentin toteutuksesta. Lopussa tarkasteltiin myös jatkokehitys mahdollisuuksia ja tehtiin lyhyt katsaus erilaisiin tähän sovellukseen sopiviin kevyisiin kommunikaatioteknologioihin, jotta suunniteltu ohjelmistotuotantolinjakomponentti voitaisiin erottaa erilliseksi sovellukseksi.

Lopullinen työ oli onnistunut ainakin pilottikomponentin osalta, sillä se oli helppo ja selkeä toteuttaa, mutta todellinen työn onnistuminen on mahdollista mitata vasta tulevaisuudessa, kun komponentteja ja suunnitelmia käytetään uudelleen useita kertoja.

PREFACE

This thesis was written for Kalmar, which is a branch of Cargotec Finland Oy. I would like to thank Petteri Kylliäinen and Hannu Santahuhta from Kalmar for giving me a chance to write this thesis, guiding me in the process and for showing great patience even when the completion was delayed multiple times. The whole process took over a year, but I learned much from doing it, about many technical aspects as well as working with multiple parties and balancing their opinions. The experience gained from doing of academic research and scientific writing will also certainly be useful in the future.

From Tampere University of Technology I would like to thank my examiner professor Jose Martinez Lastra who, while a very busy man, was eager to give encouragement and advice whenever he had a chance.

Finally, I would like to thank all the people close to me, family, friends and my girlfriend, who have helped me become who I am, supported me through the years of school and hopefully will continue to do so after this when I embark on a new chapter in life.

Tampere, 23.05.2018

Hannu Nieminen

CONTENTS

1.	INTRODUCTION	6
2.	PROBLEM DOMAIN AND PLANNED REUSE	8
2.1	Maritime container terminals	8
2.2	Container handling equipment types.....	9
2.3	Kalmar terminal automation system	11
2.4	Planned reuse.....	13
2.5	Software product lines examples from the industry.....	14
3.	SOFTWARE PRODUCT LINES	16
3.1	Brief history of software product lines.....	16
3.2	Software product lines.....	17
3.3	Extracting software product lines.....	21
3.4	Advantages and challenges of software product lines	22
4.	IMPLEMENTING SOFTWARE PRODUCT LINE APPROACH.....	24
4.1	Domain engineering	24
4.1.1	Domain analysis.....	24
4.1.2	Domain design	29
4.1.3	Domain implementation.....	32
4.1.4	Domain testing	34
4.2	Application engineering	35
4.2.1	Application analysis.....	35
4.2.2	Application design	36
4.2.3	Application implementation.....	36
4.2.4	Application testing	37
4.3	Separating driver module into a standalone application	38
4.4	Future development.....	40
5.	CONCLUSIONS.....	42
	REFERENCES.....	44

LIST OF SYMBOLS AND ABBREVIATIONS

ACS	Access Control System
AGV	Automatic Guided Vehicle
ALV	Automated Lifting Vehicles
AMQP	Advanced Message Queuing Protocol
ASC	Automatic Stacking Crane
CHE	Container Handling Equipment
CS	Control System
CSP	Control System Platform
DM	Driver Module
EIS	External Interface Service
FODA	Feature Oriented Domain Analysis
HT	Horizontal Transport
MIL	Model Interconnection Language
MQTT	Message Queuing Telemetry Transport
RTG	Rubber Tyred Gantry
RMG	Rail Mounted Gantry
SC	Straddle Carrier
SHC	SHuttle Carrier
SPL	Software Product Line
STS	Ship-To-Shore
TEU	Twenty-foot Equivalent Unit
TLS	Terminal Logistic System
TOS	Terminal Operating System
ZMQ	ZeroMQ Message Transfer Protocol

1. INTRODUCTION

The idea of software reuse has been around for almost as long as software development in general and the groundwork for software product lines (SPL) was also made already in the 1970s. Reuse in small scale can be fairly simple thing, but the complexity increases with scale. Many of the earliest attempts at medium or large scale reuse did not turn profit, but the potential time and cost savings were so great that interest always remained high. Software product lines began to gain wider popularity in the 1990s and by the turn of the millennium they had been proven effective.

This thesis was done for Kalmar, which is a branch of Cargotec Corporation. Kalmar provides intelligent cargo handling solutions for maritime container terminals. Currently Kalmar has five individually developed and maintained control systems (CS) for their vehicles, which is not such a high number in itself, but different configurations of them are deployed on 9 different sites, all of which requires a customized version of the control systems. Each site doesn't have all the CSs, but regardless there are currently over 30 development and maintenance contexts, and the number is only increasing with every new products and customers. Considering future plans this number may rise up to 50 or even more in just a few years. This has been recognized as a problem at Kalmar and they have begun implementing planned reuse already. Building everything from scratch or at one time is not feasible in this situation so a phased and extractive approach has been chosen.

The first phase, that was already well underway when this thesis was started, was to identify the similarities in the control system between all the control systems and build one CS platform that every vehicle would use. The goal of this thesis is to lay foundations for phase number two, which is to choose and apply a reuse methodology to the part of the CS that is not common for all CSs. These two parts together will then make up the control system. This latter part that is at center of this thesis is called a driver module.

Chapter 2 will contain descriptions on maritime container terminals in general and the container handling equipment of which's control systems will be handled in the thesis. Also the software stack that control system is in will be described. Finally the approach for reuse is chosen.

Chapter 3 is a detailed look at software product lines (SPL) explaining the different possible methods and steps for designing and implementing software product lines and what are the resulting documents and artefacts. Some challenges and advantages to implementing SPL today are presented to give an understanding on when they are a good fit.

Chapter 4 will describe implementing the steps explained in chapter 3 to the problem domain. The tools and processes followed are presented and all the resulting artefacts such as requirements, reference architecture and feature model are documented. In addition to the reusable artefacts also one implementation of a driver module is created for Kalmar automated guided vehicle (AGV). The pilot driver module is not a final polished product, but more of a proof of concept. Also considerations that came up during the implementation and future development ideas are presented. Most importantly to give direction to near future development a preliminary comparison of applicable communication technologies for separating the driver module into a standalone application from the common control system platform is made and one technology is chosen as a recommendation. Final discussion on the quality of the final product is presented in Chapter 5.

2. PROBLEM DOMAIN AND PLANNED REUSE

In this chapter the basic concepts of maritime container terminals are briefly explained and the container handling equipment (CHE) of which's control systems are relevant to this thesis are described. The software stack that the control systems are a part of and needs to interact with is introduced and finally possible approaches to planned reuse in the problem domain are discussed.

2.1 Maritime container terminals

Maritime container terminals are used to store containers when they are between being transferred from to sea to land vehicles or the other way around. Terminals come in many shapes and sizes and each one is at least slightly different needs from others, which is why solutions for them must always be tailor-made. However the basic functions of each port can be divided into a few functional areas. First is the unloading and loading of container ships. Second is the transport between the ship operation area and storage area. Third is the stacking operations and fourth the landside truck and train operations. These areas are pictured in figure 3 for clarification. [1]

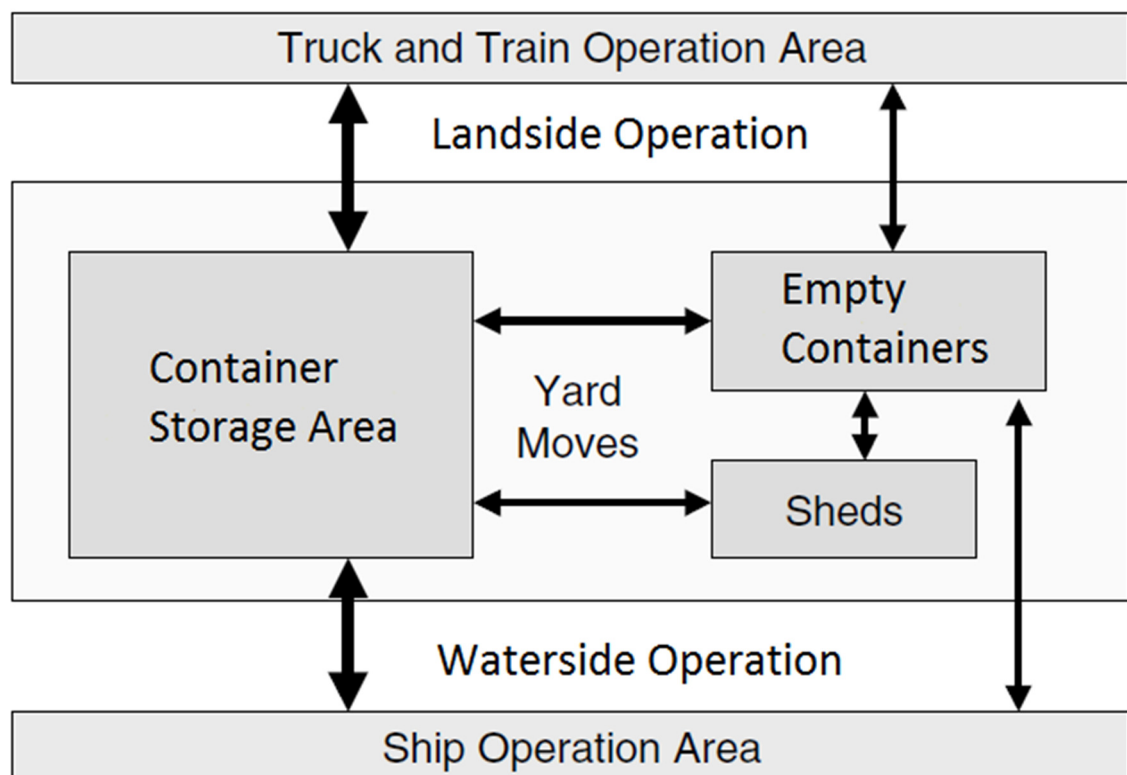


Figure 1: The functional areas of maritime container terminals. Adapted from [1].

The choices for equipment used in each terminal depend on things such as terminal capacity, container throughput and space limitations. If a lot of space is available in relation to the traffic then it is possible to store containers without stacking them or even store them on chasses for cheap but low stacking density solution. [1] For medium-density solution either straddle carriers (SC) or reachstackers can be used for stacking 2-4 containers. The highest stacking densities can be reached with gantry cranes. They include rubber-tyred gantry (RTG) cranes and rail-mounted gantry (RMG) cranes. They organize containers in tightly stacked blocks up 7-8 high and up to 10 wide. RTG cranes are cheaper to deploy and can navigate between blocks, while RMG cranes offer the highest stacking density and efficiency. [2]

Container traffic is usually measured in twenty-foot equivalent units (TEU) or tonnage. One TEU is the volume of a 20-foot-long container. Container traffic has increased significantly during the 21st century. Between 2000 and 2016 the annual traffic has more than tripled from 224 774 536 TEU to 701 420 047 TEU. While the increase rate has slowed down during the last few years a bit, it is still on the rise. [3, 4] Because of this container terminals are required to constantly increase their capacity and efficiency to keep up with the demand and automation is one of the ways to do this. [5]

2.2 Container handling equipment types

Here the CHE types relevant to this thesis are introduced. Meaning that while they cover all of the areas of container handling they only are what Kalmar offers, has offered or plans to offer and competitors might have slightly differing products with the similar names or different names for similar things.

First is the ship-to-shore (STS) crane, which is the largest crane type. STS cranes are used, to move containers between the ship and the shore. They are mounted on rails along the waterside wall. They can be manual, remote controlled or partially automated. They can have spreaders that allow them to pick up two 40ft containers or four 20ft containers at a time for increased efficiency. For small ports traditional quay cranes or lifting equipment aboard the ships can be used instead of STS cranes. [6]



Figure 2: STS cranes [7]

After being unloaded from the ship, a container is transported by horizontal transport (HT) equipment. Manual options for HT include terminal tractors, reachstackers, shuttle carriers (SHC) and straddle carriers (SC). Currently existing automatic solutions offered by Kalmar include automatic guided vehicles (AGV) and automated versions of SHC and SC, which are sometimes called automated lifting vehicles (ALV). [5] Terminal tractors and AGVs don't have capability to lift containers, which means that stacking crane and quay crane operations need to be synchronized with them often leading to waiting time. [1]

SHC and SC are in practice almost the same except for their height. SHC are large enough to drive over one container while holding a container. They are usually only used for transportation, while SCs are slightly larger and able to drive over stacks of up to 4 containers while holding a container. Example of SCs can be seen in figure 2.



Figure 3: Straddle carriers [7]

Rubber-tyred gantry (RTG) and rail-mounted gantry (RMG) cranes are large stacking cranes. Fully automated RMG cranes are branded automatic stacking cranes (ASC) at Kalmar. Main differences between the two are that RTGs are able cheaper to install, while RMGs have a higher stacking density and better performance. There are usually two stacking cranes in one block, one for landside and one for waterside operations, but there can also be only one or in case of ASCs also 3 cranes can operate on one block.



Figure 4: RTG cranes

2.3 Kalmar terminal automation system

The software stack for Kalmar terminal automation solution known as Terminal Logistic System (TLS) can be seen in the figure below. In the figure everything together, excluding Terminal Operating System (TOS), make up the TLS. TLS is a distributed solution with many systems on different platforms.

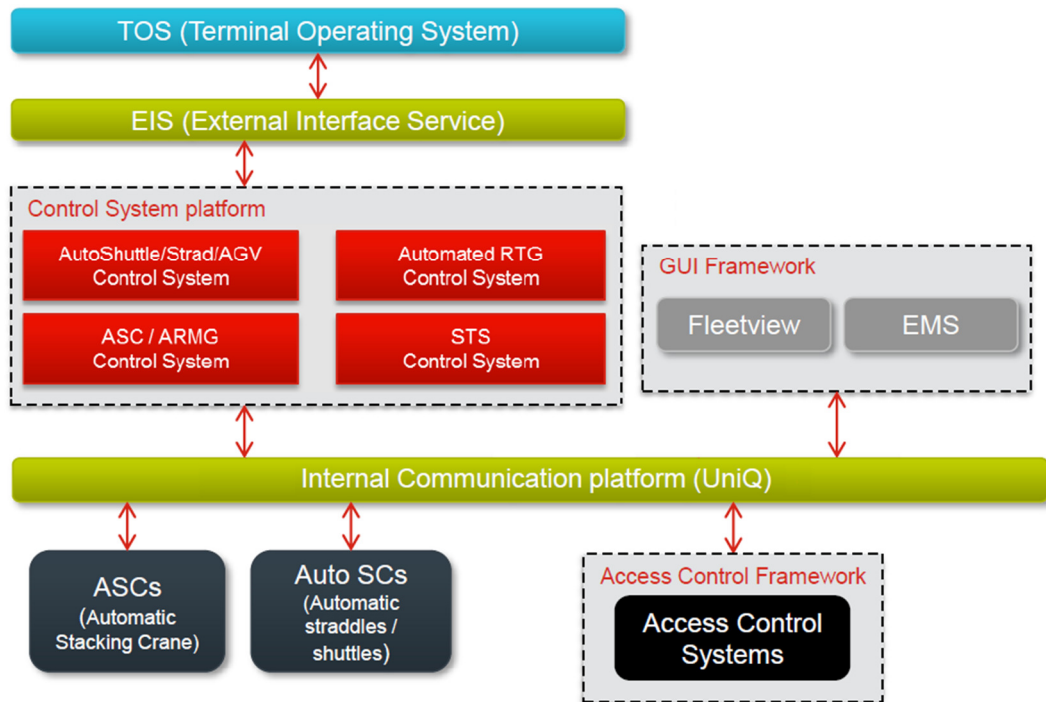


Figure 5: TLS architecture, adapted from [8]

Terminal operating system (TOS) handles all the administrative and logistic functions of the system like reporting, document management and messaging. It can be from Kalmar, client or third party. TLS is designed to work with any TOS and is they are decoupled via the External Interface Service (EIS). EIS is the communication interface between TLS and TOS. All terminal specific components are encapsulated in the EIS layer so that TLS doesn't need to be updated even when a new TOS is introduced. Data moving between TOS and TLS is high level information like job status, job queue, CHE dispatch and container updates.

The control system is responsible for controlling the execution of job orders for each piece of equipment and interaction between other control systems managing different equipment types. One control system is usually responsible for all of the CHE of the same type. Functionalities that are common to all CHE types are all currently packaged into a common platform that could be used as is with every machine known as control system platform (CSP). Common functionalities include things such as job validation, job assignment, routing, state machine bookkeeping, keeping track of containers and obstacles and reserving space so that machines have necessary space to operate. The rest of the control system that is specific to CHE type is contained in the driver module behind an internal interface. The main responsibilities of the driver module are converting the routes into a form that the machine understands handling, hiding the communication interface from the rest of the control system and recovering from small problems.

The internal communication platform is known as UniQ and it acts as a cross-platform data distribution layer and provides common ways to communicate between UniQ services aiming to help solve scalability and reliability issues in a heterogeneous hardware and software environment. It introduces information hiding and data semantics to provide common interface for all applications.

The final layer is the machine on-board software and subsystems like access control. It includes PLC, softPLC and other devices operating on the machines. On-board systems convert the routes received from the control system into the tire rotations and other actions to actually execute the orders.

2.4 Planned reuse

The choices in this thesis have been limited to planned reuse from the start even though there are success stories for opportunistic reuse as well. This is because opportunistic reuse has already been implemented at Kalmar using clone-and-own approach and while it has been beneficial it does not scale very well [9]. Opportunistic reuse works best in small companies when same developers work with multiple projects and can reuse pieces of code that they are already familiar with, which is often not the case at Kalmar.

For building multiple similar products or just one base product with customizability, product line engineering is widely used. According to S. Ouali et al. software product line (SPL) engineering is even considered unavoidable in planning reuse for systems development. [10] SPLs have been successfully used in the industry since the early 2000s [11] so they have also had time to mature as a technology. SPL as defined by Clements and Northrop are “a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.” [12] To clarify SPL is an approach for developing multiple software products with similarities using reusable components. The definition for SPL is fairly loose and it can be implemented in many different ways.

As for the applicability of software product lines in container terminal systems all terminals have different layouts and needs concerning things like throughput and storage capacity, which leads to different combinations of varying amounts of container handling equipment. The domain of modern automated container terminal is well suited for software product lines due to the similar but slightly different contexts met at each terminal and even between equipment inside a single terminal.

Originally most of Kalmar’s equipment has had their own control systems and while opportunistic reuse has been done the results have been separate products with their own maintenance requirements. Since every site that the CHE are deployed on has different requirements this leads to a high number of maintenance contexts, which would quickly

become infeasible with new products and new customers if nothing was done. This was identified as a problem some time ago and steps were taken to begin transition into SPL architecture. [8] The complexity of the control systems, their number and available resources didn't allow for a "big bang" approach of converting everything simultaneously so an incremental approach was chosen. The goal is to have one single control system for all of the CHE, that will take care of assigning jobs, container tracking, space reservation, routing etc. while driver module will contain all the necessary CHE specific functionality. Things like converting the received routes into a form that is understood by the machine, recovering from small problems and taking care of the communication between the machine and the server side control system. The responsibilities of the control system and driver module will be explained in greater detail later.

2.5 Software product lines examples from the industry

Software product lines have been increasingly gaining attention of both research community and the software industry for the past 20 years [13] and many success stories have been reported. [14] The website for The Software Product Line Conference (SPLC) hosts a hall of fame of SPLs that have been well defined, influential, commercially successful and there is enough documentation available that the claims can be verified. Among them are solutions from big industry names like operational flight programs from Boeing, gasoline engine controls from Bosch, electric power generation plant monitoring and control from Toshiba and powertrain control software from General Motors. [15] Other examples of successful SPL implementations of which there is also architectural information available are Koba, Opel, Danfoss and Alcatel-Lucent. [16, 17]

Koba is an industrial automation company that uses SPL for injection-molding and robotics solutions. They apply layered platform architecture. Multiple system platforms are derived from reference system architecture and domain solutions are built on top of each system platform, by exploiting interfaces on the platform. New functionality is added on top of domain solution using cloning and products can also be configured in installation phase for extra fine tuning. [16]

Opel, owned by Groupe PSA, is another company using SPL for developing tools to handle complexity introduced by new technologies such as alternative-fuel and hybrid engines. They have a hierarchical product line of product lines architecture, with focus on highly general components. [16]

Danfoss Drives is a company using SPL for producing frequency converters. They used to use clone-and-own approach, but decided to migrate to SPL using extractive approach with great results. They apply embedded-platform architecture with extensions to realize 14 different main products along with extensions. Variability is realized by conditional compilation. Feature models are used and mapped to code artefacts with commercial tool pure:variants. [18]

At Alcatel-Lucent SPL is used for a telecom software product. They also used extractive approach and paired it with agile principles of incremental and iterative reengineering to migrate a large scale legacy product family into a SPL with the main focus on trying to achieve staged success and early returns. [17]

Every one of the above three companies uses slightly different approach, but still achieves success. Only thing common is that each of them has dedicated teams for each of their software product lines. Though even there are differences in how size of the team and size of the area of responsibility for each team is. [16]

3. SOFTWARE PRODUCT LINES

Software reuse in general has been around since at least 1968 [19] and SPL is only a few years younger. Both of them are still being researched since the ideas are relevant and not all challenges have been overcome. [20] The benefits of reusing software components is easy to understand since being able to reuse existing components that have been well-tested leads to improved code quality, productivity and cost savings. [21] Considering that most software systems are not completely unique, but more like variants of other existing systems the potential time and cost savings of reuse are high. Especially when building large and complex software. [21]

In this chapter first the history of software product line research is briefly reviewed to quickly give an idea of the lessons learned so far. After that software product lines and the steps in their implementation are explained in detail and some relevant technologies are introduced. Following the basic concepts they are expanded upon by explanation of the extractive approach that will be used in this thesis. Then a look at some of the advantages and challenges of SPL are explained and the chapter ends with a look at some of the successful implementation of SPL in the software industry.

3.1 Brief history of software product lines

The beginning of planned software reuse is often attributed to a paper by McIlroy published in 1968 at the first ever software engineering conference. [19] Where he suggested that there should be a subindustry in software for producing reusable components. [22] Following that in 1976 a paper by Parnas “On the design and development of program families” introduced the concept of program families. In it he talks about how to develop a set of similar programs over a period of time by approaching it with a “stepwise refinement”-method of building software in steps that allow forking into different directions to create different programs. [23] This became the foundation for modern software product lines (SPL).

Other important landmarks that affected the evolution of SPL directly or indirectly through advancing the decrease in granularity of the software components and by promoting software architectures were a paper by De Remer and Kron in 1976 “Programming-in-the-large versus programming-in-the-small”, where they talked about programming on two levels, the rise of object-oriented programming through 1980s and 1990s [24], the creation of feature models in 1990, [25] and the introduction of and domain analysis as a part of Draco project around 1980s, [19] which also promoted the idea of reusing design information in addition to code. [26] Around late 1980s the introduction of software architectures, [27] design patterns [28] and frameworks [29] brought software

reuse forward by bridging the gap between whole systems and classes and therefore allowing for design and development on many levels. [30]

Early 1990s saw a paradigm shift towards planned reuse also in the industry [31] this along with standardization of commercial middleware and growing pressure on lower time to market led to the increased interest on software product lines. [32] The first conference on them was held in 1996 [33] and success stories from the industry implementing SPL began to show a few years later. [34, 35] Software product lines were the first intra-organizational software reuse approach that had been proven to be successful and it gained considerable adoption in the software industry by the early 2000s. [11]

Later developments have been mainly about using different technologies to implement SPL. Remote procedure calls, [36] service oriented architecture and [37] dynamic SPL [38] have allowed SPL to evolve to new fields.

3.2 Software product lines

Software product lines (SPL) are a set of software products that share common, managed features and a set of artefacts used in developing the products. The artefacts are known as core assets and include for example requirements, architecture, processes and components.

Software product line development is divided into two phases: domain engineering and application engineering. Domain engineering is the analysis and development of the reusable assets. This analysis can be of concepts of the problem domain when building a system from scratch or be directed on to existing systems. Application engineering is responsible for assembling together the reusable assets provided by domain engineering and developing the application specific parts of the application to if they are needed. [39] The relationship between domain engineering and application engineering can be seen the figure 6 below for clarification.

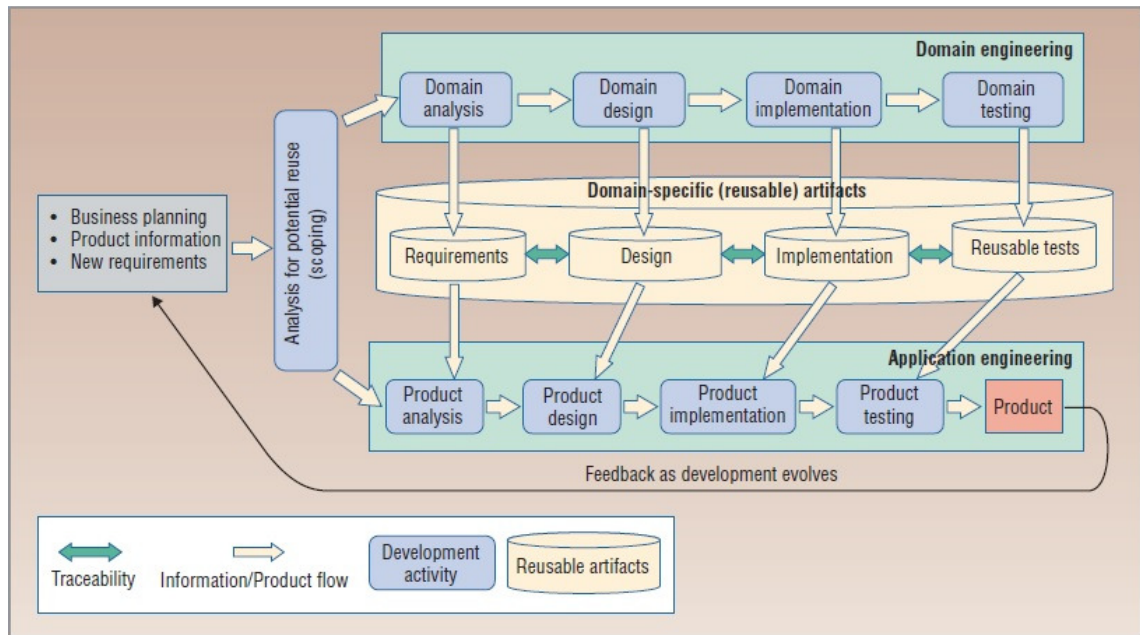


Figure 6: Software product line development cycle. [40]

Domain engineering is usually divided into four steps. Those parts are domain analysis, domain design, domain implementation and domain testing. [39]

Domain analysis is the first step of domain engineering and its output is a domain model. The exact contents of domain model are not consistent in literature, but a survey done on domain engineering by M. Harsu [39] attempted to collect the most common elements from several sources and the resulting list is as follows: domain scoping, commonality analysis, domain dictionary, notations and requirements engineering. [39] Domain scoping is about defining the boundaries for the domain and identifying the subdomains that are important to the SPL and where reuse is economically useful. [41] Commonality analysis studies the requirements and properties of the target domain and should produce a document that defines the commonalities and variabilities in the domain. Domain dictionary defines the terms that should be used when discussing and documenting the domain to allow clearer and precise communication between all the parties concerned with the SPL. Notations define a common way to describe the concepts used in domain modeling. These notations can be for example object diagrams, state-diagrams or data-flow diagrams. [39] In the past 25 years over 50 SPL modeling languages have been developed and choosing the right one for the context and user requirements can be difficult. [42] Things like familiarity should be taken into consideration, but on the other hand it might be confusing if familiar notations are used in inappropriate abstraction levels. Choosing the correct notation is important because they have big role in providing understanding of the domain.

The final step in domain analysis, requirements engineering is about reusing and configuring the requirements for each application in the domain. [39] It is concerned with real-world goals of the system. [43] Requirements need to be gathered, specified, verified,

modelled, planned out, implemented and changed as needed if changes happen later in the development lifecycle. [44] Requirements engineering can also define features in addition to or instead of requirements. [39] Features have had several slightly contradicting definitions in the past [45], but here they will be considered a characteristic of the system that is visible to the user or otherwise relevant to some stakeholder. [39] Documenting the variability of the features is important and there are two main ways to do this: integrated and orthogonal. Integrated documentation commonly uses feature models. Feature models provide notations for feature-properties such as mandatory, optional and alternative. Other notations may also be available depending on the chosen modeling language. Figure 7 shows an example of a feature model diagram for clarity. Orthogonal documentation differs in that it separates variability into a dedicated model meaning that there will be one model for software development artifacts and another for the variability between them. Orthogonal documentation has some advantages over integrated that make it a better choice for large scale systems. [46] However this thesis will be dealing with a domain that has a fairly small number of features so integrated documentation will be used since it can contain all the required information in one diagram.

All the results from domain analysis together form the domain model. [39] A well-defined domain model allows developers to have a clear direction through the development and helps guarantee that the final product is satisfactory to all stakeholders. [44]

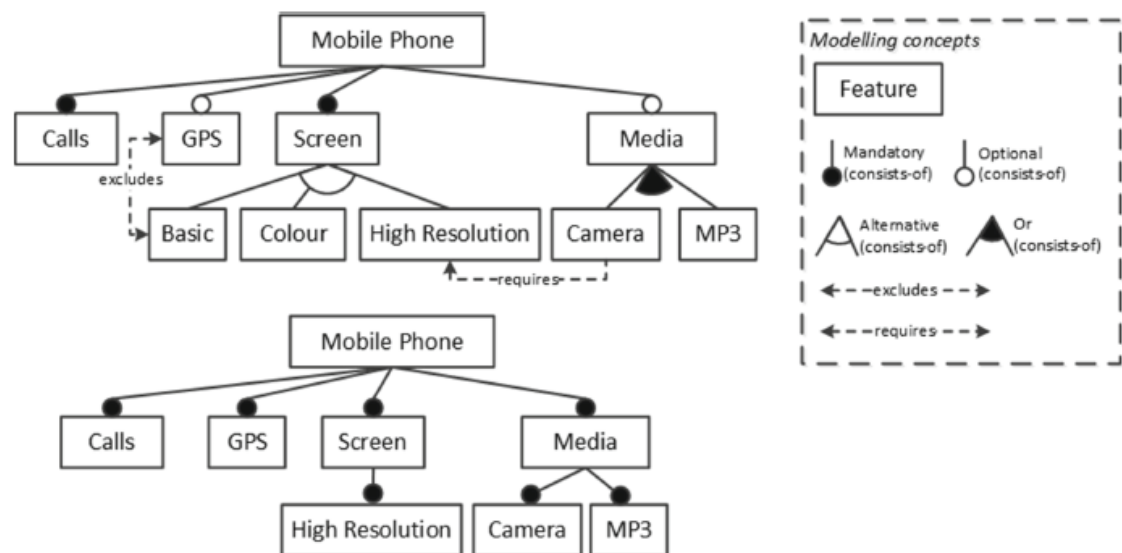


Figure 7: Example of a feature diagram. [47]

Domain design comprises of everything done to define a common architecture for the product line. One of the first actions in this step is choosing the architecture style. SPL architectures have traditionally been component-based, but recently also aspect-oriented and service-oriented approaches have been proposed. [46] Reference architecture has had many definitions one them by Kruchten is: “A reference architecture is, in essence, a

predefined architectural pattern, or set of patterns, possibly partially or completely instantiated, designed and proven for use in particular business and technical contexts, together with supporting artifacts to enable their use. Often, these artifacts are harvested from previous projects.” [48] Quality attributes can also be assessed in domain design. [46]

In domain implementation the designs from previous phases are realized. This means building the components, tools, generators and writing more detailed documentation. [39] When implementing artefacts it is very important to make sure that traceability between the features/requirements from previous phase and the corresponding code artefacts is documented and maintained. Systematic traceability makes it easier to for all stakeholders to communicate and can be used for impact analysis for estimating change effort among other things. [49] Many approaches for implementing variability with common programming languages exist such as inheritance, aspect-oriented programming and conditional compiling, but feature and variability focused new programming languages have also been proposed. [46] Recent increase in demand on runtime adaptation has also lead to increase in research on dynamic software product lines (DSPL) that can be context-aware, post-deployment configurable or runtime adaptive SPLs. [38]

Domain testing aims to produce reusable test cases that can be executed in application engineering to find possible defects, but also for domain engineering to test reusable artifacts before they are used in applications. Test cases should cover different test types including system, integration and performance tests. Tests and quality assurance is maybe even more important in SPL development than single-system engineering since defects in reusable artifacts can affect multiple applications. [46]

After the domain engineering has progressed sufficiently it is possible to begin application engineering. To reiterate, application engineering is about resolving the variability from domain engineering to fit the needs of a particular application, and this way building concrete applications. [46]

First step of application engineering is product analysis. During this step the requirements for a product are specified. In most cases all the necessary requirements should have been specified in domain engineering and this step would be about choosing which requirements to include. [46]

In product design the alternatives from domain design are evaluated in accordance to the application requirements and the best one is chosen. A detailed architecture is then designed based on the reference architecture. Application architecture is often a specialization of the reference architecture, but sometimes it might be required to adjust or extend the architecture to meet the application requirements. Problems in this step can even lead to changes the reference architecture. [46]

Product implementation is the actual building of an application. In ideal case it is possible to build all applications with only the core assets using software configuration and composition techniques or code generation, but in practice this is often not possible and application specific glue code or even modifications to domain artifacts need to be made. [46]

Product testing is the last phase and while some test have been performed in domain testing already they are not enough. In most cases it is impossible to test all variations in domain testing and application specific customer demands might require additional testing that couldn't have been foreseen in domain engineering phase. [46]

The steps described might give an impression that SPL would have to follow waterfall model, but this is not the case and any life-cycle or process model can be used. Even trying to follow the waterfall model is not advised since it can be difficult in real world situations to elicit all requirements in the beginning and technical problems in application engineering might requires changes in domain artifacts, therefore it is good to be prepared to iterate the earlier phases. [46]

3.3 Extracting software product lines

There are three possible approaches to SPL adoption: proactive, reactive and extractive. In proactive the domain engineering is performed to cover the full scope of the products and the reusable artefacts are then used in application engineering to build products. In reactive approach the product family is grown incrementally and domain engineering is applied every time a new product is introduced. In extractive approach common and variable artefacts are extracted from legacy systems and then migrated into an SPL. [50] Extractive approach was the most used industrial practice at 50% according to a survey made in 2013. [51] It is also the one that is used in this thesis.

Extracting reusable artefacts is mainly a manual task since usually documentation on the relationships between variants does not exist. [52] Some tools to support the reengineering process have been proposed, but even authors of those tools recognize that they still need improvement and that their usability impacts the effort saved. Integration of the existing tools into popular frameworks and even automation of the whole reengineering processes are important future research considerations. [50]

Extractive approach, also known as bottom-up approach, has three different phases according to Anwikar et al.: detection, analysis and transformation. In detection phase relevant information such as structure functionality, relationships and data flow are extracted from the source code. In analysis phase variability and commonality are identified from the recovered information and it is reorganized into partitions that segregate the features. Finally in transformation phase transformations are performed on the identified artefacts to migrate the identified features into a functionally separated source to create the SPL.

[53] The steps are very similar to every other SPL development method main difference being in how the steps are performed in practice.

Many strategies exist for the execution of the steps described above the most popular ones being expert-driven, static analysis and information retrieval. Other possibilities are dynamic analysis, search-based and different combinations of any of them. [50] Expert-driven simply means that specialists do everything manually, static analysis is about analysing structural information of static artefacts without considering their execution, information retrieval use the information on identifiers and comments in the code, dynamic analysis are based on analysing the execution of the artefacts and finally search-based strategies apply optimization algorithms. [54] Not all strategies can be applied to all phases of SPL extraction. Most of them focus on detection and/or analysis with transformation having the least options. [50]

On a technical level SPL migration is still considered difficult [55] and there is a lack of comprehensive and usable tools, though some like Bottom-up Technologies for Reuse (BUT4Reuse), a framework for SPL adoption, have been proposed. [56]

3.4 Advantages and challenges of software product lines

Despite their long history software product lines still have many challenges to overcome, but the promise of the many advantages outweighs the risks in many cases. Many of the SPL advantages and challenges are shared with other software reuse methods, while some unique ones exist as well. The most important advantages of SPL are:

- Reduced final costs. Being able to reuse same components again in the several products without having to design, develop or validate everything from scratch means significant cost savings. [14]
- Customization. A software product line approach makes it easy to tailor each product or a family of products according to the customer's wishes with little work. [14]
- Faster time to market. Standardized software or individual development is slow and they might only provide one or a few options. SPL allows for quick reactions to market changes while maintaining high customization. Even if additional components need to be created it is faster to develop on top of well-designed platform than to build from scratch. [14]
- Improved quality. The quality of the software products increases when it's possible to use well tested components that can be relied on to work better than hand-crafted new software. The quality of the reusable assets only increases with use when they are tested more in different combinations. [14]

Still SPL are not perfect and challenges and open research questions still exist as well. Some of them are:

- High upfront cost. SPL requires a larger investment at the beginning than individual development since reusable parts and variabilities that might not be used in the first product need to be designed and implemented. [14]
- Increased maintenance costs. Maintaining all the assets and especially architectural variability up-to-date and synchronized with source code is a complex and costly task. [57]
- Impact of requirement changes. Many things from customer demands to technical obstacles can lead to changing requirements in later stages of development. How to propagate the changes to domain artefacts affected by this is? [46]
- Mapping product line variability and software variability. Product line variability often crosscuts more than one artefact. How to map them together and implement step-wise refinement of product line variability to software variability? [46]

There are many other research questions as well spanning all the fields found in SPL, but finding or listing all of them would serve no purpose in this thesis. [46]

4. IMPLEMENTING SOFTWARE PRODUCT LINE APPROACH

In this chapter the application steps of developing software product lines, as described in chapter 2, is documented. Methods used for each step used are explained and the resulting artefacts are either presented. Documentation may be vague on some details to protect the interests of Kalmar.

4.1 Domain engineering

The implementation will start with domain engineering. The domain is analyzed and documented as well as possible, tools to use are chosen and the reusable core assets are developed.

4.1.1 Domain analysis

First the scope must be strictly defined. Fortunately for this thesis the scope is fairly clear, but for the definition some background information is required. Previous work has already identified all the commonalities that can be found in every one of the control systems for every CHE type and these commonalities have been or are currently being implemented into the control system platform (CSP). Currently CSP is used in two different control systems (CS), which are responsible for three CHE types; RTG, SHC and SC. All other control systems are considered legacy systems that have similarities between each other and CSP, but that similarity is not yet effectively leveraged for reuse.

In the future every CS will consist of two parts: CSP and a driver module (DM). Currently driver modules exist in the two CSs using CSP, but they are simply copy-and-paste of the relevant features from the legacy systems and therefore even though the responsibilities are the same the implementations are very different.

Scope of this work is to define and implement reusable core assets for building driver modules that leverage the similarity between all CHE types. These assets are: requirements, feature model to document variability, reference architecture, reusable code artefacts required to implement a driver module for AGV and a pilot implementation of the AGV driver module. Some artefacts that are common between more than one DM and not the one for AGV most likely exist, but they will not be considered in this thesis.

Application specific artefacts must also be created and the pilot DM needs to be tested. Container handling for AGV doesn't contain any actions other than parking and waiting for another crane-type CHE to either pick or ground a container on top of it. Therefore it

is mainly a responsibility of the CSP to synchronize the actions of the AGV with another vehicle. That is why it is enough if the pilot DM is able to perform park moves correctly in an environment with obstacles and report everything as it should.

Next a domain dictionary that collects all the less common or application specific terms and explains them is created to avoid any confusion. The dictionary is presented in table 1 below.

Table 1. Domain dictionary.

Control System Platform (CSP)	Part of the control system containing only functionality that is common to all CHE types.
Driver Module (DM)	Part of the control system containing CHE type specific functionality.
Leg	A unit for CSP routing. Contains a small part of a route or an action such as container pick or ground. Legs are sent to CHE as soon as CS has finished routing, but it doesn't give the vehicle permission to drive.
Grant	A grant is a permission to execute a single specific leg.
NURBS route	Route format understood by freely moving CHE types: SHC, SC, AGV.
RoutePart route	Route format understood by CHE types that work inside a block: RTG, ASC.

Deciding on the notations is the last step before going into the more specific details of the analysis. While previous work has been done on migrating to the SPL approach it has only been about identifying commonality and no notation has yet been used for variability modeling. Since no people with experience on variability modeling are part of the project and available for counseling the decision is based on research and especially a recent guide on choosing a SPL modeling language [42].

The chosen notation is the Feature Oriented Domain Analysis (FODA), which is the oldest option with the original version dating back to 1990. It is a mature, precise and intuitive method for expressing requirements in the form of features. It is the simplest and

lightest among the options and widely used. Main downside of using FODA is that it doesn't keep track of changes reported to the feature model and since features and requirements are often evolving. [42] Tool used for creating the feature models will be FeatureIDE, which is a popular open source plugin for Eclipse that could aid in all stages of feature-oriented development. [58]

However in addition to feature models also textual requirements will be elicited. Textual requirements will be in general lower abstraction than features and can be used to specify details better, while features are important in documenting variability.

Textual requirements will be defined first. Methods for eliciting the requirements were documentation analysis, code analysis, interviewing software developers working on the control systems as well as the principal engineer overseeing the project. Although all the requirements are presented here at the beginning and many of them were defined early, the process has been iterative and the list has evolved throughout the development. All the requirements for driver modules are listed below.

- Hides the communication interface to CHE from CSP.
 - Specifically the functionality that belongs in the driver module and needs to be hidden is:
 - Feeding routes to vehicle
 - Granting routes for vehicle
 - Discarding legs
 - Updating obstructions and other environment information
 - Reporting on vehicle status e.g. position and handled containers
 - Reporting on errors e.g. out of route, connection lost
- CSP-DM interface will be as common between all the CHE types as possible.
- CSP-DM interface will not have any extra unneeded functions for any machine.
 - It is not allowed to leave unused functions from one CHE type to clutter the interface of any other CHE type.
- DM-CHE interface will contain only message forwarding.
 - DM will be separated into a standalone application in the near future so the interface needs to be easy to modify without having to change anything else.
- Is compatible with the current control system platform.
 - Implements the currently used interface through an adapter to allow testing with CSP.
 - Has the redesigned interface under the adapter so that it can be deployed simply by removing the adapter.
- DM-CHE interface will contain only message forwarding.
 - It has to be possible to change the interface without having to modify any other functionality.

- The reference architecture for DM needs for to be simple and avoid extra layers to enhance readability.
- All high level decision making should be included in one component.
- All important decision making for the DM should be included one class so that it is easy to understand the big picture.
 - Everything else should be packed into helper classes to avoid cluttering the main logic class.
- Naming conventions used in CSP should be adhered to.
- Kalmar's guidelines for coding conventions should be adhered to.
- Has common route storage for legs.
 - Route storage is common between all CHE types because it will store the routes as legs before conversions.
- Different route converters should use common interface since use is identical from a high level point of view.
- Performs all commands given to it one-by-one in the order they were given.
- Has a route converter that converts legs into the form that CHE understands.
 - Different converters should stick to common naming convention with functions performing same responsibilities.
- Validates all routes and grants received before forwarding them to CHE.
- Allows aborting all vehicle action and discarding all planned routes.
- Has a synchronization method for reporting position, spreader status and current errors at once.
- Has its own internal thread instead of being on the same one as CSP.
- Allows implementing limp mode for relevant CHE types.
- Allows implementing hibernation for relevant CHE types.
- Allows implementing heartbeat for relevant CHE types.

After requirements the next step is defining features. Features are highly abstracted and their main purpose is to help describe variability and commonality in an easily understandable format, not to explain how the software will be built.

Features based on the requirements have been collected in table 2. The list is not a comprehensive list of all the features of the CHE types. Only features that are relevant from driver module's point of view are included and one feature might include multiple related smaller functionalities. For example having a spreader always means having twistlocks and container handling reports, but they are listed as one feature. DM doesn't do almost anything without directions from CSP or reports from CHE and therefore every feature also contains the required communication functions in the CSP-DM and DM-CHE interfaces for receiving and sending messages relevant to those features.

Table 2. *Feature list*

CSP-DM interface	An interface that is separated from the implementation. Used to communicate orders and reports between CSP and DM.
DM-CHE interface	An interface that is separated from the implementation. Used to communicate orders from DM to CHE and reports from CHE to DM. Differs in implementation between some CHE types.
Spreader	Contains spreader status handling (width, twistlock status, height etc.), handling twistlock restrictions, container handling and reporting.
Hibernate	Hibernate is a low power mode from which waking up takes some time. This feature contains order handling for wake up and sleep-orders as well as status reporting.
Limp mode	When a CHE counters certain type of error it goes to limp mode, where it is still allowed to move, but with limits on speed and it will not accept any pick or ground jobs.
Route converter: NURBS	Route converter that converts legs from CSP into NURBS. These are used by freely moving CHE types such as SHC, SC and AGV.
Route converter: RoutePart	Route converter that converts legs from CSP into RoutePart-type routes. These are used by CHE moving inside a block such as RTG, ASC and STS.
Route storage	A storage that stores legs received from CSP before conversion.
Route validation	Performing checks to make sure received legs make sense.
Grant handling	Storing grants, validating them and mapping them to legs.
Abort handling	Allows aborting all vehicle action.
Fault handling	Recovering from small errors such as short connection losses.
Vehicle mode handling	Handling changes in vehicle mode (automatic, manual, remote).
Synchronization	Reports all statuses (position, errors, spreader report) of the CHE at once to sync them with the CSP.
Heartbeat	Some CHE types send heartbeats and require them from their DM

Also features that are common to every CHE type are listed separately after the feature table. Common features are things like route storage and communication issue handling that while common, must be included in the DM because they are needed to support the other operations that must remain hidden.

The features have then been arranged a feature model, seen in figure 8, to describe the variability. Some notes need to be made about the feature model. It takes into consideration some near future plans for CHE. Terminal tractor, reachstacker and forklift do not currently have automated versions and therefore no route converters either, but they are considered here as having that possibility. The reason for this is that full automation is the vision of the future and designing their DMs to include the possibility of automation is more efficient because taking them into consideration now doesn't increase the total amount of work significantly and might lead to changes in the reusable artefacts that could be difficult to implement once they have been taken into production use.

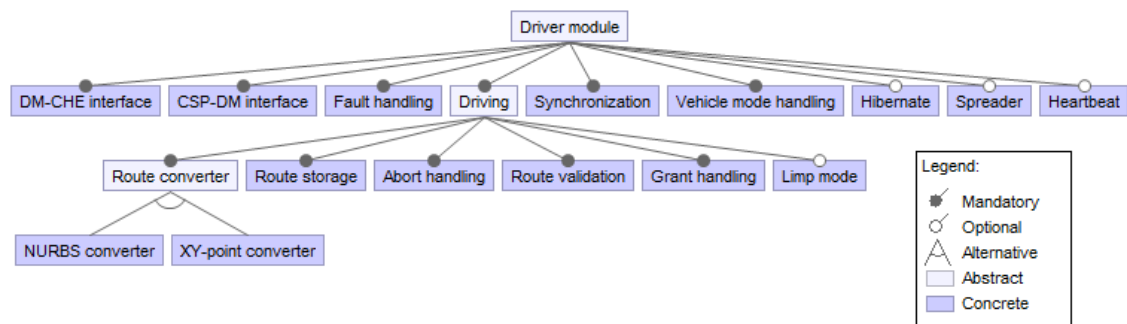


Figure 8: Feature model for of the driver module features.

The feature model is very simple, which is can be as long as the features are explicitly defined. This kind of approach requires separate explanations for features. Another possibility would have been to use smaller grain features which would have resulted in a much larger model, but would have required less explanation.

4.1.2 Domain design

In domain design the main goal is to create reference architecture for the driver module. Concrete software architectures are designed in specific contexts and the effects of the different goals, stakeholders and required functionalities and qualities have been well studied. In contrast the design of reference architectures happens in broader and less-defined contexts since it is meant to facilitate design and development on multiple projects and they are often designed in a non-structured way. [59] The design process for reference architecture is generally poorly described in literature and there are no design standards, which makes it difficult to follow the different methods and approaches. [60]

When designing the reference architecture for this thesis the process called ProSA-RA was followed except that the proposed RAModel was not used in documentation. The

process consists of four steps: information source investigation, architectural analysis, architectural synthesis and architectural evaluation. [61]

Information source investigations is about finding the sources of information that can describe the processes and activities of the domain and investigating them for relevant information. Sources identified for this thesis were two control systems that are already using the CSP and have separate driver modules, all the documentation existing on the control systems and driver modules architectures, responsibilities, interfaces and future visions for each of the CHE types was collected and finally three software developers in charge of different control systems and the principal engineer in charge of the bigger picture were identified as sources of consultation.

Documentation found that proved valuable were textual explanations of the main components of current driver modules and future visions for the driver modules. However when investigating it was found out that the existing driver modules didn't have any architectural documentation available, most likely due to the ad-hoc method they were created with. The CSP documentation was well done, but included the DM only as a single module. Architecture documentation for other control systems without separate DMs also either didn't exist or were not of any use due to the high granularity. To have somewhere to begin an analysis of the current DMs code was made to find out the architectures used. The expertise of the DMs developers was also leveraged to learn how the DMs work and what they considered to be good or bad about the current solutions.

The first driver module to be analyzed is being used for SHC and SC. It implements architecture with six main layers. Each layer has a fairly distinct responsibilities and the implementation of functionality is divided along almost all of the layers. This makes it easy to follow the execution of the code when looking for something specific, but as a downside there is always only a small part of the relevant information available at a time and it can be difficult to grasp the bigger picture of the logic and the relationships of the components. According to interviewed developers in charge of this DM the number of the layers should be reduced for readability.

The second driver module that was analyzed is being used for RTG and ASC cranes. In this DM the main logic has been divided between two components: one for the CSP-DM interface, which has some functionality for collecting and formatting reports for CSP, but is mainly used to forward messages. The other component handles the implementation of the DM-CHE interface and most of the decision making. This component is fairly big with over 2000 lines of code even though it makes use of many smaller classes. The large size of the main logic class is very difficult to avoid due to the large amount of mostly simple functions that the RTG requires. Interviewed developers feel that it is good to contain all the important decisions making in one component, but that great consideration should be put to including everything else in helper classes to avoid cluttering the code.

Second step in the ProSA-RA process, architectural analysis is about identifying the requirements for the system, requirements for the reference architecture and domain concepts. This step was already documented in domain design. The definition and elicitation of requirements is a process that continued throughout the whole development and led to new iterations on the way. The list of requirements on chapter 4.1.1 contains all the requirements identified at all stages and therefore they are not repeated here. Domain concepts would have the same role as features so they will not be separately defined either.

Third step is the architectural synthesis, which means creating the architecture based on the data from previous steps. The exact process is not described in the ProSA-RA since it is very case specific and often the remaining gaps are filled based on developer intuition rather than systematically. For the case in this thesis it was fortunately possible to define fairly strict requirements and to use the existing concrete architectures as a base from which to start working. Most important considerations were separation of responsibility where possible and avoiding unnecessary layers.

A component diagram of the final reference architecture created can be seen in figure 9.

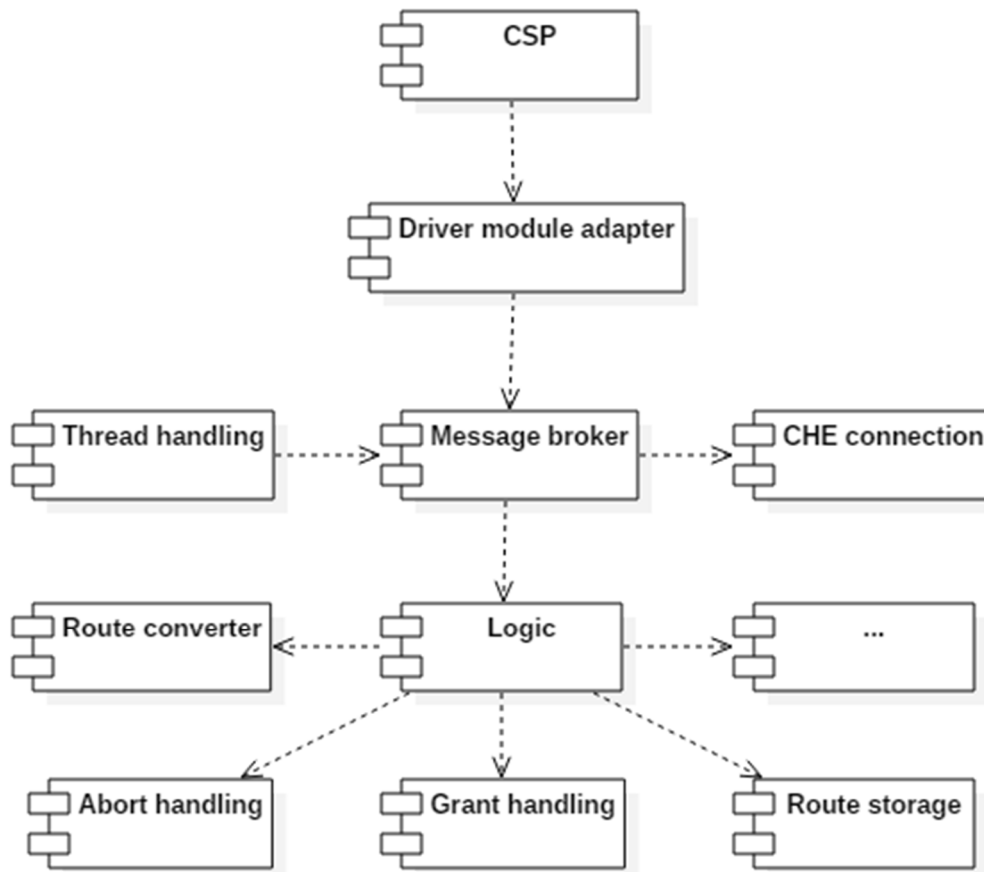


Figure 9: Component diagram of the reference architecture.

The reference architecture has four main components: Driver module adapter, logic, CHE connection and message broker. Driver module adapter implements the interface to the CSP simply forwarding all the messages it receives without altering them in any way. CHE connection does the same for the DM-CHE interface. Logic contains all the decision making and will in all likelihood become a fairly big, which is why when implementing a great considerations needs to be put to contain as much of the execution of algorithms into helper classes to keep the code as readable as possible. Message broker is working in the middle of everything to relay messages between the other three. All messages are routed through the Logic even if nothing will be done to them before sending them forward. Messages are queued in some manner and sent as sets between set intervals.

All the classes that the Logic depends on are helper classes that provide interfaces for the Logic component for important decision making while hiding the algorithm execution. A few example classes that are common to all CHE types are shown in the picture with the “...”-component representing all the varying DM specific additional components.

For architectural evaluation the ProSA-RA proposes a checklist approach where different stakeholders inspect the architecture and answer a set of questions. [62] However this checklist is not freely available and creating one for a project of this scale seems unnecessary. Therefore an evaluation was performed by discussing the architecture with stakeholders in a non-structured manner by discussing the proposed architectures with stakeholders and implementing changes until all parties were satisfied.

4.1.3 Domain implementation

In domain implementation required reusable artefacts are realized. In this project some of them could be extracted and used directly from existing projects, some need to be modified and others need to be made from scratch. The steps taken and results are described textually, but finer details and code will be omitted. All the code artefacts were written in C#.

The work will start from the definition of the CSP-DM interface. This part is the only one where any implementation is done for other future DMs other than the one for AGV. In order to promote readability there should be no unused functions on the interface for any CHE type. Initially an approach of defining a set of functions from which a subset could be chosen for creating an interface for a DM according to its needs was considered, but this led to maintenance concerns where every DM needs to be updated separately when changes are made. The approach that was finally chosen was to define the set of functions as earlier, but to group them into small parts of the interface that could be then combined to achieve all the required functionality. This solution is less flexible than the other, but sufficient for the foreseeable future and allows for better maintainability.

First all the functions required for all CHE types were collected. For the existing driver modules it was simple case of copying them from the existing interfaces, but for other control systems with less clear definitions their parts were derived using the already found functions for other CHE types as a base and using the expertise of the developers to account for the remaining features. The functions were then analyzed for any overlapping and the naming were changed to follow uniform convention and changed if they weren't descriptive enough, resulting in a list of all functions for all CHE types.

Functions were then grouped into sets based on the variability of features between CHE types. Four sets were identified: Base, Container handling, Stacking crane and Horizontal transport. Base is common for all DMs and contains routing, abort, error handling, connection status and vehicle mode handling. Container handling has reporting on handled containers and the status of the spreader. Stacking crane has functions for communicating detailed information on the environment that the RoutePart-type routing requires. Horizontal transport has functions for hibernation and the environment descriptions required for NURBS-type routing.

Three different configurations of these four sets cover all the current CHE types as seen below.

- RTG, ASC, STS
 - Base, Container handling, Stacking crane
- SHC, SC, Reachstacker
 - Base, Container handling, Horizontal transport
- AGV, Terminal tractor, Forklift
 - Base, Horizontal transport

After the definitions first an adapter was implemented to allow communication over the current interface and then an implementation of the newly designed interface was made below it. The interfaces were made to contain as little other functionality than the message pass-through as possible, but until the DM is separated into a standalone application the synchronization function requires the vehicles current statuses as a return value. Since the implementation for messaging between components doesn't allow for direct function calls to the Logic or CHE the synchronization was implemented by making it wait using TaskCompletionSource for external asynchronous operation. This contradicts with the requirement for separating implementation from interface, but it is something that can't be avoided due to how current CSP works. The problem will also get fixed without any extra work when the DM is separated into a standalone application in the near future since the interface implementation will have to be remade and it will no longer require a direct return value so the waiting can be removed.

Creation of the message broker and thread handling components came next. All messages are sent as lists of messages that have common base class and they are handled in their destination based on their derived class that defines the exact message type. Message

broker takes all the lists of messages that have accumulated since last cycle from each main component and forwards them to their destination. Base classes exist separately for each message type and they are called `CspRequestForDm`, `DmRequestForChe`, `CheReportForDm` and `DmReportForCsp`. Thread handling executes the process of collecting and forwarding messages in set intervals. DM needs to perform actions quickly for maximum productivity, but it doesn't have real-time constraints.

Other reusable components can be extracted from the existing DMs. First and the largest is the interface implementation for communicating with the CHE. The implementation is identical on the functionality that overlaps between CHE types using NURBS routing. It has already been implemented in the DM for SHC/SC and can therefore be extracted instead of writing from scratch. Code artefacts for receiving and sending of messages were manually identified, extracted and modified to work with the message broker for communication inside the DM. Fortunately the extracted implementation had easy to understand methods for invoking messages that didn't require deep understanding of the underlying code making this process straightforward even though it is a somewhat large component with around 1500 lines of code in total.

Other extracted reusable components were route converter for NURBS, route storage, route validation, granting, abort handling, hibernate handling, report converters, sanity checker, synchronization and obstruction handling. A few of them were directly copy-and-paste usable such as the route storage, but most required refactoring since they were used in different contexts before and were possible divided to multiple layers. These components all will be used in the Logic component as helper classes.

4.1.4 Domain testing

In most cases, as in this thesis, complete products are not obtained during domain engineering phase meaning that system testing is possible domain testing only in a limited way. Domain testing includes testing of the core assets and possibly the creation of generic test cases. [63]

Testing of core assets was done by creating a simplified version of the Logic component for a mock-up of the complete system and making sure that they output results that make sense. Unit testing was not done very thoroughly with the components that were extracted with only slight alterations to code since they have been well tested already. Most of the other components were interfaces and message passing so making sure that they output sane values with correct types when they are given inputs from CSP was enough testing. If building completely new components a more comprehensive unit testing is recommended since problems in the core assets can affect multiple applications.

AGV is one the simplest CHE types in terms of number of features and almost all of the test cases for AGV system testing can be reused with other CHE types. Below is a list of

system tests that cover everything within the scope of this thesis and can all be reused with any future DM.

- Starting control system (CSP and DM) without available connection to a CHE or a simulator.
- Connecting a single CHE or simulator to DM.
- Break connection to CHE while idle.
- Drive a straight line without obstacles.
- Drive a route with at least one turn. No obstacles.
- Drive routes that require driving in reverse.
- Drive multiple park jobs in a row.
- Break connection to CHE while moving.
- Attempt to route CHE to drive over an obstacle.
- Connect more than one CHE.
- Attempt to route more than one CHE so that their reserved spaces cross when driving.

The execution of the tests is done in the application testing phase when the whole system is implemented.

4.2 Application engineering

In domain engineering the reusability of created assets was the focus. Application engineering is about assembling the reusable assets and filling the remaining gaps with application specific artefacts. With the DMs all of the components will be reusable between at least two applications so they can be made mostly by using the domain assets with only a few or no application specific assets needed.

4.2.1 Application analysis

Application analysis usually contains the requirements that are application specific meaning that they haven't been yet defined in domain analysis. However in the domain of this thesis all requirements are common to at least two different CHE types so all the requirements are included in domain analysis. The only thing left for this phase then is to resolve the variability for the AGV. This is done by resolving all the alternate or optional features in the feature model from chapter 4.1.1 to get a model with only mandatory features. The resulting model can be seen in figure 10.

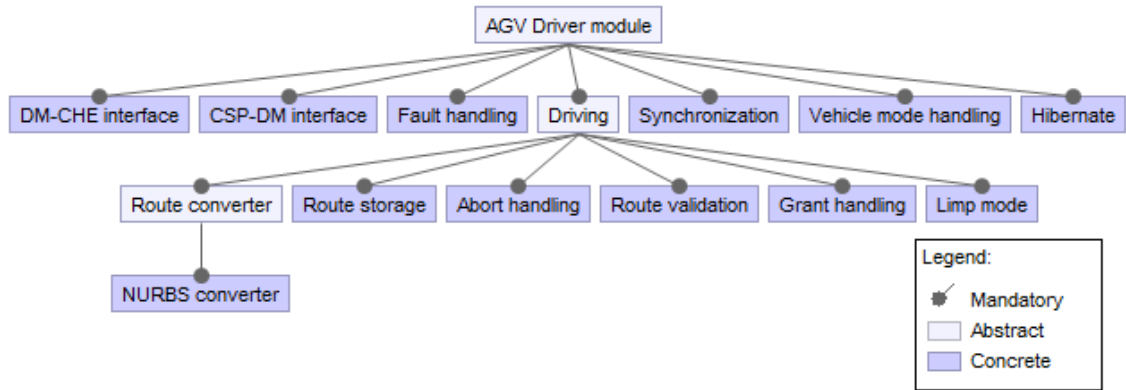


Figure 10: Resolved feature model for the AGV DM.

When all features from the model are implemented the result should be a complete driver module.

4.2.2 Application design

Reference architecture is directly usable with AGV DM without modifications. Only the exact helper classes for Logic should be different between each DM. For AGV the algorithms that are complicated enough to warrant separate components were all described in the domain implementation phase. There are other operations as well but if they require only a few lines of code it is better for readability to have them directly in the code.

4.2.3 Application implementation

As with any software, developing the AGV driver module included many technical decisions on how to implement everything. One of the biggest is the use of Ninject for dependency injection. Dependency injection is a software design pattern that enables developing loosely coupled code with the purpose of improving maintainability, flexibility and testability. Ninject is an open-source technology for helping implement dependency injection. Ninject was used with the DM since it is already well utilized in the CSP and it is easier for developers if the same technologies are used throughout the application.

Events were used in one of the existing DM implementations for intra-DM communication, but the new message broker that passes around lists of messages was implemented for the AGV DM. This was due to wanting to emphasize the importance of the order in which actions are executed. When using a single thread, C# events should also be executed in the order they are received so the change was not necessary, but it was still changed in for extra clarification and to allow for possible future changes to the thread processing.

Interface CSP-DM has functionality in synchronize function since it expects a return value. This is not a problem even though it contradicts the requirements since when DM

is separated into a standalone application this functionality can be erased since it will be decoupled and returning the message will happen with a different function.

A simulator was required for testing and a compatible AGV simulator did not yet exist. Creating a full simulator would have been far beyond the scope for this thesis so another approach was required. A simple mock-up simulator that would be able to respond to some simple and predetermined requests correctly was considered, but ultimately configuring an existing SC simulator to act like an AGV was the best choice. SC and AGV use the same routing so the SC simulator would produce correct position reports from being given the same routes. The simulator had a configuration file that contained descriptions of the models it can handle. A model for the AGV was made to appear to the simulator as an SC that had the dimensions of the AGV and whose spreader maximum height was just above the ground in order to stop it from driving over containers since AGV can't do that. It is impossible to test AGV container handling with the SC simulator without extensive changes, but since container handling was ruled out of scope for this thesis the SC simulator can be used to test all the required cases. Also container handling for the AGV includes only driving under other cranes with container manipulation capability. AGV only has static structural supports to hold the containers that are grounded on top of it in place. It doesn't need to perform any actions other than wait when a container is being picked from or grounded on it. Therefore AGV container handling is a job for the CSPs to synchronize actions between different CHE, but since that is not implemented in the CSP yet it can't be tested.

Dynamic keyword was used in Logic for handling different messages to allow quick prototyping. It has the advantage of requiring only a few lines of code, but in general the use of dynamic is discouraged since it can leave user to discover an error instead of programmer because some errors only appear at runtime. For final version of identifiers could be added to the messages for a non-dynamic solution. This would require more complicated message handlers, but the result would be more reliable.

A factory class was also created as a method for CSP to initialize as many driver modules as required since a single CSP is usually responsible for all the instances of a particular CHE type.

4.2.4 Application testing

Testing AGV DM driving and reporting capabilities were done using the simulator described in application implementation. Testing was performed throughout the application engineering process by adding each feature one-by-one and testing that it worked. Method for testing was to give CSP job orders and to monitor on a graphical user interface what positions and completed Legs the DM was reporting. Test cases used to test the completed system are the ones defined in domain testing on chapter 4.1.4.

All system tests were passed although in one case just barely. All the initializations and connections and routing worked as intended. Only problem that was unsuccessfully attempted to solve is that driving multiple jobs fails too often. This is of course a fairly major problem, but the first jobs worked every time and second jobs most of the time with the probability of failure rising quickly after that. The apparent inconsistency made it difficult to locate the problem and it was decided to accept the current status as the final form for this thesis.

4.3 Separating driver module into a standalone application

The most important future development and one that will happen in the near future is separating the driver module from the control system into its own application. Currently DMs are made by the developers working on the control system, but making DMs requires intricate knowledge of the CHE and it would be preferable to have vehicle manufacturers or other parties with better existing knowledge of the CHE to create the DMs. Separating the DM from the CSP makes this more feasible.

For this purpose a technology needs to be chosen for the interface between the control system and the driver module. The requirements for the technology might vary depending on the final deployment of the driver module, but there are three possibilities. First is that the driver module will stay on the same platform as the control system and be implemented with C#. This is highly unlikely to be the final form of the driver module in any case, but might be implemented as the quick-and-dirty temporary solution. Second possible case is that the driver module stays on the same platform as control system, but might be implemented in another programming language, by Kalmar or some other party. Third case is that in addition to undefined language the driver module might be moved to another platform.

The third choice of varying platforms and languages is the most likely solution. DMs made in-house at Kalmar most likely remain on windows platform and maybe even on the same computer as the CSP, but third party made DMs could be local, onboard the CHE or anywhere between and the platforms could vary as well. Exact list of programming languages that need to be supported does not exist, but support for all mainstream programming languages such as C#, C++, Java and Python would be highly appreciated and all other supported languages are considered an advantage.

In addition to the platform and language support, constraints and requirements considered for choosing a technology are ease of use, performance, reliability and the ability to guarantee message order.

The first option was of course to consider using existing technologies. The UniQ messaging framework could have been used for this purpose. However it is a solution that is fairly heavy and more complex than what is required for this purpose. UniQ is also a

proprietary solution and it would be better if it wasn't necessary for third party DM developers to learn it. Due to UniQ's shortcomings it is planned to be replaced in the long term and the CSP-DM interface would be a good place for one of the first applications for a new technology.

Other technology that has been used at Kalmar already is ActiveMQ JMS, which is in use in the External Interface Service connecting TLS and TOS. JMS and it's has a good set of features, but it is also somewhat heavy. The heaviness is such a big problem because there are cases on very small customer sites where there are only low performance vehicle on-board computers that need to run the message brokers because there can be multiple entities on a single machine that need to be able to communicate even without network connection.

A survey on current inter-process communication was made and out of the many possibilities three lightweight messaging protocols choices stand out thanks to their support for a very large number languages and platforms: Advanced Message Queuing Protocol (AMQP), ZeroMQ Message Transfer Protocol (ZMTP) and Message Queuing Telemetry Transport (MQTT). These three were chosen for a more detailed look and comparison due to their widespread use and promising feature sets.

AMQP and MQTT are two of the most widely used protocols for message oriented middleware. They both implement similar publish-subscribe message delivering techniques where a message is sent by client to a queue on the broker from where all subscribers of that queue receive it as push notification. Both protocols have reliable messaging systems that can guarantee message delivery without losses as long as loads don't exceed buffer capacities. [64] They use TCP as transport protocol and TLS/SSL for security with AMQP also having support for SASL. [65] Both are ISO standards being developed by OASIS. [65, 66]

AMQP is an open standard for enterprise messaging. [67] It differs from MQTT in that messages in AMQP are self-contained and data content is opaque and immutable with no limit on message size. Both point-to-point and store-and-forward message are supported and AMQP support more security features than MQTT. As a downside according to a study from 2015 AMQP messages are sent in inversed order during message bursts. [64] AMQP requires a fixed header of 8-bytes. [65] Implementations of AMQP include RabbitMQ, Apache Qpid. [68]

MQTT is a very lightweight machine-to-machine messaging. It has resource efficient data exchange process and does not specify data format. Messages are published to an address known as a topic, which clients can subscribe to. [64] MQTT provides only point-to-point messaging and is a very basic protocol offering only a few control options, but thanks to this it requires only 2-byte fixed header. Maximum message size is limited to 256MB. Interoperability on MQTT is not as good as with AMQP since MQTT does not

provide message labelling meaning that all clients must know message formats beforehand to allow communication. [65] MQTT implementations include Mosquitto, RabbitMQ and HiveMQ. [69, 70]

As its name suggests ZMTP is the wire-level protocol for ZeroMQ. ZeroMQ supports publish/subscribe, but also request/response patterns. [71] ZeroMQ is capable of high throughput and for example Twitter uses it to transmit data, but it offers only non-persistent message queues which means data can be lost if systems go down. [72] System parts are directly connected meaning there are no brokers or daemons. Message sender is responsible for routing the message to right destination and receiver is responsible for queuing. ZeroMQ has no type specification and has to be used with external serializer. Routing is available, but complex to implement. [70]

Based on the above MQTT with Mosquitto is recommended for the efficiency and simplicity. This solution guarantees message order, offers three levels of Quality of Service (QoS) for reliability, is easy to use and very lightweight. As downsides functionalities such as message priority and routing do not exist as part of the library and if they are needed that will require extra development effort [70]. This brief survey is a preliminary comparison of the popular technologies suitable for this application. A more detailed research might uncover details that were overlooked and adding other possible technologies to the comparison such as Kafka and Constrained Application Protocol (CoAP) would help guarantee that the best option is chosen.

4.4 Future development

Separating driver module to a standalone is a future development direction, but one that is already concrete and in the first stages of development. This section is for presenting two ideas for what could be done to improve the software product line made in this thesis in the more distant future.

For now using the AGV DM as a template for new driver modules might work, but when new DMs are made and new reusable components are added to the core assets it will become important to develop an actual solution for storing all the assets and their documentation so that they are easy to find, use and study the available assets even if the person looking is not familiar with the system. Kalmar is not a small business and people working on projects change and if the components aren't readily available it can lead to wasted time in developing them again.

Another idea would combine all CHE from Kalmar to understand same route format. This would go beyond just the driver module, but would allow streamlining the whole system greatly. However it is likely that neither of the existing formats could be directly made to work with the CHE types using the other, so a completely new format or heavy modifications on both sides would be required. It is possible that the work this would require

would make it infeasible to implement, especially since Kalmar works also with CHE from other manufacturers and it is not possible to change the route formats that they understand so multiple route converters would still be needed. It might not be feasible, but spending time to research and estimate the amount of time and resources needed to make it happen should be made because of the potential this change would have on simplifying the systems.

5. CONCLUSIONS

According to a literature research software product lines a popular way to implement planned software reuse especially in medium and large companies. However they require commitment from the organization and skilled developers in order to gain all the potential advantages. Literature often focuses on greenfield cases and the big-bang approach of developing all the assets for all applications at once when in fact the extractive approach is the most used method in the industry because only a few companies are willing to invest the higher upfront cost of designing components to be reusable as a part of the product line before having multiple similar and successful software products.

In this thesis a part of legacy control systems known as a driver module for multiple container handling equipment from Kalmar were migrated to a software product line. Existing and planned future control systems were analyzed and a set of core assets was developed. An extractive approach of using components from existing systems was used especially in acquiring code artefacts for reuse, but also with the architecture to an extent. Not all of the legacy systems are being migrated to software product line at once since that would require too much resource commitment. The assets created in this thesis cover only the assets required to create a single pilot application as the first step of a phased migration. The pilot application was a driver module for automated guided vehicle. This thesis lays down the groundwork for the software product line and when other control systems are migrated they only have to develop supplementary components to account for the additional functionality not supported by the AGV. Assets that were created in this thesis are requirements, features, reference architecture, code artefacts and test cases. Since AGV is one of the simplest CHE a significant part of the assets developed for it are reusable with all other DMs.

A large amount of time was spent studying the domain as is required for a successful software product lines. Existing control system's code and architectures proved valuable sources of information as expected and when considering future products with only a little documented information, the existing frameworks were used as reference points on which to base decisions. Since the existing systems have differences some streamlining was required and was designed based on the future development directions and developer opinions, but might require adjustments when actually implemented.

The implementation of the pilot DM was realized almost completely with reusable assets and it was made to include an adapter through which it can work with the current system, but can be easily deployed in updated environment just by removing the adapter.

A near future development direction is to separate DM into a standalone application. A preliminary survey and a brief comparison of applicable messaging technologies was

made resulting in a recommendation of using MQTT implementation Mosquitto for the communication due to it being very lightweight and simple.

The success of software product line can't be fully measured at this point in time when only one product has been made using the created assets and that product was at the center of the asset development. However implementing the AGV DM was easy after all the core assets were defined so for now the work is considered at least a partial success.

REFERENCES

- [1] D. Steenken, S. Voß, R. Stahlbock, Container terminal operation and operations research - a classification and literature review, *OR Spectrum*, Vol. 26, Iss. 1, 2004, pp. 3-49.
- [2] B. Brinkmann, Operations Systems of Container Terminals: A Compendious Overview, in: Anonymous (ed.), *Handbook of Terminal Planning*, Springer Science+Business Media, LLC, New York, 2011, pp. 25-39.
- [3] Annual container traffic 2000-2016, The World Bank Group (accessed 22.1.2018), <http://data.worldbank.org/indicator/IS.SHP.GOOD.TU>.
- [4] Annual container traffic 2008-2014, UNCTADstat (accessed 27.4.2017), <http://unctadstat.unctad.org/wds/TableViewer/tableView.aspx?ReportId=13321>.
- [5] J. Böse, General Considerations on Container Terminal Planning, in: Anonymous (ed.), *Handbook of Terminal Planning*, Springer Science+Business Media, LLC, New York, 2011, pp. 3-22.
- [6] C.A. Thoresen, *Port Designer's Handbook: Recommendations and guidelines*, Thomas Telford Publishing, London, 2003, .
- [7] Internal brand bank, Kalmar, Copyright© Cargotec 2014, .
- [8] Kalmar Automation System Architecture, Cargotec Finland Oy, Internal documentation, 2014.
- [9] G. Wang, The Generalized Reuse Framework – Strategies and the Decision Process for Planned Reuse, *INCOSE International Symposium*, Vol. 26, Iss. 1, 2016, pp. 175-189. <https://onlinelibrary.wiley.com/doi/abs/10.1002/j.2334-5837.2016.00153.x>.
- [10] S. Ouali, N. Kraiem, H.B. Ghezala, Framework for Evolving Software Product Line, *International Journal of Software Engineering & Applications*, Vol. 2, Iss. 2, 2011, pp. 34-51.
- [11] J. Bosch, *Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization*, Proceedings of the 2nd International Conference on Software Product Lines, Springer, Berlin, Heidelberg, .
- [12] P. Clements, L.M. Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley, Boston, 2002, .
- [13] R. Bashroush, M. Garba, R. Rabiser, I. Groher, G. Botterweck, CASE Tool Support for Variability Management in Software Product Lines, *ACM Computing Surveys (CSUR)*, Vol. 50, Iss. 1, 2017, pp. 1-45.

- [14] S. Apel, D. Batory, C. Kästner, G. Saake, Feature-Oriented Software Product Lines, 2013th ed. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, .
- [15] SPL Hall of fame (accessed 19.1.2018), <http://splc.net/hall-of-fame/>.
- [16] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, K. Czarnecki, What is a Feature?
A Qualitative Study of Features
in Industrial Software Product Lines, Proceedings of the 19th International Conference on software product line, ACM, pp. 16-25.
- [17] G. Zhang, L. Shen, X. Peng, W. Zhao, Incremental and Iterative Reengineering towards Software Product Line: An Industrial Case Study, 27th IEEE International Conference on Software Maintenance, .
- [18] H.P. Jepsen, J.G. Dall, D. Beuche, Minimally Invasive Migration to Software Product Lines, 11th International Software Product Line Conference, IEEE, pp. 203-211.
- [19] L. Barroca, J.G. Hall, P. Hall, Introduction to Software Project Management, 1st ed. Auerbach Publications, London, 2000, .
- [20] M. Dabhade, S. Suryawanshi, R. Manjula, A systematic review of software reuse using domain engineering paradigms, 2016 Online International Conference on Green Engineering and Technologies (IC-GET), IEEE, pp. 1-6.
- [21] W.B. Frakes, Kyo Kang, Software Reuse Research: Status and Future, IEEE Transactions on Software Engineering, Vol. 31, Iss. 7, 2005, pp. 529-536.
- [22] D. McIlroy, Mass Produced Software Components, 1968 1st International Conference on Software Engineering, .
- [23] D. Parnas, On the Design and Development of Program Families, IEEE Transactions on Software Engineering, Vol. SE-2, Iss. 1, 1976, pp. 1-9.
- [24] B. Stroustrup, Evolving a language in and for the real world: C++ 1991-2006, Proceedings of the 3rd ACM SIGPLAN conference on History of Programming Languages, ACM, New York, .
- [25] K. Kang C., S. Cohen, J. Hess A., W. Novak, A. Spencer Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Software Engineering Institute, Carnegie Mellon University, 1990, Available: <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231>.
- [26] J. Neighbors, The Draco Approach to Constructing Software from Reusable Components, IEEE Transactions on Software Engineering, Vol. SE-10, Iss. 5, 1984, .
- [27] M. Shaw, D. Garlan, Software Architectures: Perspectives on an Emerging Discipline, Prentice-Hall, Inc., New Jersey, 1996, .

- [28] E. Gamma, Design Patterns: Elements of Reusable Object-Oriented Software; Addison-Wesley, Boston, MA, 1995, .
- [29] F. Mohamed, D. Schmidt, Object-Oriented Application Frameworks, Communications of the ACM, Vol. 40, Iss. 10, 1997, pp. 32-38.
- [30] M. Aoyama, New Age of Software Development: How Component-Based Software Engineering Changes the Way of Software Development? Proceedings of 1998 International Workshop on Component-Based Software Engineering, pp. 124-128.
- [31] W. Frakes, Systematic Software Reuse: A Paradigm Shift, Proceedings of 1994 3rd International Conference on Software Reuse, IEEE Computer Society, .
- [32] L. Northrop, Conference Chair's Preface, Proceedings of the First Software Product Line Conference, Springer, 20.12.2017, .
- [33] Software Product Line Conference History (accessed 20.12.2017), <http://splc.net/history/>.
- [34] J. Dager, Cummins's Experience in Developing a Software Product Line Architecture for Real-time Embedded Diesel Engine Controls, Proceedings of the First Software Product Lines Conference, Springer, Boston, MA, pp. 23-46.
- [35] K. Lee, K. Kang, E. Koh, W. Chae, B. Kim, B. Choi, Domain-Oriented Engineering of Elevator Control Software: A Product Line Practice, Proceedings of the First Software Product Lines Conference, Springer, Boston, MA, pp. 3-22.
- [36] Wu He, Li Da Xu, Integration of Distributed Enterprise Applications: A Survey, IEEE Transactions on Industrial Informatics, Vol. 10, Iss. 1, 2014, pp. 35-42.
- [37] D. Krafzig, K. Banke, D. Slama, Enterprise SOA: Service-oriented Architecture Best Practices, Pearson Education Inc., 2005, 21 p.
- [38] M. Bashari, E. Bagheri, W. Du, Dynamic Software Product Line Engineering: A Reference Framework, International Journal of Software Engineering and Knowledge Engineering, Vol. 27, Iss. 2, 2017, pp. 191-234.
- [39] M. Harsu, A survey on domain engineering, Institute of Software Systems, Tampere University of Technology, 2002, Available: [https://tutcris.tut.fi/portal/en/publications/a-survey-on-domain-engineering\(7495329f-8846-4cf5-972b-ca986cd1650e\).html](https://tutcris.tut.fi/portal/en/publications/a-survey-on-domain-engineering(7495329f-8846-4cf5-972b-ca986cd1650e).html).
- [40] S. Hallsteinsen, M. Hinchey, S. Park, K. Schmid, Dynamic Software Product Lines, Computer, Vol. 41, Iss. 4, 2008, pp. 93-95.
- [41] I. John, M. Eisenbarth, A Decade of Scoping - A Survey, SPLC '09 Proceedings of the 13th International Software Product Line Conference, pp. 31-40.

- [42] A. Asmaa, R. Ounsa, S. Nissrine, S. Camille, Selecting SPL modeling languages: A practical guide, 2015 Third World Conference on Complex Systems (WCCS), IEEE, pp. 1-6.
- [43] C. Alves, V. Alves, N. Niu, G. Valena, Requirements engineering for software product lines: A systematic literature review, *Information and Software Technology*, Vol. 52, Iss. 8, 2010, pp. 806-820.
- [44] Z.S.H. Abad, A. Shymka, S. Pant, A. Currie, G. Ruhe, What are Practitioners Asking about Requirements Engineering? An Exploratory Analysis of Social Q&A Sites, 2016 IEEE 24th International Requirements Engineering Conference Workshops (REW), IEEE, pp. 334-343.
- [45] A. Classen, P. Heymans, P. Schobbens, What's in a Feature: A Requirements Engineering Perspective, 11th International Conference on Fundamental Approaches to Software Engineering, Springer Berlin Heidelberg, pp. 16-30.
- [46] A. Metzger, K. Pohl, Software product line engineering and variability management: achievements and challenges, *Proceedings of the on Future of Software Engineering*, ACM, pp. 70-84.
- [47] J. Tiihonen, M. Raatikainen, V. Myllärniemi, T. Männistö, Carrying Ideas from Knowledge-based Configuration to Software Product Lines, *Proceedings of the 15th International Conference on Software Reuse: Bridging with Social-Awareness*, June 5-7, 2016, Springer International Publishing, Switzerland, pp. 55-62.
- [48] E.Y. Nakagawa, P. Oliveira Antonino, M. Becker, Reference Architecture and Product Line Architecture: A Subtle but Critical Difference, in: I. Crnkovic, V. Gruhn, M. Book (ed.), *Proceedings of the 5th European Conference on Software Architecture*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 207-211.
- [49] T. Vale, E.S. de Almeida, V. Alves, U. Kulesza, N. Niu, R. de Lima, Software product lines traceability: A systematic mapping study, *Information and Software Technology*, Vol. 84, 2017, pp. 1-18.
- [50] W. Assunao, R. Lopez-Herrejon, L. Linsbauer, S. Vergilio, A. Egyed, Reengineering legacy applications into software product lines: a systematic mapping, *Empirical Software Engineering*, Vol. 22, Iss. 6, 2017, pp. 2972-3016.
- [51] T. Berger, R. Rublack, D. Nair, J. Atlee, M. Becker, K. Czarnecki, A. Wąsowski, A survey of variability modeling in industrial practice, *Proceedings of the Seventh International Workshop on variability modelling of software-intensive systems*, ACM, pp. 1-8.
- [52] D. Wille, T. Runge, C. Seidl, S. Schulze, Extractive software product line engineering using model-based delta module generation, *Proceedings of the Eleventh International Workshop on variability modelling of software-intensive systems*, ACM, pp. 36-43.

- [53] V. Anwikar, R. Naik, A. Contractor, H. Makkapati, Domain-driven technique for functionality identification in source code, *ACM SIGSOFT Software Engineering Notes*, Vol. 37, Iss. 3, 2012, pp. 1-8.
- [54] W. Assunção, S. Vergilio, Feature location for software product line migration, *Proceedings of the 18th International Software Product Line Conference*, ACM, pp. 52-59.
- [55] L. Li, J. Martinez, T. Ziadi, T. Bissyandé, J. Klein, Y. Traon, Mining families of android applications for extractive SPL adoption, *Proceedings of the 20th International Systems and Software Product Line Conference*, ACM, pp. 271-275.
- [56] J. Martinez, T. Ziadi, T. Bissyandé, J. Klein, Y. Traon, Bottom-up technologies for reuse, *Proceedings of the 39th International Conference on Software Engineering Companion*, IEEE Press, pp. 67-70.
- [57] C. Lima, C. Chavez, E.S. de Almeida, *Investigating the Recovery of Product Line Architectures: An Approach Proposal*, Springer International Publishing, Cham, pp. 201-207.
- [58] S. Krieter, M. Pinnecke, J. Krüger, J. Sprey, C. Sontag, T. Thüm, T. Leich, G. Saake, FeatureIDE: Empowerin Third-Party Developers, *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B*, ACM, pp. 42-45.
- [59] S. Angelov, P. Grefen, D. Greefhorst, A framework for analysis and design of software reference architectures, *Information and Software Technology*, Vol. 54, Iss. 4, 2012, pp. 417-431.
- [60] Francisca Losavio, Oscar Ordaz, Victor Esteller, Refactoring-Based Design of Reference Architecture, *Revista Antioqueña de las Ciencias Computacionales*, Vol. 5, Iss. 1, 2015, pp. 32-48.
- [61] E.Y. Nakagawa, M. Guessi, J.C. Maldonado, D. Feitosa, F. Oquendo, Consolidating a Process for the Design, Representation, and Evaluation of Reference Architectures, *2014 IEEE/IFIP Conference on Software Architecture*, IEEE, pp. 143-152.
- [62] J.F.M. Santos, M. Guessi, M. Galster, D. Feitosa, E.Y. Nakagawa, A Checklist for Evaluation of Reference Architectures of Embedded Systems (S), *The 25th International Conference on Software Engineering and Knowledge Engineering*, Boston, MA, USA, June 27-29, 2013., June 2013, pp. 451-454.
- [63] J. Lee, S. Kang, D. Lee, A survey on software product line testing, *Proceedings of the 16th International Software Product Line Conference*, ACM, pp. 31-40.
- [64] J.E. Luzuriaga, M. Perez, P. Boronat, J.C. Cano, C. Calafate, P. Manzoni, A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks, *2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*, IEEE, pp. 931-936.

- [65] N. Naik, Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP, 2017 IEEE International Systems Engineering Symposium (ISSE), IEEE, pp. 1-7.
- [66] N. Tantitharanukul, K. Osathanunkul, K. Hantrakul, P. Pramokchon, P. Khoenkaw, MQTT-Topics Management System for sharing of Open Data, 2017 International Conference on Digital Arts, Media and Technology (ICDAMT), IEEE, pp. 62-65.
- [67] Advanced Message Queuing Protocol (AMQP), Linux Journal, Iss. 187, 2009, .
- [68] V. John, X. Liu, A Survey of Distributed Message Broker Queues, Computing Research Repository (CoRR), Vol. abs/1704.00411, 2017, .
- [69] R. Neisse, G. Steri, G. Baldini, Enforcement of security policy rules for the Internet of Things, 2014 IEEE 10th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), IEEE, pp. 165-172.
- [70] S. Patro, M. Potey, A. Golhani, Comparative study of middleware solutions for control and monitoring systems, 2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT), IEEE, pp. 1-10.
- [71] B.J. Shaw, P. Amaudruz, D.P. Bishop, Web-based parameter control and real-time waveform display for the GRIFFIN experiment, 2016 IEEE-NPSS Real Time Conference (RT), IEEE, pp. 1-5.
- [72] Z. Wang, W. Dai, F. Wang, H. Deng, S. Wei, X. Zhang, B. Liang, Kafka and Its Using in High-throughput and Reliable Message Distribution, 2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS), IEEE, pp. 117-120.