LAPPEENRANTA UNIVERSITY OF TECHNOLOGY

FACULTY OF INDUSTRIAL ENGINEERING AND MANAGEMENT

DEGREE PROGRAM IN TELECOMMUNICATIONS SOFTWARE

Master's Thesis

Tommi Kankaanranta

SOFTWARE PRODUCT LINE FOR POWER CONVERTERS

Examiners: Professor Jari Porras

D.Sc. (Tech.) Kimmo Rauma

Supervisors: Professor Jari Porras

D.Sc. (Tech.) Kimmo Rauma

ABSTRACT

Lappeenranta University of Technology

Faculty of Industrial Engineering and Management

Degree Program in Telecommunications Software

Tommi Kankaanranta

SOFTWARE PRODUCT LINE FOR POWER CONVERTERS

Master's Thesis

2014

108 pages, 31 figures, 27 tables, 5 appendices.

Examiners: Professor Jari Porras

D.Sc. (Tech.) Kimmo Rauma

Keywords: power converters, software product lines, software architecture styles, component-based development, variation management, embedded systems software

This work was commissioned by Visedo Ltd. Objectives of the work were to research the current state of Visedo Ltd.'s software development practices, identify steps for improvement and develop guidelines for future improvement based on the observations. Visedo Ltd.'s software development practices were analysed through four selected focus areas: architecture style, component-based development practices, software product line practices and management of diversity. Observations were made on the four selected focus areas. Based on the observations guidelines for improvement were suggested for the structure of the architecture, distribution of the components, the adoption of component composition and the development of component versioning strategy. Also a software product line structure which combines the layered architecture and component-based architecture styles was proposed. The proposed software product line structure enables to manage a product range consisting of a population of low-end and high-end products.

TIIVISTELMÄ

Lappeenrannan Teknillinen Yliopisto

Tuotantotalouden tiedekunta

Tietoliikennetekniikan ohjelmistojen koulutusohjelma

Tommi Kankaanranta

OHJELMISTOTUOTELINJA TEHONMUOKKAIMILLE

Diplomityö

2014

108 sivua, 31 kuvaa, 27 taulukkoa, 5 liitettä

Työntarkastajat: Professori Jari Porras

TkT Kimmo Rauma

Hakusanat: tehonmuokkaimet, ohjelmistotuotelinjat, ohjelmistoarkkitehtuurityylit, komponenttipohjainen ohjelmistokehitys, ohjelmistotuotevariaatioiden hallinta, sulautettujen järjestelmien ohjelmistot

Keywords: power converters, software product lines, software architecture styles, component-based development, variation management, embedded systems software

Työn tilaajana toimi Visedo Oy. Työn tavoitteina oli tutkia Visedo Oy:n ohjelmistokehityksen nykytila, tunnistaa seuraavat parannuskohteet ja antaa ohjeita havaittujen parannuskohteiden korjaamiseksi. Visedo Oy:n tehonmuokkain ohjelmistokehityksen nykytilaa käsiteltiin neljän valitun osa-alueen näkökulmasta: ohjelmistoarkkitehtuurityyli, komponenttipohjainen ohjelmistokehitys, ohjelmistotuotelinjojen kehitysmenetelmät ja ohjelmistovariaatioiden hallinta. Valituilla osa-alueilla havaittujen parannuskohteiden perusteella annettiin korjausehdotuksia: ohjelmistoarkkitehtuurin rakenteeseen, komponenttien jakautumiselle, komponenttien koostamiselle ja komponenttien versioinnille. Lisäksi ehdotettiin uudenlaista ohjelmistotuotelinja rakennetta, joka yhdistää kerros- ja komponenttipohjaiset arkkitehtuurityylit mahdollistaen ominaisuuksiltaan eroavien tehonmuokkain ohjelmistojen hallinnan.

PREFACE

The work for this master's thesis was done for the Lappeenranta University of Technology in Lappeenranta between August 2013 and May 2014. The work was commissioned by Visedo Ltd. The list of people to whom I'm grateful for making this, long and at times demanding project, happen is a long one and here's my best effort in remembering all of them. I'm thankful for my work's supervisors Jari and Kimmo for guiding me through this. Kimmo, who is also responsible for employing me to Visedo Ltd., has been especially hard on pushing me forward with this work - I thank you for that. Staff at Visedo Ltd. was always supportive of this work and therefore thank you all.

Obviously I'm grateful to my wife Henriika who has been very supportive of my work and had to take care countless times of everyday things that I couldn't. Also I would like to thank my children Nikke and Emil for bearing (at times) an absent-minded father who wasn't able to spend time with you as much as he would have wanted on several occasions. Finally I would like to thank Risto-Matti for checking my grammar and all of the children's grandparents, Paula, Markku, Ansa and Matti for the help in taking care of the children when it was needed the most.

Tommi Kankaanranta

10 May 2014, Lappeenranta

CONTENTS

ABBREVIATIONS AND SYMBOLS

A	Ampere, unit of electrical current
AC	Alternating Current
ActiveX	Software component framework technology by Microsoft Corporation
ADL	Architecture Description Language
C	Coulomb, unit of electrical charge
C	Programming Language
C++	Programming Language
CAN	Controller Area Network
CANopen	CAN-protocol standard for automation applications
CBD	Component-Based Development
CO_2	Carbon Dioxide
COM	Component Object Model
D-BUS	Inter-process communication bus messaging system
DC	Direct Current
DCOM	Distributed Component Object Model
ECU	Engine Control Unit
EJB	Enterprise JavaBeans
FLASH	Non-volatile memory
GNOME	Desktop environment
GUI	Graphical User Interface
$\mathrm{I}_{\mathrm{RMS}}$	Current, RMS
$\mathrm{i}(t)$	Instantaneous current at given time t
J	Joule, unit of electrical energy

JAVA	Programming language
J1939	CAN-protocol standard for vehicle applications
KDE	K Desktop Enviroment
NO_x	Nitrogen Oxide
OSI	Open Systems Interconnection
$\mathrm{P_{AVG}}$	Average power
$\mathrm{p}(t)$	Instantaneous power at given time t
RAD	Rapid Application Development
RAM	Random Access Memory
RLC	Resistor-Inductor-Capacitor
RMS	Root-Mean-Square
s	Second, unit of time
SPF	Software Product Family
SPL	Software Product Line
SPLE	Software Product Line Engineering
UI	User Interface
USB	Universal Serial Bus
V	Volts, unit of electrical potential energy
$\mathrm{V_{RMS}}$	Voltage, RMS
$\mathrm{v}(t)$	Instantaneous voltage at given time t
VRM	Variation Risk Management
W	Watts, unit of electrical power
$\mathrm{w}(t)$	Energy at given time t
XML	Extensible Markup Language

1 INTRODUCTION

Clearly defined and reusable software components are desirable goals for products that contain embedded software. During the lifetime of the product software defects will be fixed, new features added and older features deprecated. It would be also convenient if all or most of the effort put into existing software assets could be reused when new product variants are developed. For a company reutilising previous work has many benefits such as: faster product development cycle times, relaxed product development personnel requirements and the ability to build on what has been tested in previous products.

This master's thesis work is organized into seven chapters. The first chapter is the introductory chapter covering the background and objectives of this work. The second chapter will cover the basic electrical characteristics and functionalities of power converters using an electric drivetrain as an example. After an introduction to the basic properties of power converters in electric drivetrain applications, the theoretical background for software architectures will be explored in the third chapter. The theory behind software architecture styles, software product lines and the variation of software will be discussed in the fourth chapter. The fifth chapter discusses the practical aspects of this work with associated findings regarding the software product line of Visedo Ltd. and its current state. And in the sixth chapter general discussion and future developments will be explored based on the findings of the practical section of this work. Finally, the conclusions will be presented in the seventh chapter.

This work was commissioned by Visedo Ltd. The company develops electric drivetrains for hybrid work machines, buses and marine vessels. During the author's employment at the company, it has become clear that different applications have varying electrical requirements such as the AC or DC output characteristics of a power converter. A common usage for power converters is controlling the motor or the generator in conjunction with energy storage such as a super capacitor unit. On a system level, the package consisting of power converter, motor, generator and energy storage is called an electric drivetrain. One of the incentives for building hybrid vehicles or equivalent solutions has been the emission reduction targets set by many global actors. For instance, the United States of America and the European Union, have environmental regulation and reduction targets in place for NO_x and CO_2 emissions of non-road diesel engines where targets have been road mapped up to year 2015 [1].

By having a flexible embedded software platform for power converter products, the company is able to create new product variants on custom basis, depending on the customer requirements. Common differentiating characteristics of a power converter are e.g. its AC output power and communication stack implementation. Due to the limited size of RAM and FLASH memory storage space (a common trait of embedded systems) it is often desired that a product variant contains only the required parts of the software tree in compiled form. For example, two industrial communication protocol variants, CANopen and J1939, based on the controller area network (CAN) standard, are not required to be supported at the same time by the firmware. As a consequence the binary-size and the RAM footprint of the power converter firmware can be optimized together with differentiating characteristics of the underlying hardware.

Figure 1 gives an idea how the size of embedded software has been evolving over the years. By choosing automotive embedded software, the green triangle in Figure 1, as the reference software type, one can see that the size of software increased by a factor of one hundred in a period of mid-1990's to 2005 in terms of the number of object instructions [2]. Judging by the trend, one can see that the size is still increasing.

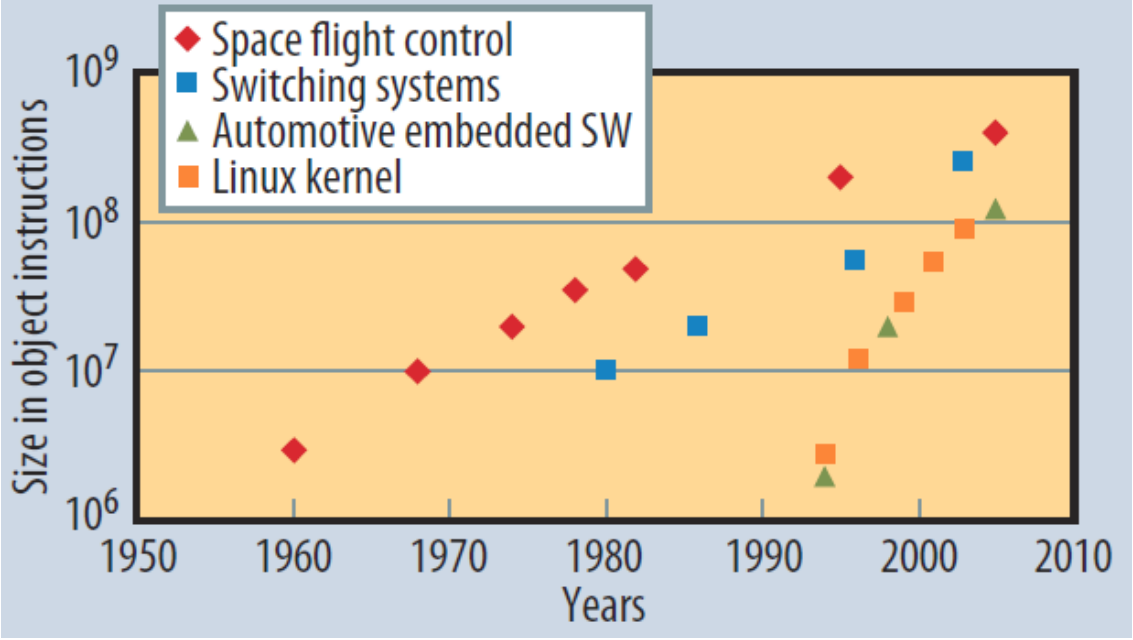

Figure 1. Evolution of embedded software size over time [2].

How does embedded software impact people's lives? The answer is that in many ways, most of which are tangible to a large number of a people. The range of products containing embedded software is very diverse, from small Universal Serial Bus (USB) memory sticks to heavy construction equipment just to give an idea of the scope.

A few examples of embedded software deployments, and their respective sizes, can be seen from Figure 2. Mobile phone segments combined form the largest number of embedded software deployments, followed by RT-Linux (Real Time), washing machine and automotive applications. Heavy construction equipment, such as excavators and wheel loaders could, however, be considered, in terms of software, close relatives of automotive software and thus comparable in size and features. Cars, mobile phones and washing machines are probably prime examples of products containing embedded software used daily by many people.

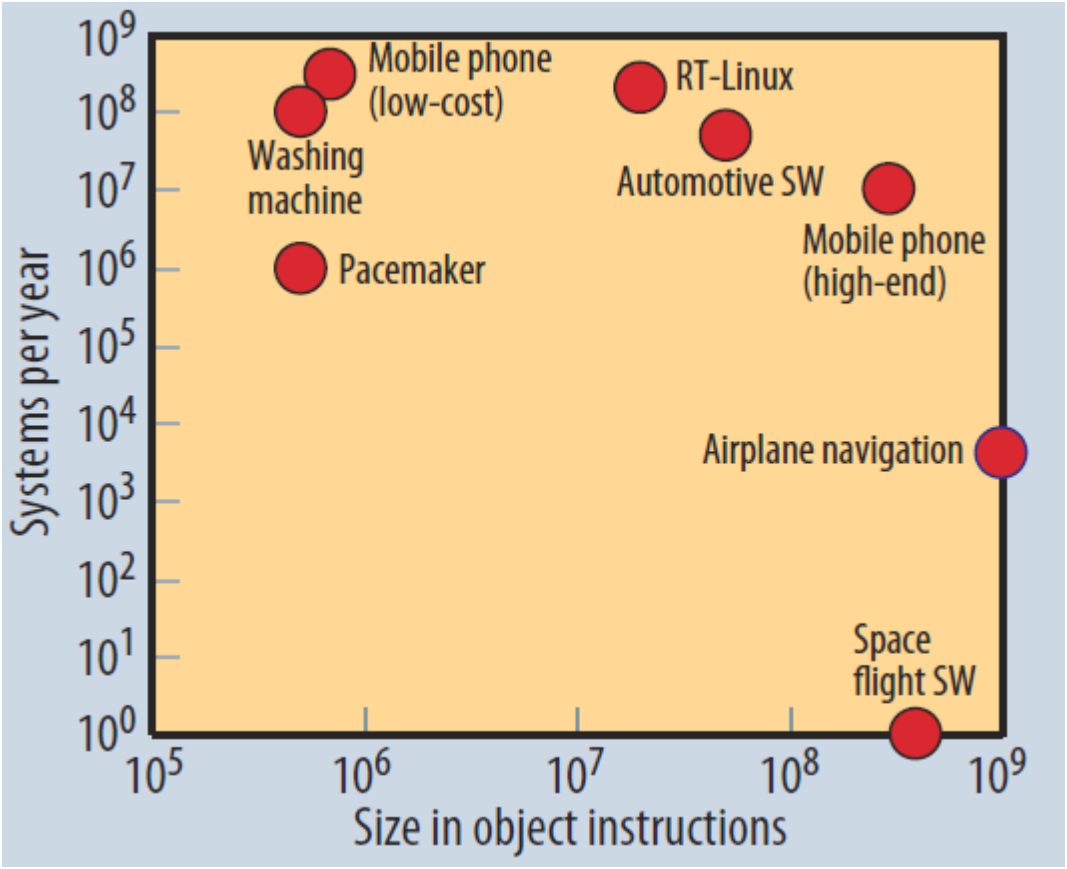

Figure 2. Deployment of embedded software over software size [2].

For Visedo Ltd. it is important to utilize the most sophisticated software development practices in order to cope with the increasing size and complexity of the embedded software in the company's power converter products. For the aforementioned reasons this work attempts to extract suitable development practices associated with the different embedded software product categories shown in Figure 2. One particular embedded software product type of interest is the mobile phone segment. The mobile phone segment has shown an extremely fast rate of development with great variety in products by utilizing modern software development methods.

1.1 PRODUCT VARIATION

Design and manufacturing companies face many challenges in their business activities such as: increasing product functionality, reducing design cycles, decreasing cost and improving quality. Product variation is not without risks, either. How will it affect the quality of the final product or what are the costs of failing to keep up with the quality requirements? There is existing research on design and manufacturing which suggests that variation should be minimized. Also the impact of variation should be systematically reduced. The process of mitigating variation risk is called *Variation Risk Management* (VRM). [3]

1.2 SOFTWARE VARIATION

Software development for embedded systems is closely dependent on the underlying hardware. If the underlying hardware is a member of a product family that consists of variations of a generic product, then it is likely that the software has to be varied also. Assuming that the previous statement is true then the software would also correlate to a family of software products by means of variation.

Contrary to single software product development, software product family development aims at an active reuse of features from a common platform. Claimed benefits for software product family paradigm are reduced development costs and development times. Common terms used by the literature in the area are *Software Product Family* (SPF) and *Software Product Line* (SPL), which are interchangeable terms. [4]

The following description is appropriate for giving a formal definition of a software product family:

"A set of software-intensive systems sharing a common, managed set of features that satisfy 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." [5]

1.3 OBJECTIVES

The primary objective of this master's thesis is to present the best practices for developing embedded software for power converters by latest and most sophisticated methods available. The primary objective can be divided into three sub-objectives which are:

1. Documenting the current state of the Visedo SPL.
2. Identifying the future improvement steps for Visedo SPL development practices.
3. Creating guidelines on how to implement the improvement steps identified in the 2nd sub-objective.

Theoretical concepts, conclusions and results will be presented as equations, tables and figures. And although control algorithms are a very important part of power converter software they will not be covered in this work, control theory being a large topic in its own right. Also detailed electrical characteristics and electrical design principles of power converters are outside the scope of this work, although a brief introduction to electrical engineering concepts will be presented in Appendix 1. The reason for outlining a basic electrical engineering theory is justifying the need for variation of software in embedded systems that are used e.g. to control power electronics components with voltage and current. Controlling a phenomenon with physical properties such as, electrical circuits, with software results in many places that need to be either varied or parameterized by the controller software e.g. switching time of transistors. Also it is good to know some basic electrical engineering theory in an environment where most co-workers have an electrical engineering background to enhance communication within the company.

2 ELECTRICAL CHARACTERISTICS OF A POWER CONVERTER

One way to define a power converter or an inverter is this: a device that is able to convert direct current (DC) into alternating current (AC) or vice versa [6]. A typical example application for power converters is converting DC power to AC power to drive e.g. three-phase electrical motor (or to provide household AC power output). A typical power source can be a wind generator or a solar panel array [7]. If the power converter can operate as a full rectifier bridge then it has the capability to transfer power bi-directionally, i.e. from AC to DC and vice versa, otherwise the functionality is limited to a DC-to-AC power conversion. As an example of power converters only capable of a DC-to-AC output are the low-cost inverters sold at auto parts stores that can be used to convert a 12V DC output to the household compatible 230V AC output.

A very topical application area for power converters has been the control of traction motors and generators for electric drivetrain systems in hybrid vehicles such as cars, buses, wheel loaders and excavators. Typically power converters are used in these applications for converting power in both directions, i.e. from AC-to-DC and DC-to-AC, such as Visedo Ltd.'s the PowerMASTER and the PowerCOMBO products. PowerCOMBO has also the additional capability of doing DC/DC power conversion, e.g. by charging the 24V batteries of a bus or a mobile work machine. [8]

An electric drivetrain typically consists of a selection of following components:

- Inverter
- DC/DC-converter
- Generator
- Traction motor
- Energy storage

The generator can also be used as a traction motor, only the direction of the energy changes. Depending on the topology of the electric drivetrain the number of required inverters may vary: e.g. a wheel loader may have traction motors for each wheel and therefore corresponding number of control inverters is required. Figure 3 presents an example of a Visedo Ltd. drivetrain in a bus application. Generator (G) is placed next to the internal combustion engine (ICE) with the inverter ($\mathrm{I_G}$) used to control it and also to provide interface to the vehicle's DC link. Traction motor (M) is placed between the rear wheel axle along with the corresponding control inverter ($\mathrm{I_M}$).

The roof section of the bus contains the energy storage (SC) units that are connected by a DC link, which are either used to release or store energy based on whether the vehicle is accelerating or decelerating.

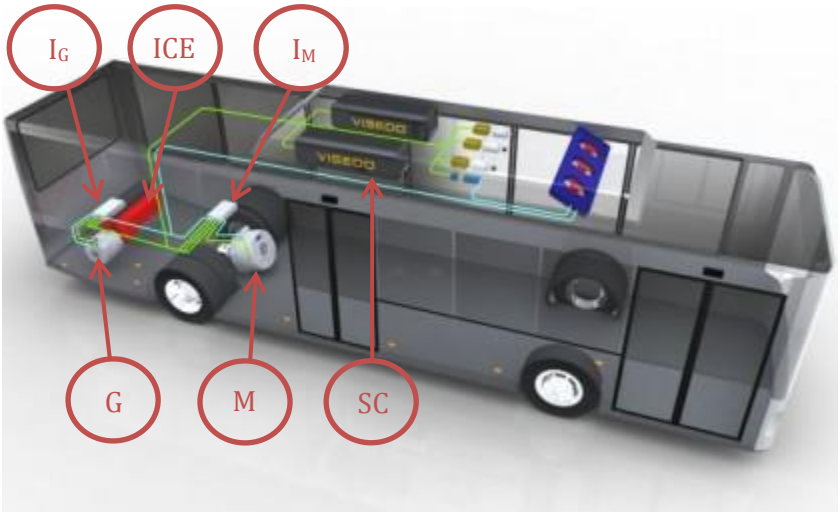

Figure 3. An example topology of an electric drivetrain and its components in a bus application where I_G is the generator control inverter, G the generator, I_M the traction motor control inverter, M the traction motor, SC the energy storage and ICE the internal combustion engine [9].

Table 1 contains a summary of different electric drivetrain components by their type and functionality.

Table 1. Electric drivetrain components and their functionalities.

Component	Functionality
1. Inverter	Converting power from DC to AC e.g. from the energy storage and supplying it to the traction motor for immediate use. Converting power from AC to DC e.g. from the generator and supplying it to the energy storage for later use.
2. DC/DC-converter	Converting high voltage DC link power to low voltage power to maintain the auxiliary functions such as charging the vehicle's batteries.
3. Generator	Generating AC power from the vehicle's internal combustion engine or when the vehicle's brakes are used.
4. Traction motor	Generating acceleration and thus speed from AC power supplied by the inverter.
5. Energy storage	Storing or releasing DC energy on demand through the inverter.

Basic electrical engineering concepts will be presented in the following subsections in order to understand the basic characteristics of an inverter. Different application requirements may require adding, removing or tuning some of these attributes which thus affects the embedded software, i.e. the firmware coupled with the product.

The following hybrid wheel loader topology example introduces the basic components of an electric drivetrain and their relations. Of course different application areas impose different requirements for the electric drivetrain topology, but the following practical application nevertheless introduces the rudimentary components. The number of required components and component types may vary between different application areas but the wheel loader topology depicted in Figure 4 serves as a good practical example of a modern hybrid work machine application.

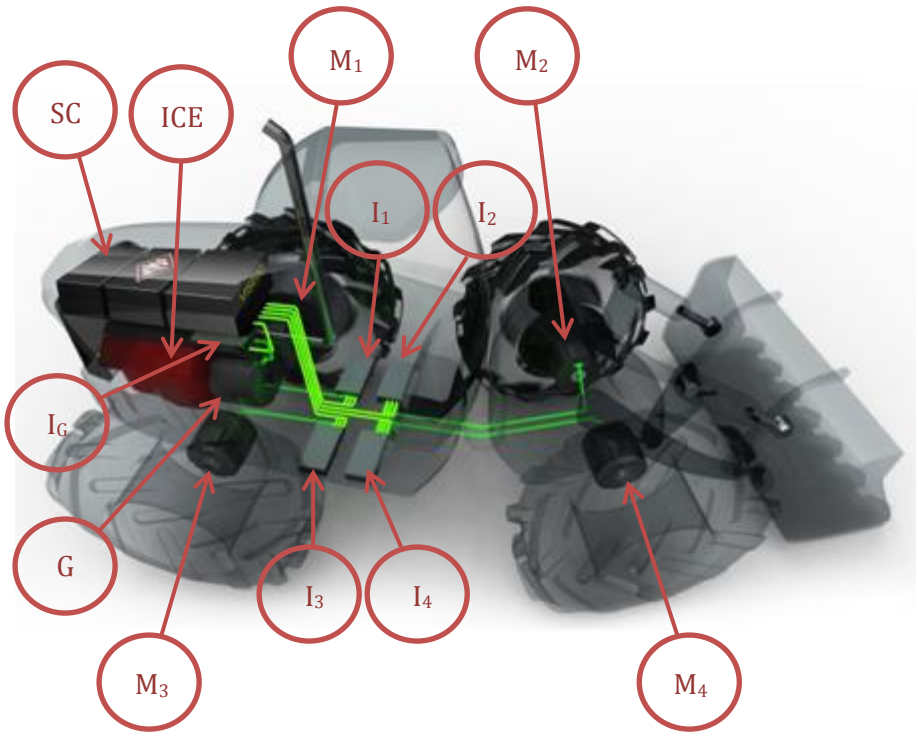

Figure 4. Example of an electric drivetrain topology for a hybrid wheel loader where I_G is the generator control inverter, G the generator, I_i the traction motor control inverters, M_i the traction motors, SC the energy storage and ICE the internal combustion engine [8].

A practical use-case of transferring DC power and energy through the inverters can be described with the example of a hybrid vehicle. Figure 5 depicts a hybrid wheel loader example topology where M_i represents the traction motor and I_i the control inverter of a wheel where $i \in \{1..4\}$. I_G is the control inverter connected to the energy storage, i.e. super capacitor SC. G is the generator and finally the ICE stands for the internal combustion engine.

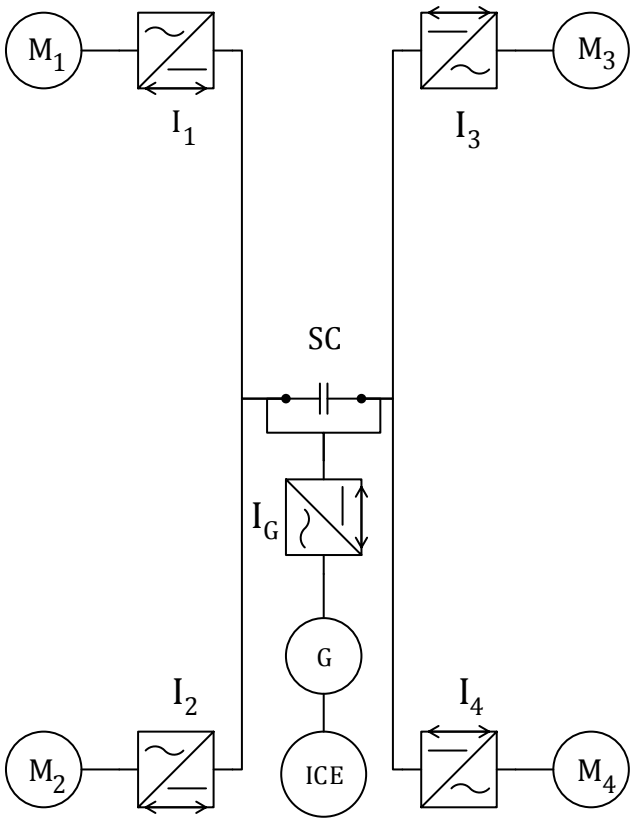

Figure 5. Example of a hybrid wheel loader topology.

Each wheel of the wheel loader has own its traction motor (M_i), which is used to rotate the wheels and thus move the vehicle. For each motor (M_i) there must be a corresponding control inverter (I_i) for converting DC voltage from the DC link into three phased AC voltage to the motor and thus transferring power between motor and DC link. The generator control inverter (I_G) attempts to maintain the DC link voltage within the predefined operating range by converting the three-phased AC voltage generated by the generator (G) into DC voltage. The generator (G) converts mechanical rotation into three-phased AC voltage, whereby the rotation of the generator is provided by the internal combustion engine (ICE).

During peak loads when more power and energy is required than the DC link is able to provide by the inverter (I_G) and generator (G), super capacitor (SC) can be used to boost (assist) the DC link to generate brief peak power and energy loads and therefore maintaining a DC link voltage level within the predefined operating range. If the inverter (I_G) is able to convert and transfer more power from generator (G) than it is required for maintaining DC link voltage level within the predefined operating range, it can be stored in the super capacitor for later use.

It is also possible to recover energy from the vehicle's traction motors (M_i) when the vehicle's brakes are used, with motors acting as generators for a brief period of time. Also the main function of the generator (G) is also reversible, the generator acting as a motor for a brief period of time and assisting the internal combustion engine (ICE). The two previously covered use-cases illustrate the diversity and scale of electric drivetrain applications. Diversity and scale come from the physical properties and load requirements of the vehicle. Moving different types of vehicles with dynamically changing mass in time within specified velocity range impose different requirements for dimensioning the electric drivetrain components such as the electrical power conversion capabilities of the power converters. Since the electrical drivetrain components must be dimensioned according to the application requirements it also imposes requirements for the embedded software of the power converters. For this work different application requirements mean exploring solutions on following items:

1. How to manage the diversity of embedded software for a range of power converter products with varying application areas and electrical characteristics?
2. How to organize the development of the software product line for a range of power converter products?
3. How to structure the software in terms of architecture styles and development practices available?

3 SOFTWARE ARCHITECTURES

Software architecture has been defined by many people over the years and there does not seem to be a single universally accepted definition for it [10]. Depending on the viewpoint, software architecture has bibliographical, classical, communal and modern definitions [11]. However the author of this master's thesis prefers the following two definitions, because they are concise. The first one is a more formal one, which is as follows:

"The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." [12].

And the second one presents a more practical view on the matter:

"The highest-level breakdown of a system into its parts; the decisions that are hard to change; there are multiple architectures in a system; what is architecturally significant can change over a system's lifetime; and, in the end, architecture boils down to whatever the important stuff is." [10].

The first definition is interesting, in a sense that it implies that the usage of components is a means of building architecture in a predefined way. This goal is achieved by the collaboration of different components for meeting business and technical requirements. Also the reference to externally visible properties gives a guideline into designing the components which is that a change in the internals of the component should not be visible to the public interface, in other words, this denotes as a type of data encapsulation.

The second definition has a common sense echo to it, namely that architecture is associated with the production costs, from the very beginning to the end. Getting architecture wrong can lead to a worst case scenario: a complete failure or, at least costing significant amount of time and resources. On the other hand, if the architecture is good and flexible enough, important features can be added and obsolete features can be removed during the lifespan of the product.

The goal of software architecture is to recognize the requirements and use-cases that affect the structure of the software. Often the requirements of different stakeholders (e.g. business, technical, user or functional) may be conflicting and therefore trade-offs will occur. However, architecture should address the following key points:

- Hiding the implementation details of individual components, but expose the general structure of the system.
- Reflecting all the use-case scenarios of the system.
- Addressing the requirements of different stakeholders.
- Meeting functional and quality requirements.

Keeping the previous points in mind one could say that the focus of architecture is on the big picture i.e. how components interact. Algorithm selection and implementation details of a component are design issues, although sometimes these may overlap with architectural concerns. Also software architecture should be built to evolve rather than built to last. [13]

On a personal note, the author finds the built-to-evolve philosophy is somehow reassuring as it has been his personal experience within the industry that getting things right from the beginning is difficult. The aforementioned experience has led to the notion: whatever you do, don't paint yourself into a corner. In other words, always leave some room to manoeuvre within the design.

Software architecture patterns and styles have emerged as a solution to common software architecture design problems. There are two commonly used terms for generic architecture design solutions where the first one is called *architecture patterns* and the second one *architecture styles*. There is a slight distinction between the two terms according to the Software Engineering Institute of the Carnegie Mellon University [11], which can be characterized as follows:

- Architecture pattern: *"A description of element and relation types together with a set of constraints on how they are used."* [11].
- Architecture style: *"A specialization of element and relation types, together with a set of constraints on how they can be used."* [11].

Some of the literature uses these terms and definitions interchangeably; however, the term architecture style will be preferred within the context of this work [14, 15].

3.1 ARCHITECTURE STYLES

Software architecture styles are analogous with conventional design patterns where the difference is the scope. Design patterns, such as for object-oriented programming, are reusable design solutions for the component level [16]. By raising the abstraction level to the next architecture styles i.e. reusable architecture designs, will emerge. Software industry and community at large have developed multiple architecture styles for different types of application such as: database, communication, desktop, or distributed. It is not always evident which architecture style should be chosen; this is where sufficient analysis of high level properties and interactions of the system are important [17].

Table 2 contains a summary of different software architecture styles; the table is not by any means comprehensive, but rather it attempts to capture those that are often used and thus a person is likely to come by if the person's work involves developing software. Two of the architecture styles, very commonly used in embedded systems software such as mobile phones will be explored in further detail in the following sub-chapters. The first architecture style is the layered architecture style and the second one is the component-based. The aforementioned architecture styles were chosen for more detailed review since power converter software typically consists of different communication interfaces, control algorithms and user application domains, thus resembling layered architecture. All architecture styles have a specific focus area by which they can be categorised. Coincidentally the focus area of both layered and component-based architectures is structure [14].

Table 2. Software architecture styles [14].

Architecture style	Description
1. Client/Server	Architecture style for distributed systems. The system consists of separate client and server applications. Example: a client application for displaying data requested from a database server.
2. Component-Based Architecture	Focuses on the design and decomposition of well-defined functional and logical components that are reusable. Components consist of communication interfaces comprised of functions, events and properties. Example: MeeGo software platform.
3. Domain Driven Design	Object-oriented design approach for building software for the underlying business domain. Focuses on identifying system's elements and behaviours while identifying relationships between them. Example: online publishing platforms.
4. Layered Architecture	Focuses in structuring related functionality into distinct layers where each layer has a predefined responsibility of the system. Layers are stacked on top of each other. Examples: Open Systems Interconnection (OSI) Reference Model, MeeGo and Tizen software platforms.
5. Message Bus	Software system where applications are able to send messages to each other without knowing details of each other. Communication is done through one or more channels. Example: D-Bus communication of GNOME- and KDE-desktop environments.
6. N-Tier/3-Tier	Architecture style similar to layered style, but in which each layer can be deployed on different computers. Typically at least three separate layers with their own responsibilities. Example: Financial application consisting of presentation, business logic and data access layers.
7. Object-Oriented	A design paradigm where data and routines for manipulating the related data are encapsulated into self-contained and reusable objects. Objects are loosely coupled, independent and communicate by sending and receiving messages. Examples: C++ and Java class library frameworks.
8. Service-Oriented-Architecture (SOA)	System is comprised of a set of loosely coupled, autonomous and distributable services. Examples: Reservation systems and online stores.

From the software architecture styles listed in Table 2 the layered- and component-based architecture styles will be covered in more detail to explore Visedo SPL's future development possibilities.

3.2 LAYERED ARCHITECTURE STYLE

Layered software architecture style is one of the most traditional ways of designing and organizing software architecture [18]. It is still used by many software systems such as the now defunct MeeGo software platform and its successor Tizen software platform for mobile devices [18, 19, 20]. Figure 6 depicts the layered structure of the MeeGo software platform and Figure 7 its close relative the Tizen software platform, both organized into layers.

Mobile devices have evolved from embedded systems software and thus the layered architecture style can also be considered a practical solution for embedded systems software. Embedded systems, and hence, layered software architectures, are still used today in many industrial applications [18]. A typical example of an embedded system software application from process industry could be controlling the pumps used to regulate the flow of liquids in a process. Another prime example of an embedded system software usage is controlling electrical motors and generators, which can be accomplished with an inverter. The aforementioned use-cases in turn can be realized with layered software architecture design.

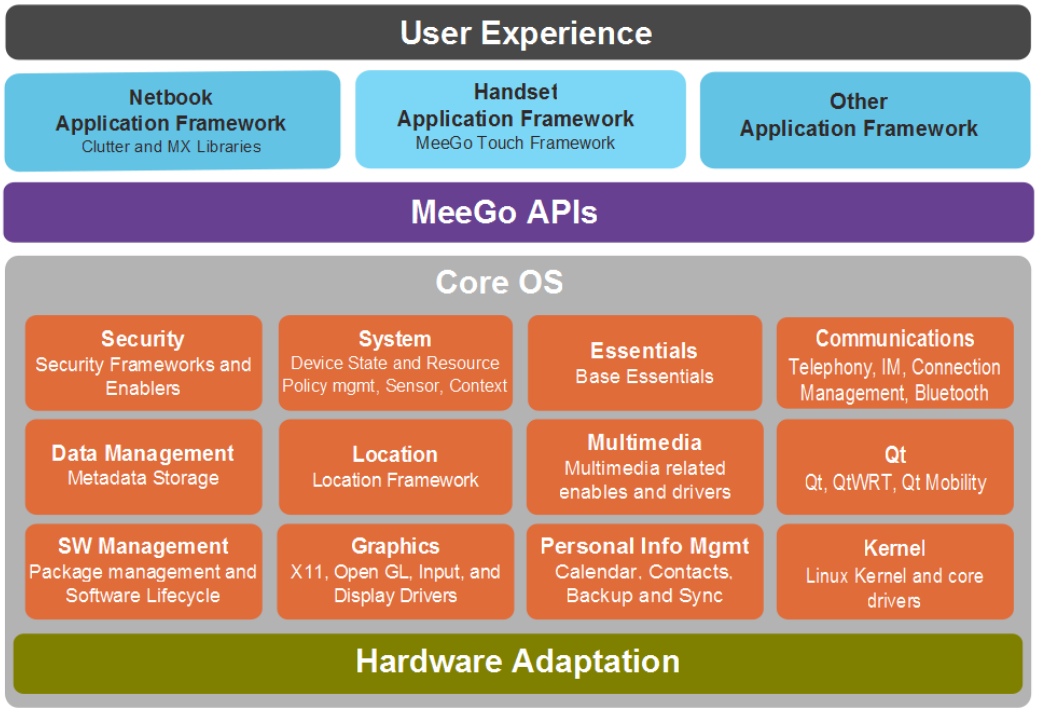

Figure 6. Layered software architecture of the MeeGo software platform [19].

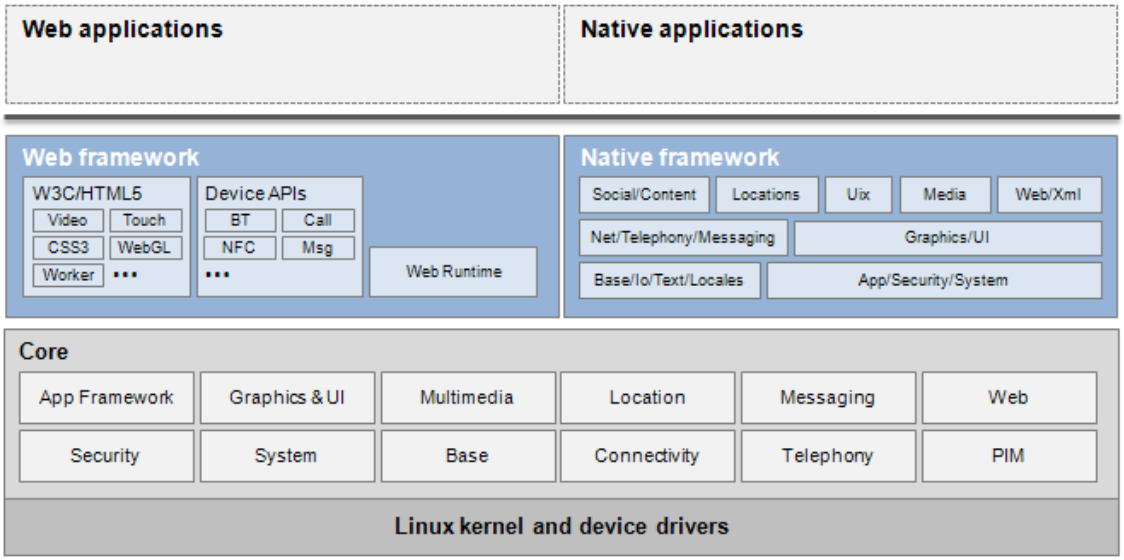

Figure 7. Layered software architecture of the Tizen software platform [20].

In layered software architecture, by the strict interpretation, each layer is responsible for providing predefined services to the layer immediately above it [14]. However, it should be pointed out that in the OSI Reference Model, for example, entities within the same layer can communicate with each other, with the layer immediately below them and notify layers immediately above them [21]. If two systems are connected to each

other via a physical medium then the layers on the same level, called peer layers, can communicate with each other [21]. For the purposes of this master's thesis the OSI Reference Model definition for the layer interaction will be assumed.

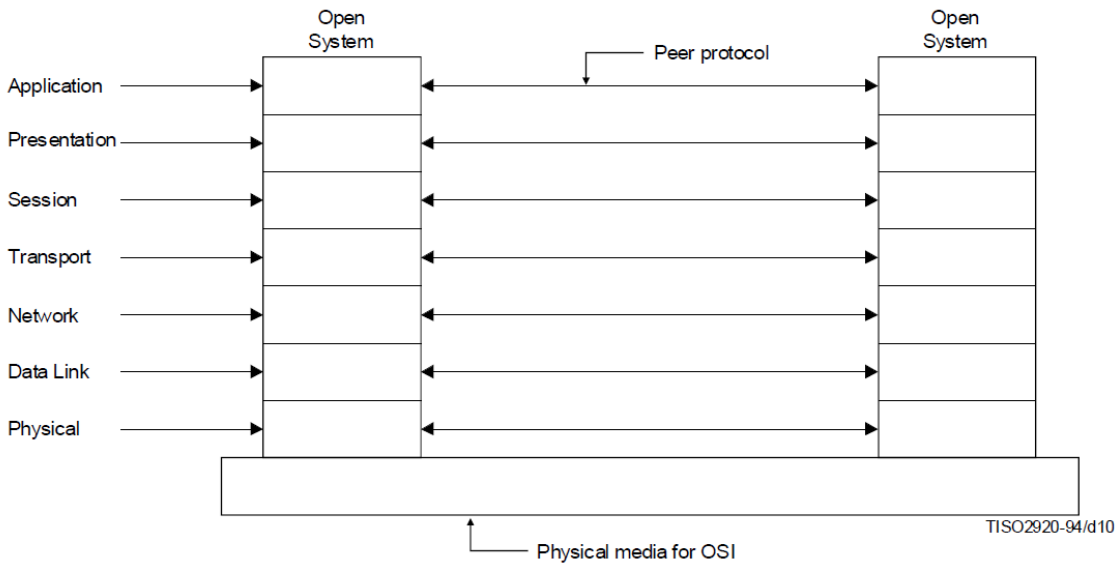

Figure 8. OSI Reference Model [21].

There are several commonly used principles regarding layered architecture designs which can be also regarded as the goals for building a sound layered architecture. Table 3 summarizes these design principles.

Table 3. Layered architecture design principles [14].

Principle	Description
1. Abstraction	View or the landscape of the software, as a whole, is abstracted into a level of detail where the responsibilities and relationships of individual layers can be understood.
2. Encapsulation	During the design phase no assumptions need to be made about functions, data types and properties as these features are not exposed to the layer boundaries.
3. Responsibilities of the functional layers	Functional responsibilities between layers are clearly separated. Example: a layer can use the services of the layer immediately below it and can only provide notifications to the layer immediately above it i.e. cannot use its services directly.
4. High cohesion	Given the clearly defined responsibilities for each layer, cohesion can be maximized within the layers.
5. Reusable	Lower layers are not dependent on the upper layers, which promotes reusability of the lower layers in other products.
6. Loose coupling	Layers communicate via abstractions or notifications and thus a loose coupling between them is enabled.

While the layered architecture style is the one of earliest styles to be utilized, and still used to design new software systems, it does have its benefits, which have not entirely gone out of fashion. Partly the reason for its continuing use is that many libraries developed in the past (such as: mathematical, algorithm and standard libraries) are still valid and reusable. This is especially true for embedded systems software written in C programming language. It is therefore appropriate to summarize the key benefits of layered architecture style, which are presented in Table 4.

Table 4. Benefits of layered software architecture [14].

Benefit	Description
1. Abstraction	Level of abstraction can be adjusted at design level for each layer.
2. Isolation	Impact of technology or implementation updates on the individual layers is isolated and thus the overall risk to the system is minimised.
3. Manageability	Code is organised into manageable sections via separation of concerns and associated dependencies.
4. Performance	Layers can be distributed over several computing nodes if required and thus scalability, robustness and performance can be improved.
5. Reusability	Reusability is obtained by clearly defined responsibilities and dependencies.
6. Testability	Well-defined layer interfaces enable better testability. Different versions of the layer interface can be also tested.

3.3 COMPONENT-BASED ARCHITECTURE STYLE

In a component-based architecture style the system design is decomposed into individual functional or logical components [14]. A component is a unit consisting of well-defined communication interfaces, functions, events and properties [14]. Larger and more complex components can be built by composing and combining the properties and functionalities of different components. Component composition provides a higher level of abstraction than individual modules or objects [14]. A good example of component-based architecture is the MeeGo software platform. Figure 9 presents the domain architecture view of the MeeGo software platform where components are grouped into domains. A domain is group of related components that are responsible for providing the service definition of the domain.

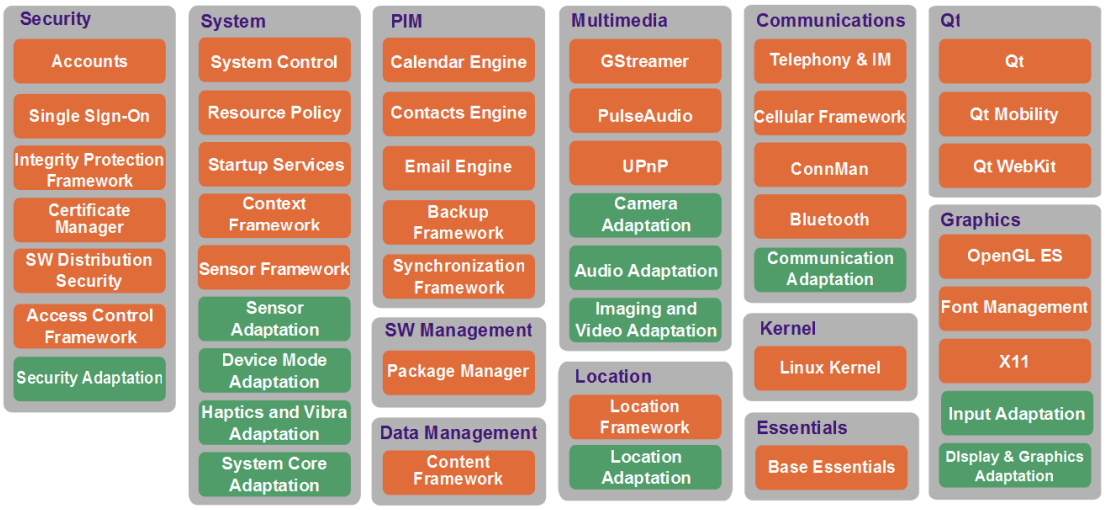

Figure 9. Component domain architecture view of the MeeGo software platform [22].

Typical examples of components are user interface (UI) components such as buttons, grids and views. For example, a button has properties that can be used to parameterize the appearance of the button, events to which it can react to, such as being clicked on. As an example of embedded system software component, a scheduler component could be constructed, where the scheduler period is parameterized with a property and events from peripherals are received from notifications. As with layered architecture style, component-based architecture style has also multiple design principles. The main design principles behind component-based architecture style are listed in Table 5.

Table 5. Component-based architecture design principles [14].

Principle	Description
1. Reusable	Components should be designed to be reusable in different scenarios and applications.
2. Replaceable	Components should be designed to be replaceable by other components with equivalent functionality.
3. Independent of the context	Components should be designed to be independent of the context and environment they are used. For example, state information should be supplied from the outside and not have a dependency from the inside.
4. Extensible	Existing components can be used to introduce new behaviour to a component. This design principle conflicts with the independency principle.
5. Encapsulated	Components should not expose internal processes, functions or state. Caller is only allowed to use component's well-defined public interface.
6. Independent	Dependencies with other components should be minimized and thus make them easier to be deployed. This design principle conflicts with the extensibility principle.

Two of the design principles, i.e. extensibility and independency, have somewhat conflicting goals and thus actual practice often requires striking a balance between them. There are still many technologies using the component-based architecture approach such as Component Object Model (COM) and Distributed Component Object Model (DCOM) technologies by Microsoft and the Enterprise JavaBeans (EJB) technology by Oracle. Given the previously presented technologies there must be obvious benefits for choosing component-based architecture by two established software development companies like Microsoft and Oracle. Therefore Table 6 contains a summary of benefits for component-based architecture style.

Table 6. Benefits of the component-based architecture [14].

Benefit	Description
1. Ease of deployment	New versions of the component can be deployed in place of the existing one, provided that they are binary compatible, with no impact to the system as a whole.
2. Reduced cost	Cost of development and maintenance can be spread out if third-party components are used.
3. Ease of development	Components provide their functionality via clearly defined interfaces and thus help minimizing the impact to the rest of the system.
4. Reusable	Cost of component development can be amortized across several systems.
5. Mitigation of technical complexity	Complexity is mitigated through component containers and associated services, e.g. component lifetime management service.

3.4 COMBINING ARCHITECTURE STYLES

There are many factors to consider when choosing the architecture style. These factors include: resources of the organization for design and implementation, experience of the developers and environmental constraints such as infrastructure [14]. Based on the technical requirements and organizational constraints, suitable software architecture style should be chosen. Often it might be that any single software architecture style doesn't fulfil all the technical and organizational constraints and a combination of software architecture styles must be chosen instead. As an example of the organizational constraint could be the MeeGo software platform's different domain components (Figure 9) developed by teams specialising in particular domains. Analogously there could be a technical constraint that certain components are on a specific layer of the architecture to provide services for other components; a prime example of this is the kernel component of MeeGo software platform in Figure 6. One should also note that the component domain architecture view (Figure 9) should not be taken as an indication of the physical layout of the components. In other words, a component within the domain may be dependent on other components of the same domain or the layer immediately below. For the aforementioned reason component domain architecture view (Figure 9) should be cross-referenced with the layered architecture view (Figure 6).

4 SOFTWARE PRODUCT LINES

Formal definition for *Software Product Line* was given in Chapter 1.2. The objective of this chapter is to explore further what software product lines are, what their benefits are, what they compose of and how they are related to software variation. Software product line engineering (SPLE), as a practice, involves collaboration of many disciplines, such as software engineering, technical- and organizational management. This section will focus on engineering and technical aspects of software product lines and thus organizational management practices will mostly be omitted from the scope of this thesis. Existing research in the field of SPL uses few basic terms and some of them have alternatives with equivalent meaning. Table 7 summarizes the most important terms for the reader.

Table 7. Summary of SPL basic terminology [5].

Term	Alternative term	Description
Software Product Line	Software Product Family	A set of related products.
Core assets	Platform	Basic building blocks of SPL such as: requirements, documents, software and processes.
Software Product Line Engineering	-	Systematic way of reusing core assets to build products.
Domain	-	Area of expertise.

Product lines are about building a collection of related products from a pool of generic components applicable in each product's context; this generic approach can also be applied to software and thus a product line of related software products can be constructed [5, 23, 24, 25]. Being able to manage commonalities of different products by using shared assets but at the same time to address the subtle differences between related products by means of variation is a tempting notion for companies seeking increased efficiency [5, 24]. The aforementioned development approach provides a way of developing and producing products in a more economical way, provided that attached activities and practices are sufficiently executed, be it core asset development, product development or management activity [26]. Rewards for properly executed and implemented SPLE practices are: reduced development costs, shorter time to market and systematic reuse of software [27, 28]. On the other hand, when used in single system development or merely component-based development, SPLE

will not offer the full benefits of SPLs, either, since previously developed assets are not utilised in an organised and efficient manner [5].

When an organization starts to develop a SPL they should define the scope of the product line. Organization may wish to target a broad market area where each product is aimed at specific market segment. Therefore the scope of the SPL should be carefully chosen to accommodate the range of possible variations supported by the core assets of the SPL. It is beneficial for a company to choose the scope of the SPL and target market segments according to the interests of the company. A company may also choose to develop custom assets to expand product diversity and therefore target new market segments provided that it is in the company's interests. As a consequence, out-of-scope variation can be achieved through custom assets. Figure 10 illustrates the aforementioned concepts as three different subsets within the space of all the possible products [24].

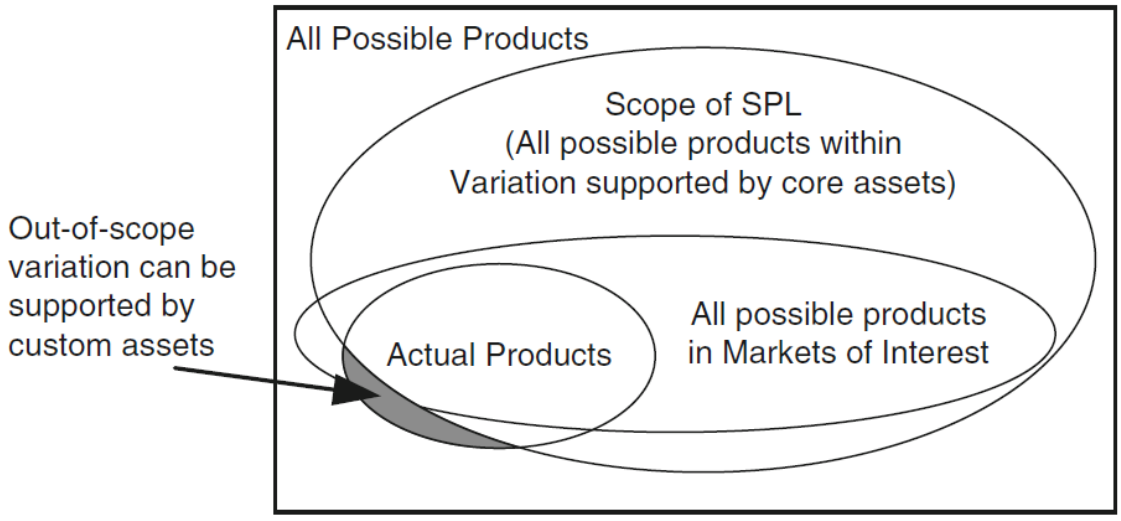

Figure 10. Scope of the software product line [24].

Product line development and subsequently SPLE practices are especially beneficial to products containing embedded systems software. There are many examples of products containing embedded systems software, such as mobile phones, car engine controller units (ECU) and car parking assistant units [29, 30, 31].

All previously mentioned products have many differentiating features: examples of such features include:

- For mobile phones, type and size of the screen, amount of internal storage memory.
- For ECUs, type of the engine (diesel vs. gasoline), engine performance profile (economic vs. sport).
- For parking assistant units, assisted parking (supported vs. not supported).

4.1 BENEFITS AND ACTIVITIES

Modern software engineering has striven to develop methods for increasing software reuse, improving quality, managing complexity, reducing lead times and production costs [27, 32], all of which are essential for any organization. Even more importantly seeking this kind of competitive advantage is important to small companies (like Visedo Ltd.) which are attempting to enter into specific market segments with their products.

There are many motives for different organizations to adopt SPL paradigm. A global electronics company (Philips) has identified in its research four main motives which are as follows:

1. Management of size and complexity.
2. Maintenance and pursuit of high quality.
3. Management of product diversity.
4. Reductions in product development lead times.

These four motives are shown on the left hand side in Figure 11. Interestingly the three solutions shown in the middle of Figure 11 form together the solution for SPL. Philips manufactures a wide range of products from consumer electronics to healthcare equipment and thus managing complexity while maintaining high quality with reduced lead time makes sense from the business perspective.

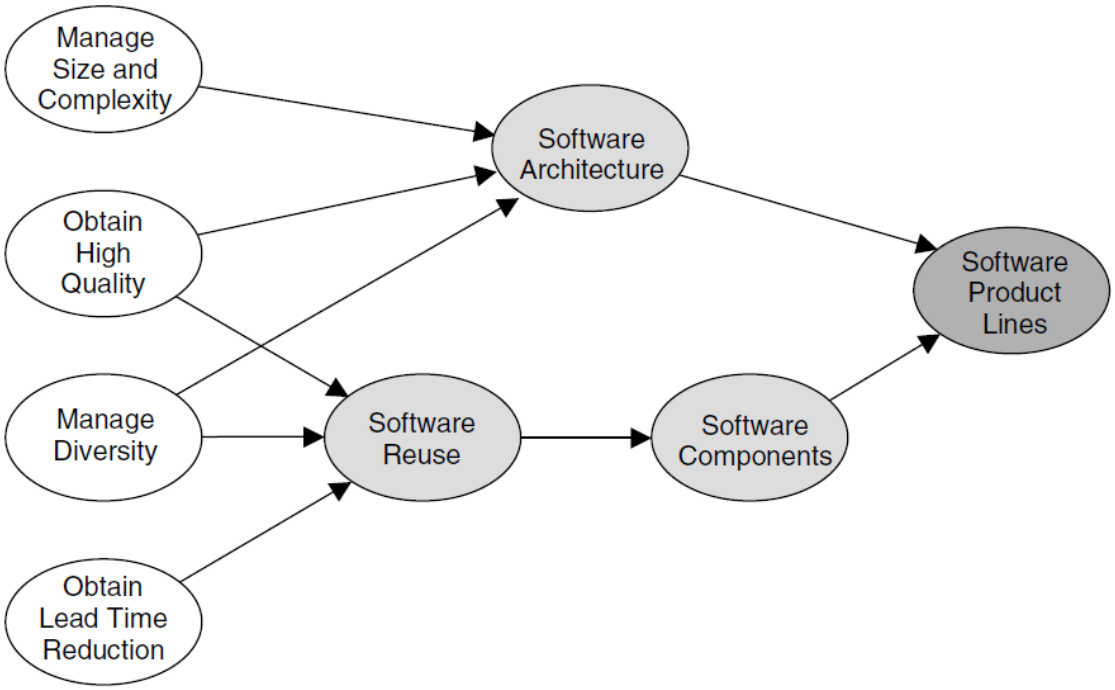

Figure 11. Four main motives identified by Philips for adopting SPL paradigm [32].

Now, when a company develops a new product using SPL paradigm, existing assets can be utilized instead of developing single-system software for each new product. This is an important factor especially for embedded systems product development given the trend of increase in size and complexity in embedded systems software (Figure 1).

There are many types of reusable assets associated with SPLE, the most important prerequisite being that there is a centralised repository of core assets from which products can be derived from. Of course, there are costs associated with building a core asset repository which may not be feasible for a single product, but if the costs can be amortised over many products, considerable expenditure of effort can be avoided. Reusable asset types and associated benefits can be formalised and they are presented in Table 8.

Table 8. Reusable asset types and associated benefits [27].

Reusable asset	Associated benefits
1. Requirements	Common requirements for product line where as product requirements are deltas to common.
2. Architecture	Same architecture can be used for each product line product. Considerable effort saved by not having to do multiple times provided that the architecture is correct.
3. Components	Reused within the product family. Differentiation e.g. via parameterization or component versioning.
4. Modelling and analysis	For example performance, resource allocation and timing analysis models can be reused which builds confidence that aforementioned issues are solved for each product.
5. Testing	Test cases, plans, resources, data and processes in place for each product, though some product specific customization may be required.
6. Planning	Baselines for production plans, budgets and schedules from previous product line product development projects can be utilized.
7. Processes	Tools, procedures and processes for management and development teams are in place. Also experience in all aforementioned practices.
8. People	Fewer people are required overall. Also people can be moved around within the product line with less effort.

The list of reusable assets and associated benefits can be very tempting, but depending on the context one should carefully consider the costs associated with SPL development. For instance if the products produced by the organization are not sufficiently related then launching a software product line is questionable.

Even if the organization does produce related products, adopting SPL development paradigm is a business decision [27]. Business goals should be carefully weighed against the achievable asset type benefits and associated costs. Constructing a SPL requires a substantial start-up investment as well as the ongoing maintenance costs of core assets [27]. Given the initial start-up costs and different organizational contexts it is also reasonable to summarize the associated costs for each asset type, which are presented in Table 9. Obviously the benefits of asset types should outweigh the costs.

Table 9. Asset types and associated costs [27].

Asset type	Associated costs
1. Requirements	Recognizing requirements for a group of systems may require laborious analysis and negotiations to agree upon the common requirements and variation points for each system.
2. Architecture	Product line wide variation support imposes additional constraints to architecture and thus requires greater talent to define.
3. Components	Components should be designed to be generic without noticeable impact on performance. Also components must be designed to be robust and extensible to accommodate a range of products within the product line. Variation points must be placed or at least anticipated.
4. Modelling and analysis	Additional constraints may be imposed on moving and synchronization of existing processes to different processors. Additional processes may also need to be created.
5. Testing	Test artefacts must be robust and extensible to accommodate different products and associated variation points.
6. Processes	Process definitions, tools and procedures must accommodate unique product line needs. Also managing differences between managing a product line and managing a single product should be taken into account.
7. People	Existing personnel must be trained to understand software product line practices in addition to general software engineering and corporate procedures. Training material needs to be created to address the product line practices.

Table 9 summarized the associated costs for the subset of the benefits. Looking through the associated costs one might also come to the conclusion that adopting product line practices require many things from different units of the organization and thus it may not be suitable for every organization [27]. However there are many companies that have been able make the transition to software product line practices and able to reap the benefits [27, 29]. For a new company the associated costs listed in Table 9 can be considerable and therefore investing in a subset of the reusable asset development can be a viable option.

4.2 ESSENTIAL ACTIVITIES

Once an organization has committed itself to SPL paradigm it should also be prepare to adopt the three essential activities of SPL. The three activities are: core asset development, product development and management [26]. The activities are closely related to each other. Core asset development activity provides assets (e.g. components) for product development activity which is responsible for assembling a product out of the core assets [26]. Management is responsible for supervising, providing needed resources and coordinating the two other activities [26].

Core asset development, as a term, is interchangeable with domain engineering, which as an activity consists of the development of common features and identification of the variation points [26, 33]. Product development activity has also alternative terms such as product or application engineering [26, 33]. It is the responsibility of product development activity, besides building the concrete product, to integrate commonalities into a product and manage the corresponding variation points [33].

The aforementioned activities are highly iterative and interlinked; thus forming feedback loops to the other activities [26]. The iterative and interlinked nature of the activities is presented in Figure 12. Properties of the core asset development (Chapter 4.2.1) and the product development (Chapter 4.2.2) activities will be explored in further detail in their respective subsections. Management activity will mostly be outside the scope of this work and thus it will only be referred when appropriate from the point of view of the other activities. For a reader interested in the management activity aspects of SPL's the chapter *Technical Management Areas* in *Software Product Lines* by Clements et al. can serve as a good starting point to the subject.

Figure 12. Three essential activities of software product line paradigm are: core asset development, product development and management [26].

4.2.1 CORE ASSET DEVELOPMENT

As previously stated the first and foremost goal of core asset development activity is to establish a production capability for products [26]. Core assets can be either extracted from existing products, thus establishing a baseline for subsequent core assets, or developed from the ground up [26]. Given the iterative nature of core assert development activity, one can reason that each iterative step requires some form of input to produce the desired output. Actually, examples of input and output have already been given, i.e. extracting core assets from existing products (input) to produce a product (output). Of course the aforementioned input example is not the only one and therefore a more comprehensive list of possible inputs is given in Table 10 with associated descriptions.

Table 10. Core asset development activity inputs [26].

Input	Description
1. Product constraints	Determining the commonalities and variations of the products within the product line. Identification of behavioural, technological, quality and performance constraints.
2. Styles, patterns and frameworks	Identification of required architecture styles and design patterns. Architecture styles prescribe how components (subsystems) interact i.e. affect the coarse-grain design of the SPL, whereas design patterns prescribe how individual components are built, i.e. affect the fine-grained design of individual core assets in the SPL.
3. Production constraints	Identification of military, commercial or company-specific standards that apply to the products in the SPL. Aforementioned aspects may have considerable impact on e.g. time-to-market or time-to-initial-operating-capability.
4. Production strategy	Determining whether to build from top-down (starting with core assets to construct products) or from bottom-up (extracting core assets from existing products). Production costs of generic components versus purchasing from open markets.
5. Inventory of pre-existing assets	Determining whether existing or legacy systems can be mined for assets and thus proven concepts and structures be borrowed. An inventory of libraries, frameworks, algorithms, tools and components that could be utilized. Also what kind of technical management, funding models and training could be adapted to the SPL.

Inputs should be regarded as prerequisites for creating the outputs. Prior to each iteration step inputs may change and thus affect the output of the iteration. As with the inputs, a list of possible outputs with associated descriptions is given in Table 11.

Table 11. Core asset development activity outputs [26].

Output	Description
1. Product line scope	Description of the commonalities of all products and the differences between them. Alternatively a list of products that constitute the product line within the defined scope.
2. Core assets	Foundation for building the actual products. May include common software architecture, components, test plans, test cases, design documentation, requirements and domain models.
3. Production plan	Description of how products are built from core assets. Each core asset has an attached process that describes how it should be utilized in the product development.

4.2.2 PRODUCT DEVELOPMENT

The goal of product development activity is to build products out of core assets; however, mature product line organizations actually emphasize the product line over individual products, despite the ultimate goal of producing products [26]. Nurturing product line and its core assets carefully is a long-term investment when compared to short-term investment of producing a single system product.

As previously with core asset development activity, product development activity has also dependencies on other essential activities. Table 11 presents the three outputs of core asset development that are inputs on the product development activity and thus will not be presented as a table of product development inputs again. Additional input outside of the core asset development outputs are the requirements of the individual products [26]. One thing that a product line organization should also be aware is that in case a previously unrecognized commonality is found among products in the product line then core assets may have to be updated accordingly [26].

Now revisiting the ultimate goal of product development activity, which was the building of products out of the core assets. The definition can now be extended as follows: given the individual product requirements in the product line scope product is developed out of core assets with the help of corresponding production plans [26]. Subsequently the output of product development activity is the concrete product.

4.3 MANAGING DIVERSITY

An essential part of the software product line development paradigm is the management of diversity among products within the product line, which is a process also known as either variability or variation management [24, 25, 32, 34]. The major difference between conventional software engineering and software product line engineering is the way how variation is implemented [34].

Traditionally software variation has been dealt with over time, i.e. as the software has evolved, which has been also been referred to as configuration management. However, variation in software product development engineering is multidimensional where variation is done in both space and time. Variation in time, in the context of software product lines, refers to the traditional configuration management whereas variation in space refers to the differences between different products of the product line. [34]

There are a few common strategies for implementing variation within product line context. Initially product line research focused on defining a variant-free architecture based on the analysis of commonalities and differences between products [32, 35]. Based on the definition of variant-free architecture, variation can be introduced via variation points [23, 24, 25, 32]. A second plausible strategy is to employ a component composition scheme [32, 34]. In a component composition scheme diversity between products is implemented by composing different combinations of components [32, 34]. However, aforementioned strategies are not necessarily used in their purest form to achieve the most flexible outcome and therefore the concept of combining variation and composition has been introduced [32, 34].

Figure 13 depicts how the focus of research within software engineering community has evolved over time from individual products to entire product populations. Initially there were individual software products and then composition by software components. A prime example of the software component composition paradigm is the ActiveX technology by Microsoft where rapid application development (RAD) tools, such as Visual Basic, were used to add the glue logic to compose the desired application functionality [32]. When research focus shifted to product families and the analysis of commonalities between different products, software variation concept was introduced to cope with diversity and functionality between products [25, 32]. Ultimately, when software component composition and variation elements are carefully combined, the most evolved concept of product populations will be available [32]. Another way to

characterize the combined approach is that different variations of files can be composed to produce variations of components and variations of components can be composed to produce variations of products [34]. As shown in Figure 13, the concepts of product family and product population are encompassed within the software product line paradigm. Consequently the main tools for implementing software product line paradigm are variation and composition [32, 34].

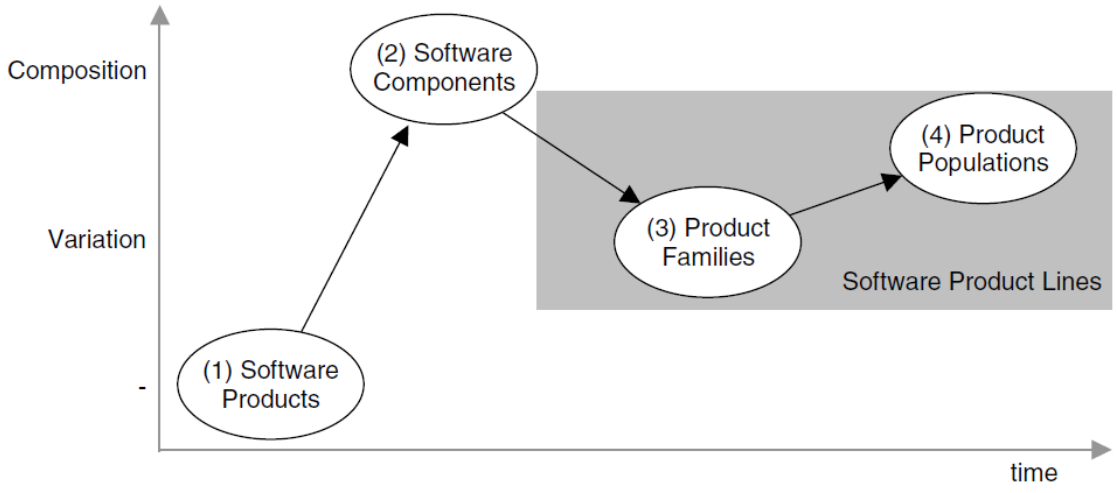

Figure 13. Evolution of software engineering research focuses over time [32].

4.3.1 VARIATION

Traditionally variation has concentrated on commonalities rather than differences in functionality between the products of a product family that form the actual software product line [32]. For the aforementioned reason variation points have been used to represent the differences between products by establishing a variant-free architecture [32]. A variation point is a place in design or implementation that identifies the location where variation should occur within bounds of the particular use-case [25, 36].

Variant-free architecture can be defined as a generic software architecture style for a particular product line. The variant-free architecture style attempts to be a fully described architecture where differences between products are not considered to be architecturally important. Differences between products are considered to be issues of design and implementation instead of an architectural one. One of the benefits of variant-free architecture is that it is suitable for a product line which must support a wide range of options related to a particular aspect. An example case of the aforementioned product line is a family of inverter/converter products where the underlying hardware platform is not visible to the upper architectural levels. By having a good abstraction between hardware and software platforms new products can be

derived with relative ease from the product line architecture. As a consequence implementing differences between products becomes a task of defining and implementing product specific parts into the design and construction of the product [37].

Figure 14 presents a graphic example of a variant-free product line architecture. The variant-free skeleton defines a fixed structure of units to implement the common parts of the product family. Selectable plugins, such as individual modules or compilation units, are used to implement the diversity between products. The former holds true on small scale and thus can be considered as variation, but on larger scale if the plugins are shared between multiple product family members, then signs of composition are beginning to appear. [32]

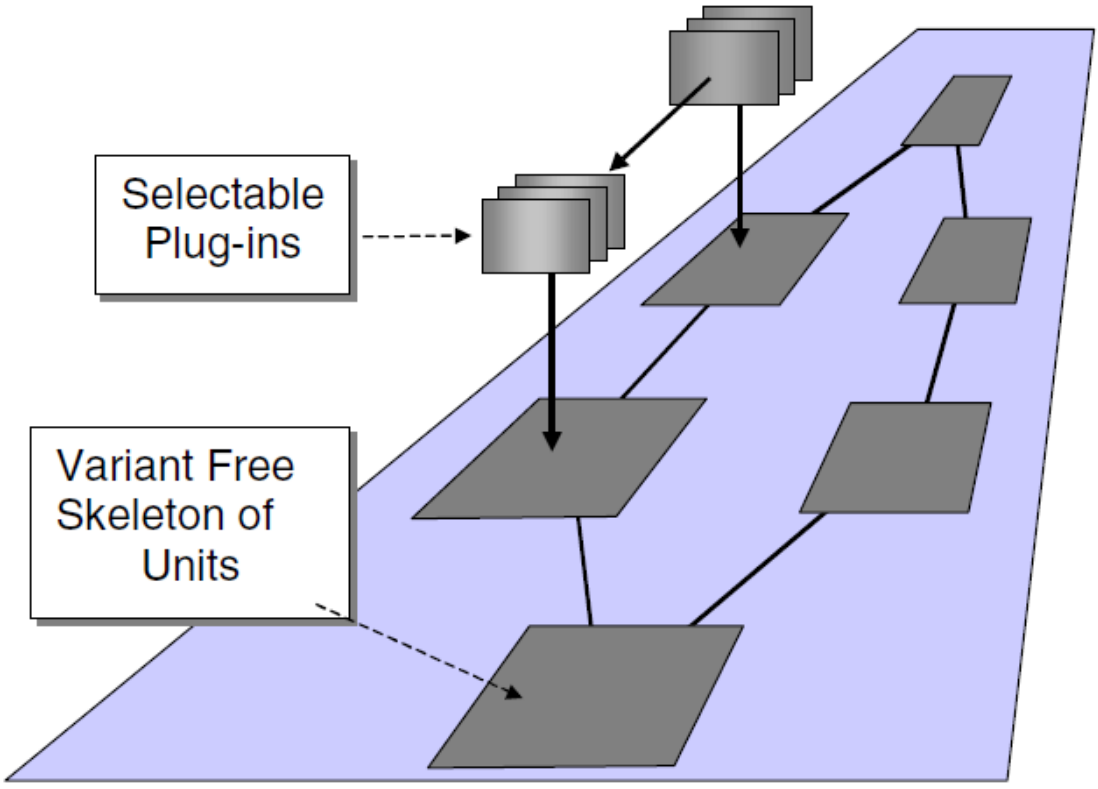

Figure 14. An example of the structure of a variant-free architecture [32].

4.3.1.1 LEVELS OF VARIATION

Variation can typically occur on different abstraction levels of a software product line through the realization of variation points [24, 25]. Software engineering research community [24, 25, 34, 38] has identified a number of variation abstraction levels that are introduced in Table 12 with associated descriptions. The order of Table 12 is descending, i.e. the highest level of abstraction is on top and the lowest is on the bottom.

Table 12. Variation abstraction levels [24, 25, 38].

Variation level	Description
1. Architecture	Architecture level variation enables to support several product specific architectures by reorganizing, via variation points, specific parts of the product line architecture, e.g. by changing the ordering of components and how they are interconnected.
2. Design	In design level variation the actual programming language can be used to introduce variation, e.g. by instantiating functionally different implementations of the same component interface or by inheritance.
3. File	In file level variation variability is implemented at the level of the source code files. For example, some programming languages, such as C and C++, provide conditional compilation support that can be used to implement functional variation.

Regarding Table 12, one should note that architecture level variation isn't typically supported by the programming language and thus customised tool and language support may be required in the form of e.g. Architecture Description Languages (ADL) [38]. The distinction between architecture and design level variation can be a fine one [24]. However, architecture level variation is typically created by external tool support, e.g. by programming, whereas design level variation is directly supported by the implementation programming language [24]. File level variation can be implemented with built-in support of the implementation language or by combination of external build scripts and source code configuration management tools [24, 34].

4.3.1.2 VARIATION MECHANISMS

Once a variation point is identified in the product line architecture it should also be implemented by some form or mechanism [25, 38]. As a consequence, when a variation point is identified and assigned, which is known as the realization of the

variation point, it refers to the selection of the actual implementation mechanism associated with the variation point [25, 38]. Table 13 contains a summary of different variation mechanism implementation types and how they relate to the variability of product line architecture.

Table 13. Types of variation mechanism and their corresponding descriptions [39].

Variation mechanism	Description
1. Inheritance	Utilized when each product in the product line requires either a different or an extended version of the functionality of a common function.
2. Extensions and extension points	Utilized when parts of a component can be augmented with additional behaviour or functionality.
3. Parameterization	Utilized when component's behaviour can be described by a placeholder that Is defined and resolved at build time. Examples of parameterization mechanisms include macros and templates.
4. Configuration and module interconnection languages	Used to manage the build-time structure of the system. Components can be selected and deselected from different product builds.
5. Generation	A higher-level language is used to define properties of components.
6. Compile-time selection of different implementations	Conditional compilation is used to implement the variability of the component and thus is realized by choosing different implementations of e.g. files or individual lines of code.

4.3.1.3 VARIATION PHASES

As the design and implementation of a software product family evolves and variable features are recognized, so called variant features, and thus realized by the means of variation points presented earlier. Recognition of a variable feature is not always an easy task but typically they are identified if the concept of features has been used in the system modelling phase [25, 38]. During the development life cycle of a software product family a variant feature can have a specific state with reference to time.

Initially a variant feature has to be identified for selected group of variants. When the variant feature has been identified it is said to be in implicit state [38]. A variant feature moves out of implicit state when it is introduced to the software product family [38]. The next state for the variant feature is to populate it with its variants [38]. And finally, deciding which variant of the variant feature should be attached to the product family

will be known as the bound state [38]. Table 14 summarizes the aforementioned states with a more detailed description of the specific states characteristics.

Table 14. Summary of variant feature states over the lifecycle of a software product family development [38].

Variant feature state	Description
1. Identified	Initial state of the variant feature once recognized, e.g. during feature modelling.
2. Implicit	Variant feature has not been yet realized, i.e. has not been implemented in the software product family. Exists only as a concept and implementation is deferred to a later stage.
3. Introduced	Commitment to implementing the feature variant has been made and as a consequence it has a representation in the design or the implementation of the software product family. Representation is realized as a set of variation points in the implementation.
4. Populated	Previously introduced variation feature is to be populated with all the variants. For each variant, software entities are created and instrumented in a way that they fit together with the variation points.
5. Bound	In bound state decision has been made on which specific variant to select from the pool of variants particular to the variant feature. Variation points related to the variant feature are now connected to the specific software entities representing the variant.

Relations between variant feature, variant, software entity and variation point can further be clarified as follows:

- Variant feature has a one-to-many relation to a variant, meaning that the variant feature may have one or several possible variations of the feature.
- Variant has a one-to-many relation to a software entity, indicating that a variant may consist of one or more software entities, i.e. references to the actual software resources.
- Variant feature has also a one-to-many relation to variation points that provide the actual implementation of the feature by connecting to zero or more of the software entities.

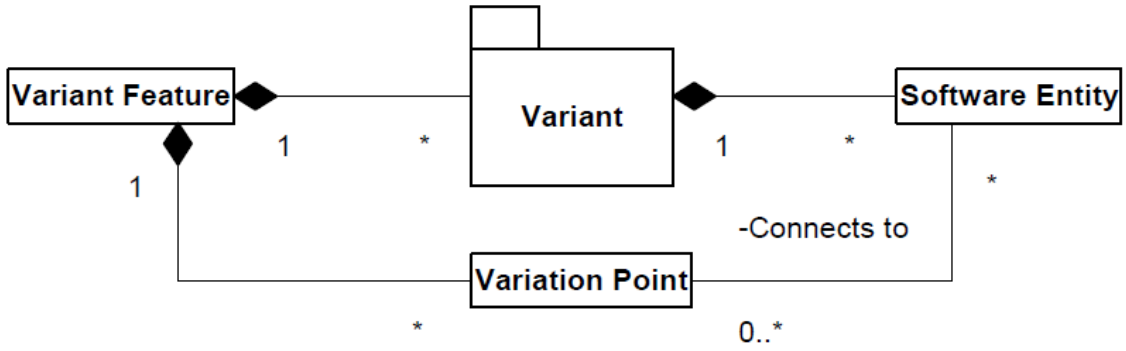

Figure 15. Relations of variation terms [38].

4.3.1.4 VARIANT BINDING TIMES

The previous subchapter discussed the different states of a variant feature. The ultimate state was the bound state, referring to the fact that decision has been made on which variation points and software entities are used to represent the feature. However, the question of "*When to bind*?" was previously left unexplored. One of the purposes of introducing a variant feature is delaying the decision on when to commit, in other words bind, to a particular variant of the feature [38]. Binding can occur at several phases of the software product line development or even in late as when the system is running [38]. There are four possible binding times [38] which are as follows, from the earliest to the latest:

- Product architecture derivation, i.e. during the design phase of the product.
- Compilation, i.e. the phase when software is being compiled to the machine code by the compiler.
- Linking, i.e. the phase when the software is being assembled into the final product by the linker.
- Run-time, i.e. when the system is running the final product of the previous phases.

Of course, during a variant binding phase there are many aspects to be considered which are summarized and described in a more detailed manner in Table 15.

Table 15. Summary of different variant feature binding times [38].

Variant binding time	Description
1. Product architecture derivation	During product architecture design phase some of the unbound product family architecture variation points are bound to produce the architecture for a particular product. This phase typically involves the usage of software configuration tools or high level architecture description languages (ADL).
2. Compilation	After the source code has been finalized by sweeping, e.g. with a pre-processor, through the source code for possible conditional compilation flags and macro expansions, the final machine code product will be produced.
3. Linking	Depending on the programming and runtime environment, binding time is just after the compilation phase or when the system has just been started.
4. Run-time	Variant is bound at runtime from a collection of either closed or open set of variants. Closed set of variants means that the system cannot be extended during runtime contrary to the open set. Typically this type of variation relies heavily on the linking mechanism of the programming language and the underlying platform.

4.3.2 COMPONENT COMPOSITION

Traditional way of doing component-based development (CBD) has been the development of components in isolation without direct knowledge of each other [32, 40]. Ultimately the components are to be assembled in a particular way to create a single working product [32, 40]. At this point distinction should be made on the difference between decomposition and composition. Decomposition means that the system specification is decomposed into subsystems and subsystems in turn into components [41]. However, in composition components can be combined in multiple ways to produce subsystems which in turn can be combined into new subsystems and thus ultimately the system [41]. Figure 16 illustrates the difference between the decomposition and composition paradigms. With component composition any one of the components can become the entry point to the program.

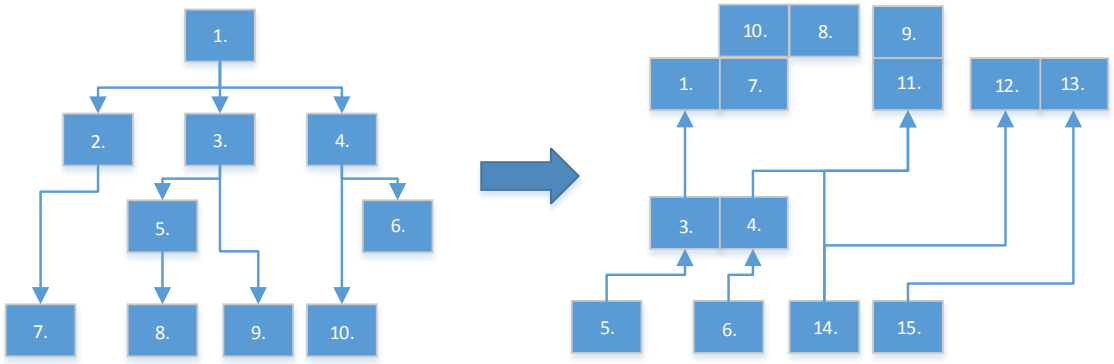

Figure 16. Decomposition versus composition [41].

Components may share interfaces defined elsewhere and may have been developed on top of a common context such as standard C- or C++-libraries [32]. So for the component composition paradigm to be effective and to manage diversity, it has been argued that components should not be directly dependent on each other but rather share some common principles (e.g. interfaces) to work together [32]. For the aforementioned reason component compatibility is a significant problem for component composition-based software systems and therefore also for software product lines [42].

Where variation is used to derive new variants from a single product, component composition can be used to build a population of products provided that variation is also utilized [40]. For the previously mentioned reason two issues should be considered:

- Component variation in time and space.
- Composition management strategies.

In order to successfully manage SPL and the constructed product populations both issues should be addressed.

4.3.2.1 COMPONENT VARIATION

Component variation occurs when component baseline evolves over time, new versions of the component where a specific change was applied to address e.g. a change in existing functional requirement or to correct an existing error. The previous kind of variation is known as time-based variation [34]. Second kind of variation occurs when e.g. new variant feature is introduced to the component; this kind of variation in turn is known as domain space variation [34]. One example of domain space variation is introducing a new product into the SPL. There may be different requirements for the new product, e.g. how the FLASH memory is organized internally and therefore a new variant feature needs to be added into the component in charge of accessing the

FLASH memory. Figure 17 illustrates the distinction of time-based variation and domain space variation as a graph.

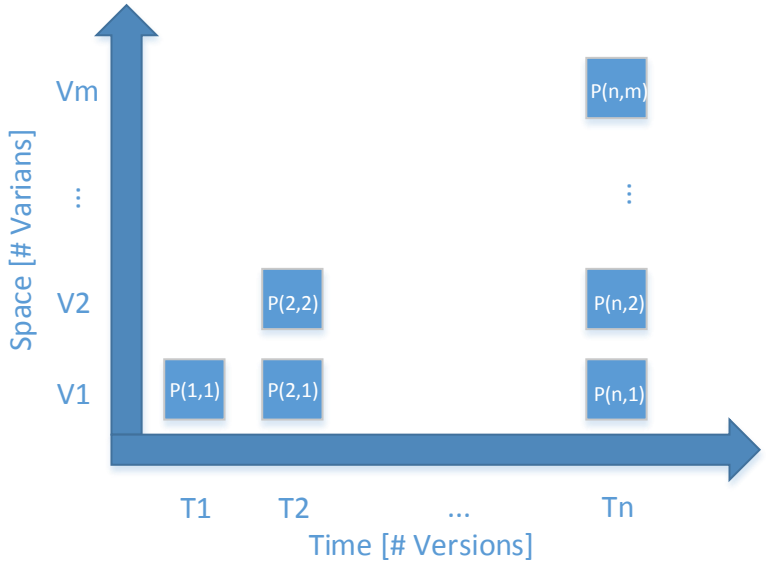

Figure 17. Time (number of component versions) versus domain space (number of variants) [23].

4.3.2.2 COMPOSITION MANAGEMENT STRATEGIES

Strategies for managing component composition arise when components are permitted to evolve independently from each other as their own component baselines rather than as a part of the product baseline [34]. If the components evolve as a part of the product baseline, then component composition is not necessary [34]. In the absence of component composition SPL does not support product populations either [40], the distinction being that the components are equivalent to modular partitions inside the whole software product, decomposed in reusable parts from the system specification, rather than intentionally composed [34]. For component-based product populations any one of the components may be the entry point for a specific software product as there is no single product [32, 40]. To accommodate the variation and versioning of the components composition management strategies are required. The first strategy is called the *shelf* and the second one is called the *context*.

The *shelf* strategy derives from the concept where all versions of the components are stored on a shelf of reusable parts [34]. In the *context* strategy an evolving context of component versions is maintained instead of all versions of each component [34]. Table 16 contains a summary of the two composition management strategies with accompanying descriptions.

Table 16. Composition management strategies [34].

Strategy	Description
1. Shelf	All component versions are treated as peers on the shelf of reusable parts. Consistent component composition combinations between versions must be carefully maintained since many of the combinations are incompatible. The number of possible combinations can quickly explode beyond what can be done manually.
2. Context	Only one possible composition of component versions is maintained. At its simplest form can be implemented as a single file containing a list of latest component compositions of the product. Maintenance of the compositions is much simpler and thus should be preferred whenever possible.

The more intricate difficulties of the component composition scheme are beyond the scope of this work, but for the interested parties, further information can be found in a paper by Garlan et al. [43].

5 VISEDO SPL FOR POWER CONVERTERS

This thesis was commissioned by Visedo Ltd., which is a company that develops and manufactures electric drivetrain components and systems for mobile work machines, marine and bus applications [44].Motivator of thesis was the product development project of the PowerMASTER heavy duty inverter (Figure 18), which is a one of the electric drivetrain components developed, produced and sold by the company. PowerMASTER development project began in 2010 and the first deliveries to customers were made during the course of 2012.

Figure 18. Visedo PowerMASTER heavy duty inverter [8].

Some of the typical heavy duty inverter use-cases in an electric drivetrain were already covered in Chapter 2. Since the original PowerMASTER development project Visedo Ltd. has started to expand its inverter and (to put it in broader terms) power converter products for different mobile work machine, marine and bus applications. One of these new power converter products is the PowerCOMBO product consisting of an inverter and a DC/DC converter the electrical characteristics of which were covered in Chapter 2.

For a small company like Visedo Ltd., it is important to adopt modern processes and methods to accommodate the requirements of the large customers. Adoption of SPL practices to Visedo SPL is very important for similar reasons discussed in Chapter 4.1 and which are:

1. To manage the size and complexity of the code base.
2. To maintain or obtain high quality.
3. Management of product variants.
4. Reduced product development lead times.

In the context of Visedo Ltd. the related products are power converters with similar features, yet varied on hardware and software levels for different customer needs. A typical Visedo Ltd. customer organization is a much larger organization than that of Visedo Ltd. Large companies have many processes in place for achieving e.g. desired lead times while maintaining a high quality. In order to meet the software quality and development requirements of the larger customers with a relatively small number of personnel SPLE and SPL practices are adopted to be used as a (suitable) tool. Many of the SPL incentives identified in Figure 11 are becoming a reality for Visedo Ltd. especially in the case of several products and customers.

The possibility of expanding beyond current market segments in the future can also be a valuable asset for a company. For Visedo Ltd. this is perhaps not the current reality, but from a strategic point of view entering a new market segment can become a future reality due to a not-yet recognised business incentive. SPLE and SPL practices can provide the flexibility to enter new market segments provided that there are sufficient commonalities to be exploited. Scope (Figure 10) of the Visedo SPL is currently determined by the existing products for the previously mentioned market segments that have been either implemented as variant (PowerBOOST) of the original PowerMASTER heavy duty inverter or as a new variant (PowerCOMBO).

5.1 OBJECTIVES REVISITED

The first objective of this work was to document the current state of the Visedo SPL. In order to document the state of the Visedo SPL the following facts should be extracted from the source tree of the product line:

- Current architecture and its style.
- Current component domains and their deployment within the architecture style.

Second objective for this work was to identify future improvement steps in terms of Visedo SPL development practices. The third and final objective was to create guidelines on how to implement the identified improvement steps.

5.2 KNOWN ASPECTS OF VISEDO SPL

There were some known aspects of Visedo SPL that should be noted because they provide the context for the practical aspects of this work and future development needs for the company. For instance, Visedo Ltd. has attempted to utilize component-based development practices in their SPL development efforts. Visedo SPL has been structured in a way where the code base has been decomposed into child projects which produce static libraries as their outputs. And although static libraries do not necessarily adhere to all of the component design principles of Table 5, they should still be regarded as components. For the rest of this work any reference to a specific Visedo SPL component should be regarded as a reference to the corresponding library.

Secondly Visedo SPL is currently in a state of flux where efforts are being made to improve the decomposition of the software platform. For this reason I (the author) have taken a snapshot of Visedo SPL in a fixed point in time, which in its turn will form a baseline for the findings of this work.

5.3 CURRENT STATE OF VISEDO SPL

As stated in Chapter 5.1 that the initial objective is to document the current state of Visedo SPL. This will be accomplished as follows:

1. Extract the architecture’s project structure and identify the architecture style.
2. Identify the component domains and their distribution within the product line. Also review component-based design practices.
3. Identify currently applied SPL practices.
4. Identify how diversity is managed currently.

Associated methods will be presented Chapters: 5.3.1, 5.3.2, 5.3.3 and 5.3.4. The topics numbered above that required more practical insight than discussed in their respective chapters on theory will be discussed in the corresponding chapters to follow. The PowerMASTER J1939 product variant will be used as the reference product.

5.3.1 ARCHITECTURE

Documenting the current state of Visedo SPL was made by identifying following characteristics from the code base for the reasons added below:

1. Structure of the project hierarchy.
    - For documenting the structure and relations of the components that comprise the actual firmware of the product variants.
2. Perceived architecture style and structure.
    - For documenting the architecture style and relation of the components in terms on what dependencies have been declared in each component.
3. Actual architecture style and structure.
    - For documenting the architecture style and relation of the components in terms on what dependencies actually exist between the components.

Once the project structure and component dependencies have been identified architecture style can be analysed from two different perspectives listed above.

5.3.1.1 PROJECT HIERARCHY EXTRACTION

In order to visualize the structure of current project hierarchy, the deployment of the components and binary targets relevant to the construction of the product firmware should be identified from the Visedo SPL source tree. Visedo SPL utilizes a high level generator tool capable of producing a wide range of build and project files, such as various: command line and graphical user interface (GUI) tools. Visedo SPL source tree consists of a root project file and several child project files under the root project file, i.e. it forms a tree hierarchy of binary targets. Some of the binary targets are not a part of the final firmware binary and therefore they should be omitted from the structure extraction process.

Project structure extraction process for Visedo SPL is straightforward: the input files of the generator tool are scanned for specific keywords linking the child project to the root project. Child projects that are not part of the product's final firmware should be discarded. After the input files of the generator tool have been scanned for the specific keyword and the irrelevant child projects have been pruned out the results, a list of

directories for the projects that comprise the actual structure of the products will be found.

A custom tool can be constructed for this kind of fact extraction process but basic text search support offered by many text editor tools can also be used. Table 17 summarizes the steps for extracting the project structure that comprises the structure of product firmware in the Visedo SPL.

Table 17. Phases for extracting child projects constituting firmware binaries from Visedo SPL.

Phase	Description
1. Identify all child projects from the root of the Visedo SPL directory hierarchy.	Root directory is scanned recursively for child project files with a tool capable of searching text files. The end result is a list of directories where child projects have been defined.
2. Discard child projects that are not part of the firmware binaries of the products.	Child projects that are not part of the firmware binaries should be discarded, either manually or by providing a list for a custom tool from the list of directories acquired in the preceding phase. At this point a list of directories and child project files relevant to the construction of the firmware binary targets of each product has been identified.
3. Construct project file directory hierarchy of the child projects that constitute the actual firmware binaries of the products.	In the final phase resulting project file directory hierarchy should be visualized as a graph. This can be made manually or with a custom tool.

For this work's context manual extraction was still a feasible option, i.e. faster than developing a custom tool, and therefore Figure 19 was produced manually with a combination of a common text editor and diagram drawing tools. The results of the previously described extraction process are depicted in Figure 19 containing the layout of the directory hierarchy of the child projects used for constructing the final firmware images of the products.

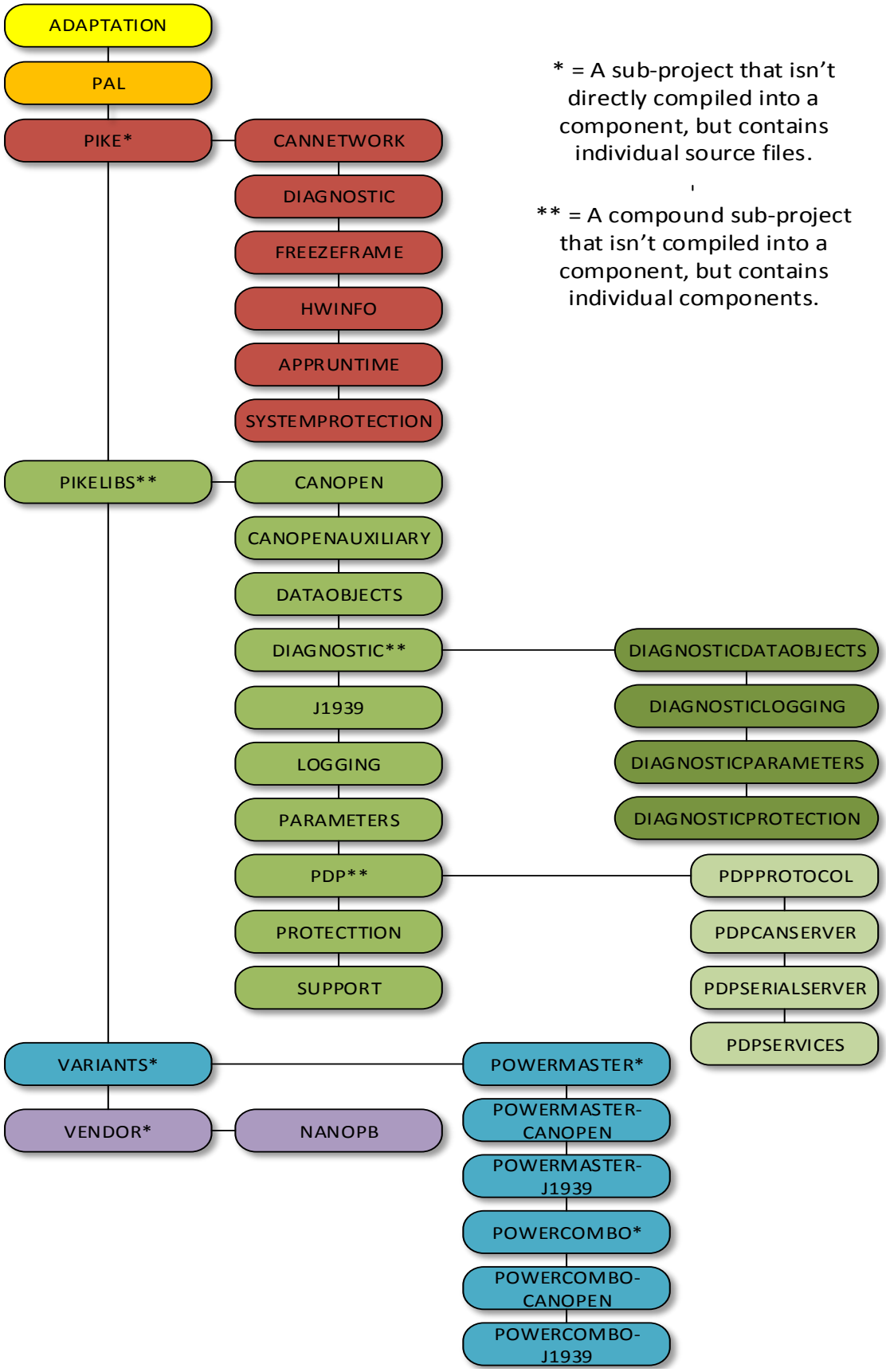

Figure 19. Project hierarchy structure of Visedo SPL.

5.3.1.2 PERCEIVED ARCHITECTURE EXTRACTION

Architecture style extraction is a more difficult task than analysing the structure of a project hierarchy that comprises the firmware images for each product. The difficulty stems from the fact that dependencies among child projects (i.e. components) should be identified in order to visualize them as a view of the architecture similarly to Figure 6 and Figure 7. However, the project and build file generator tool utilized by the Visedo SPL is able to output a dependency graph of the product's components forming the final firmware image; this will serve as a starting point for extracting the architecture style. Once the dependencies between components are known, further analysis on the structure of the architecture and its style can be made.

Figure 20 depicts the dependency graph of the components and their dependencies comprising the PowerMASTER product's J1939 communication variant firmware. The dependency scanning function of the generator tool actually produces a text file that can be loaded by an external tool to produce the final dependency graph. During the product variant's firmware build process some of the components are generated by a code generator tool receiving its input from configuration files. The three libraries that are generated by the code generator tool are:

1. DATAPARAMCONF
2. J1939CONF (J1939 variants) or CANOPENCONF (CANopen variants)
3. PROTECTIONCONF

These libraries are generated from the configuration information defined by Extensible Markup Language (XML). These XML configuration files are used to manage following product-specific parts of the firmware build process:

- Parameter and data object system values (DATAPARAMCONF).
- Communication system values (J1939CONF / CANOPENCONF).
- Protection system values (PROTECTIONCONF).

The purpose of the DATAPARAMCONF component is to set the product's parameter system in a state that reflects its intended use at the customer. For instance, DATAPARAMCONF component can be generated with preconfigured motor control parameters which do not have to be inputted by the customer, provided that Visedo Ltd. has been informed about the intended usage and requirements of the customer application in advance. Similarly J1939CONF is used to preconfigure parameter values related to J1939 communication, e.g. setting the initial J1939 bus address for the

product. And finally PROTECTIONCONF is used to preconfigure the parameter values of the power converter's protection system, e.g. what kind of faults are generated and when.

Figure 20. Component dependency graph of the PowerMASTER J1939 communication variant using the dependency output function of the generator tool.

As the dependencies between projects have now been identified, the dependency graph can be translated into a layered architecture graph based on the definition of layered architecture style given in Chapter 3.2. By arranging the components into layers according to the node path length from the root node (PowerMASTER-J1939) the layered architecture structure can be presented. The rules for node layer arrangement are summarized in Table 18.

Table 18. Node layer arrangement rules.

Rule	Description
1. Root node	Place root node into Layer(Max_Path_Length(All_Nodes)).
2. Root child nodes	Place child node into Layer(Max_Path_Length(All_Nodes) – Path_Length_To_Root(Node)).

Applying the rules given in Table 18 to nodes in the dependency graph (Figure 20) and taking into account the definition of OSI Reference Model from Chapter 3.2 the following layered architecture structure can be obtained, which is presented in Figure 21.

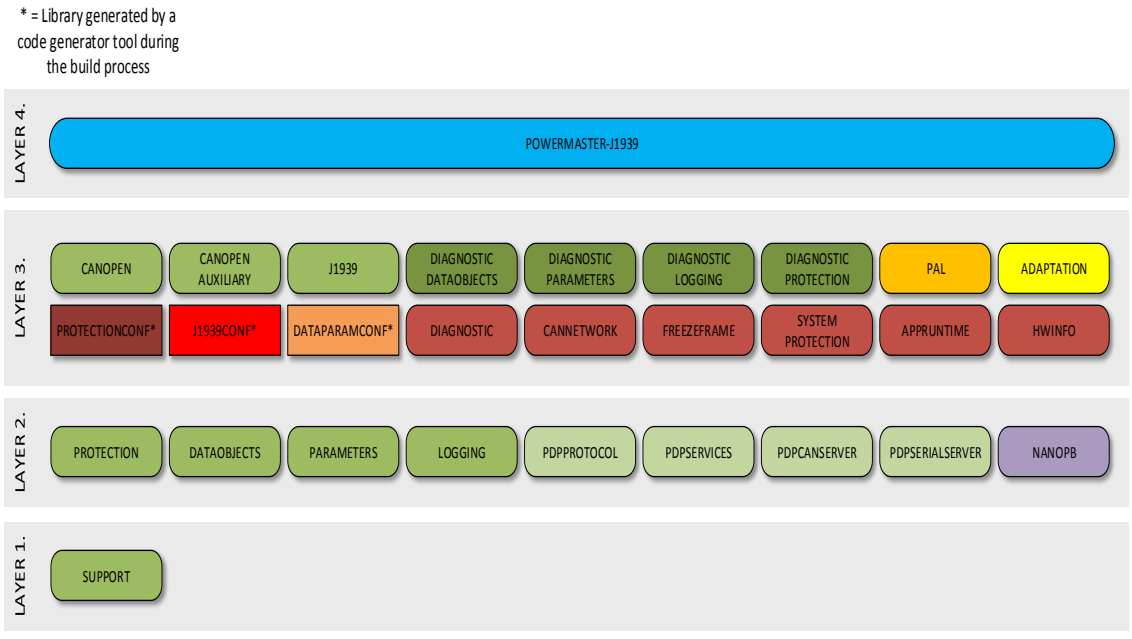

Figure 21. Layered architecture structure extracted from the PowerMASTER J1939 communication variant of the Visedo SPL.

5.3.1.3 ACTUAL ARCHITECTURE EXTRACTION

Since the components of the Visedo SPL are implemented as static link libraries, which are compiled with a C-language compiler, the symbols used by the component do not necessarily have to be explicitly declared in any of the other components that it is dependent on. Another way to put this is that even if each component explicitly defines its dependencies to other components, it still can have references to symbols that are only resolved when all components are linked to the final executable, i.e. the firmware image. The concept of these externally declared symbols that are resolved during the linking time of the final product is actually fairly common design technique employed in embedded systems software. These external symbols can also be described as implicit dependencies in contrast to the explicit ones defined in the project files. However, the drawback of the aforementioned technique, in terms of the layered architecture style, is that the perceived architecture and the actual architecture can be different than expected. Other kinds of drawbacks are the growth of the code base in size and complexity. Table 19 summarizes some of the possible drawbacks of the externally declared symbol design technique utilized in Visedo SPL from layered architecture style point of view.

Table 19. Drawbacks of externally declared symbols, i.e. implicit dependencies.

Drawback	Description
1. Perceived architecture vs. actual architecture.	Perceived architecture does not match the actual architecture, can be more complex than anticipated.
2. Unexpected component dependencies.	Components may have unexpected dependencies as a consequence of externally declared symbols that are resolved at higher level than the component's e.g. during the linking of the top level at the latest.
3. Component layer dependency leaps.	Components have dependencies that can leap several layers and thus are not in alignment with the perceived or logical distribution of components to layers.

In order to compare the differences between perceived architecture (explicit dependencies) and the actual architecture (implicit dependencies) a set of scripts for extracting architecture were developed. The scripts were developed with the Python programming language [45]. The dependency graph was drawn with the Graphviz tool [46]. The actual process of extracting architectural facts takes place in four phases which are described in Table 20 in the execution order of each phase.

Table 20. Phases of the architectural extraction process for producing a component dependency graph.

Phase	Description
1. Scan used symbol output information produced by the linker.	Linker produces a XML file containing information about each symbol used in the firmware binary. For each symbol information is provided in which source file and component it has been declared.
2. Scan symbol cross reference listing output information produced by the linker.	Cross reference listing file, produced also by the linker, contains information about each symbol's type of usage, i.e. where it is defined, declared and referenced. Location and type of usage are listed on source file level.
3. Merge used symbol and cross reference listing output information into a single flat database.	Output information from two previous phases is merged into a single flat database. Database contains information regarding the declaration (source) component of each symbol and the component using the corresponding symbol (destination). A database containing dependencies between components is produced.
4. Draw dependency graph of the components based on the symbol declaration and usage information from the merged database.	A dependency graph from the output of the previous stage is produced by drawing directed arcs based on each component's source and destination information.

Based on the phases presented in Table 20 corresponding scripts were created which can later be utilized in monitoring the structure of architecture and component dependencies. The purpose and function of each script is presented in Table 21.

Table 21. Purpose and function of scripts for extracting architectural facts.

Script	Purpose/function
1. scan_link_info_xml.py	Scan XML file produced by the linker and produce a comma-separated output of the symbol type and the location of its source file declaration with the accompanying component where it belongs to. The script is listed in Appendix 2.
2. scan_cross_reference_listings.py	Scan the cross reference listings files produced by the linker for each source file and output a comma-separated list of each symbol's usage type and location of the usage within the source file. The script is listed in Appendix 3.
3. merge_scan_results.py	Merge outputs of the two previous scripts into a single database file separated by commas. The script is listed in Appendix 4.
4. draw_scan_results.py	Reads the database file separated by commas produced by the previous script and outputs a dot notation file that is fed to Graphviz tool for producing the final dependency graph. The script is listed in Appendix 5.

The dependency graph information was extracted by tools presented in Table 21. The tools were used to construct a dependency graph in form of directed arcs between components to reveal undesired or redundant dependencies between components. This information can be used to identify and plan re-factorization efforts concerning the future development of the Visedo SPL. The constructed dependency graph is depicted in Figure 22. As expected, the dependency graph in Figure 22 is more complex than the one in Figure 20, which implies a usage of externally declared symbols and therefore that components have implicit dependencies between them. The most significant deviations between Figure 20 and Figure 22 are listed in Table 22 in descending order of significance.

Table 22. The most significant deviations between perceived and actual architecture structures.

Deviation	Description
1. Cyclical dependency	POWERMASTER-J1939_DATAPARAMCONF and PARAMETERS components depend on each other and are thus tightly coupled.
2. Cyclical dependency	ADAPTATION and PAL components depend on each other and are therefore tightly coupled.
3. Increased outgoing dependency count.	DIAGNOSTICS component is collaborating with larger number of components than expected.
4. Unexpected component dependency.	LOGGING component depends on ADAPTATION component, contrary to what was declared in the project files.
5. Layer dependency leap.	POWERMASTER-J1939 which is a layer 4 component uses services from LOGGING component which is from Layer 2.
6. Increased incoming dependency count.	Support component is used by larger number of components than expected.

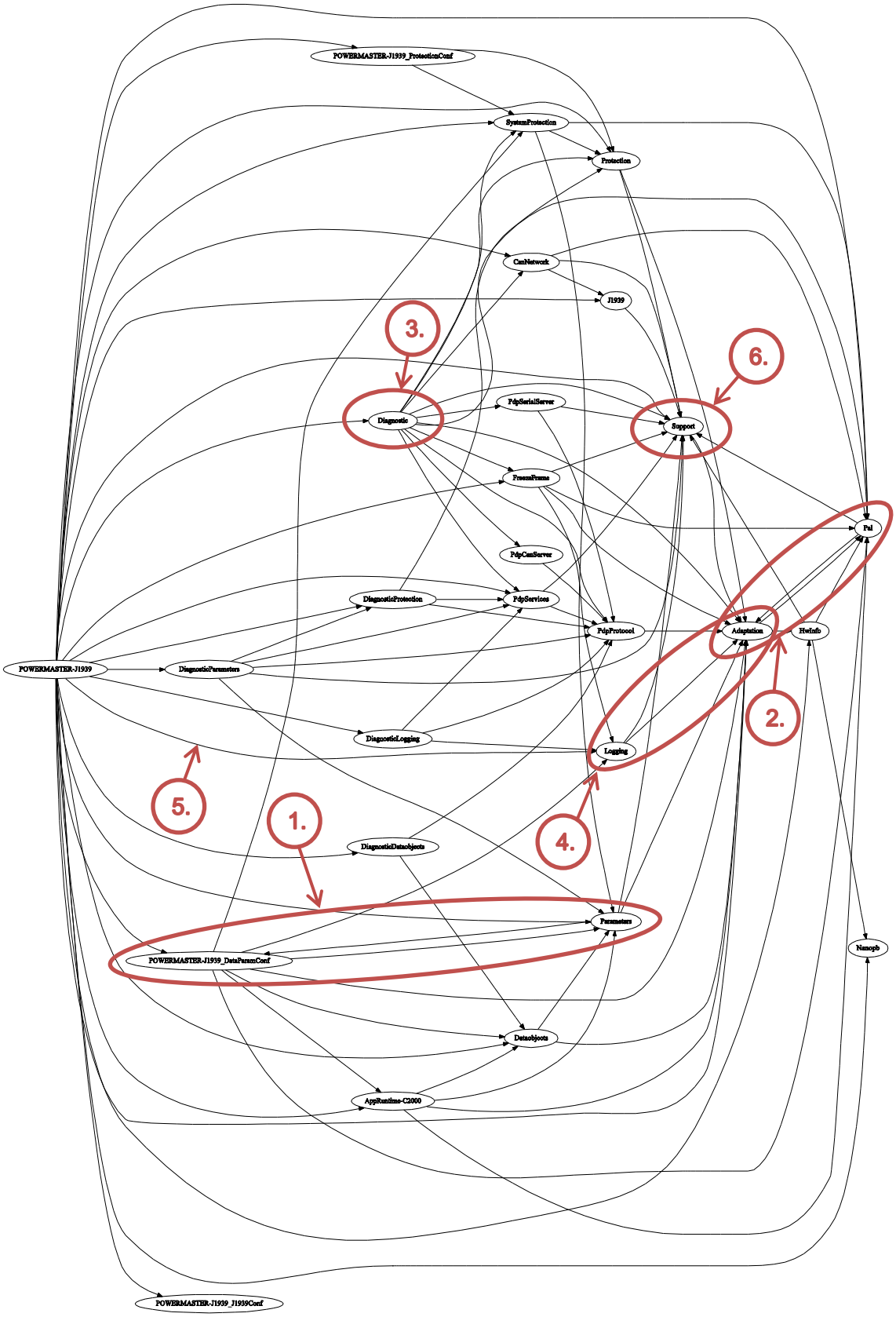

Figure 22. Component dependency graph of the PowerMASTER J1939 communication variant using the architecture extraction scripts where deviations have been marked and numbered.

5.3.2 COMPONENT-BASED DEVELOPMENT PRACTICES

In Chapters 5.3.1.2 and 5.3.1.3 the structure of architecture style was extracted with available tools for the purpose of documenting the current state of the Visedo SPL architecture. Part of the architecture documentation objective was also to document the domain layout of the components. Comprehensive component domain evaluation would require extensive efforts in analysing the functionalities, responsibilities and internals of each component. Therefore, the component domains of the current set of components will be described and they are based on the author's view on the matter. The product variant level will also be omitted from the component sets despite containing the bulk of source code and possible candidates for future components, e.g. different control domains such as motor, brake and DC/DC controls.

Adoption of component-based development practices is being evaluated by the company and therefore it is justified to identify initial component domains. The identification process for each component was performed on a layer basis (Figure 21), starting from Layer 1 and finishing at Layer 3. By examining the names of the components and roughly categorizing each component-based on their functionalities and responsibilities the following graph (Figure 23) was produced.

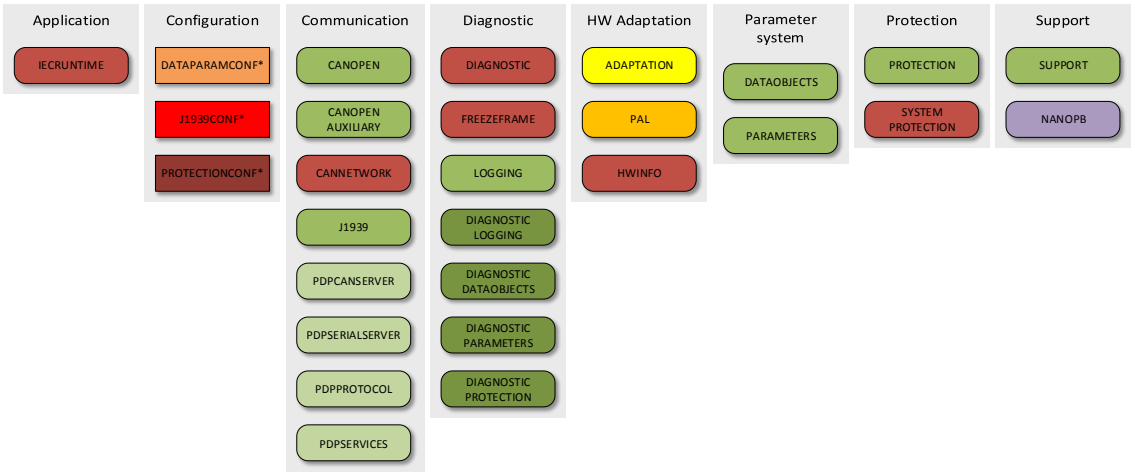

Figure 23. Component domains identified in the Visedo SPL.

Table 19 listed some of the drawbacks of the current architecture structure which are also relevant if the company decides to adopt component-based development practices. These drawbacks conflict with some of the design principles for components presented in Table 5. Table 23 lists a subset of component design principles in Table 5 which conflict with the drawbacks presented in Table 19.

Table 23. Conflicts of component design principles and their descriptions.

Design principle	Conflict description
3. Independent of the context	External symbols can be used to provide information from the outside of the component but can also be used to bypass the component's public interface and thus to bend the "Independent of context" design principle.
5. Encapsulated	Implicit dependencies conflict with the "Encapsulated" design principle in such a way that they can expose e.g. internal processes to the outside and thus can cause layer dependency leaps.
6. Independent	Unexpected component dependencies conflict with the "Independent" design principle's dependency minimization objective therefore making component harder to deploy e.g. in different product configurations.

5.3.3 SOFTWARE PRODUCT LINE PRACTICES

The Visedo SPL originates from the first heavy duty inverter (PowerMASTER) product development project which began in 2010 and the first deliveries to customers were made in 2012. The initial scope of using the PowerMASTER heavy duty inverter consisted of controlling traction motors e.g. in mobile work machines and converting AC generator power to DC to be e.g. stored in an energy storage [8]. Since the beginning of the aforementioned development project, software development has been organized in a way where platform development, i.e. core asset development, has been outsourced to a company specialising in developing embedded software. Domain and product engineering are done in-house at Visedo Ltd. in addition to the required development management activities. Development practices have remained pretty much the same with the exception of new products since the PowerMASTER development project. Currently the Visedo SPL consists of PowerMASTER, PowerBOOST and PowerCOMBO products which all have been instantiated from the same shared software core assets.

Given the previous general overview of Visedo Ltd. and comparing the current development practices with the activities and practices associated with the SPL development paradigm covered in Chapters 4.1 and 4.2, it can be said that Visedo Ltd. has already adopted parts of the SPL development paradigm. It has been important for Visedo Ltd. to reuse many of the assets already created in order also to exploit the associated benefits. Table 24 lists the same reusable assets as Table 8 along with the information whether a specific type of asset is currently being applied to the development of Visedo SPL or not.

Table 24. Checklist of the currently applied reusable assets of Visedo Ltd.

Reusable asset	Applied	Description
1. Requirements	Yes	Requirements are common for shared platform components whereas product-specific requirements are maintained separately.
2. Architecture	Yes	Architecture is similar in all products, where product specific functionality is implemented through variant level variation points.
3. Components	Partially	As discussed earlier that the decomposition effort into components is an ongoing effort and therefore more benefits remain to be obtained.
4. Modelling and analysis	Yes	Performance, resource allocation and timing analysis models can be reused due to sufficiently similar products and application areas.
5. Testing	Yes	Test cases and plans are reusable since the power converter product variants mainly consist of the same features with the exception of different power ranges.
6. Planning	Yes	Development costs and plans from the PowerMASTER product development project are amortizable with similar product variants such as PowerBOOST and PowerCOMBO.
7. Processes	Yes	Same tools, processes and practices are used in the development of each product.
8. People	Yes	Number of the people involved in the development of Visedo SPL has remained constant despite the increasing number of products to be developed and maintained.

Based on the observations given in Table 24 Visedo Ltd is already able to reuse many of the previously developed assets; some parts still require efforts from the company, but are under active development.

A general overview on the organisation of the development of the Visedo SPL in its current state was given earlier. Correlating the aforementioned state with the three essential SPL practices discussed in Chapters 4, 4.1 and 4.2 following correlations can be found, which are presented in Table 25.

Table 25. The three essential activities of the SPL paradigm and their associated descriptions in the context of Visedo Ltd.

Essential activity	Description
1. Core asset development	A large part of the platform development is outsourced to a company specialising in developing software for embedded systems, but domain engineering is done in-house. Domain engineering consists of control and parts of communication engineering activities.
2. Product development	Product development is done in-house and built from the core assets of the Visedo SPL.
3. Management	SPL management is organized and provided by Visedo Ltd.

5.3.4 MANAGEMENT OF DIVERSITY

Diversity in the Visedo SPL has been implemented in the form of product variants. The directory structure of the Visedo SPL was presented in Figure 19 where the product variants are grouped under the VARIANTS directory. Product variation resembles variant-free architecture discussed in Chapters 4.3 and 4.3.1. An additional variation mechanism is the parameter patch file system that can be used to manage and overwrite parameter system values compiled into the actual firmware. Patch files are stored into the devices FLASH memory and are loaded during its boot process, after the patch files have been read, firmware's default values can be overwritten with the values found in the patch files.

For every product in the product variant architecture there is typically a parent product (variant-free skeleton) from which the actual product variants are derived. The parent product implements the shared features of the product and the derived product variants implement the diversity on top of the parent product. In practice the derived product variants currently implement either CANopen or J1939 communication interfaces of the device via build time variation. The previous applies to PowerMASTER, PowerBOOST

and PowerCOMBO products. The parameter patch files disable and enable selected features from the PowerBOOST product's firmware at runtime. The structure and relations of PowerMASTER, PowerBOOST and PowerCOMBO products are presented in Figure 24. The root level of the product hierarchy is the VARIANTS node which corresponds to the VARIANTS directory in Figure 19. Variant free products, i.e. PowerMASTER, PowerBOOST and PowerCOMBO are derived from the VARIANTS node to provide the actual product variants.

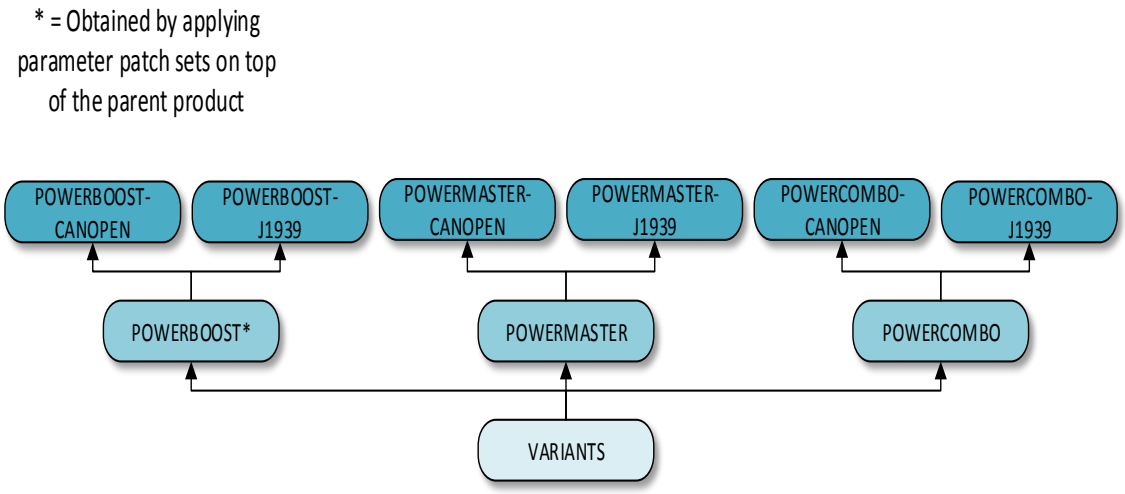

Figure 24. Product variant tree representing the Visedo product hierarchy.

In terms of the variation binding times discussed in Chapter 4.3.1.4 the product variant and the parameter patch file schemes correspond as follows:

- Product variant scheme is a combination of design and compile time variation.
- Patch file variation scheme is equivalent to run-time variation.

Different variation mechanisms are used extensively in the Visedo SPL. As previously noted, the parent product level contains variation points which are realized in the product variant level. Also patch files in association with the parameter system can be used to implement variation. Table 13 listed different variation mechanisms many of which are also used in the context of the Visedo SPL. A subset of the variation mechanisms in Table 13 were identified and are presented in Table 26.

Table 26. Types of variation mechanisms identified in the Visedo SPL.

Variation mechanism	Description
2. Extensions and extension points	Usage of externally defined symbols can be considered as extension points where the extension point is realized differently in different product variants, though this kind of variation can conflict with the component design principles discussed Chapters 3.3 and 5.3.2.
3. Parameterization	Patch files and parameter system should be considered as a form of parameterization variation mechanism despite the fact that not being implemented at build time but runtime instead.
5. Generation	Code generation tools for injecting e.g. product-specific configurations into the final binary product should also be considered as a form of altering the behaviour of the components with a high-level programming language.
6. Compile-time selection of different implementations	Compile-time selection of e.g. different lines of code is used, but considered difficult to maintain and therefore not used very extensively.

5.4 CORRELATION OF THE VISEDO SPL BETWEEN THEORY AND PRACTICE

Based on the findings from Chapters 5.3.1, 5.3.2, 5.3.3 and 5.3.4 results will be presented for each numbered topic presented Chapter 5.3. Therefore Chapters 5.4.1, 5.4.2, 5.4.3 and 5.4.4 are the respective counterparts for the methods presented earlier. Results will be presented as tables, figures and lists with related discussion on the topic.

5.4.1 ARCHITECTURE

First focus area for analysing the Visedo SPL was architecture style and structure which was evaluated in terms of three factors:

1. Structure of the project hierarchy.
2. Perceived architecture style and structure.
3. Actual architecture style and structure.

Structure of the project hierarchy was presented in Figure 19 and therefore the figure should be considered to reflect the current state of the directory layout of the Visedo SPL. Extraction of the Visedo SPL's project hierarchy (Chapter 5.3.1.1) served as an

important foundation for future improvements (Chapter 6) to be derived from this work. Discussion regarding the two remaining factors will be presented in their own respective chapters.

5.4.1.1 PERCEIVED ARCHITECTURE

The process of extracting perceived architecture was described in Chapter 5.3.1.2. The outputs of this process were given as Figure 20 and Figure 21. Figure 20 depicted component dependencies based on how the dependencies have been declared in the project file of each component. This can be described being as the perceived architecture since it reflects the structure of architecture as it was intended by the software developers. Based on the perceived architecture structure presented in Figure 20 architecture style was extracted according to the rules of arranging dependency graph nodes given in Table 18. Based on the arrangement rules of Table 18 resulting architecture style was presented in Figure 21. From Figure 21 and the corresponding component project files for each layer the following observations can be made:

- Layer 4 is the product variant level, i.e. it is in charge of assembling the product firmware image. Turns out that the bulk of the implementation is at this layer based on the list of included source files.
- Layer 3 contains most of the components and therefore provides many services related to communication and application for the product variant layer.
- Layer 2 provides the core services for logging, parameter/data object system and diagnostic communication (i.e. by the PDP-prefixed components) capabilities.
- Layer 1 contains only one component and thus provides currently only basic support services.

When Figure 21 is compared with layered architecture structure of e.g. MeeGo software platform presented in Figure 6 a few additional and rather obvious observations can be made:

- PAL (Platform Abstraction Layer) and ADAPTATION components are, contrary to what one might expect, on the third (second highest) layer rather than on the first (lowest) layer.
- Number of components per layer increases quite rapidly towards the highest layer.

These additional observations however relate closely to a known and currently ongoing effort of decomposing the product variant level further into components. One essential domain waiting to be placed into a component is the control algorithms, which depend on the services of PAL and ADAPTATION components. Now that the control algorithms are part of the product variant level the aforementioned components are “drawn” to a higher layer than necessary. As the decomposition effort progresses, the distribution of components to proper layers should align the layered architecture structure also.

5.4.1.2 ACTUAL ARCHITECTURE

Chapter 5.3.1.2 in Table 20 described the actual architecture extraction process phases with the custom tools presented in Table 21. Output of the actual architecture extraction process was shown in Figure 22. The most significant deviations between the perceived and actual architecture were noted in Figure 22 and translated into observations presented in Table 22. Based on the observations listed in Table 22 it looks as though periodic tracking of architecture deviations with the tools given in Table 21 can be used to detect deviations in architecture. These tools could be also used to plan efforts to re-factorise architecture.

The dependency graph information (Figure 22) is also useful for detecting components that are highly coupled, i.e. are cyclically dependent on each other. Removing redundant and undesired dependencies helps promoting layered architecture design principles listed in Table 3. The dependency graph also explains, along with ongoing decomposition effort, the current layered architecture presented in Figure 21 where low level components are drawn near to the product variant level. The observations and outputs from the actual architecture extraction process were discussed in Chapter 5.3.1.3.

Outputs of the extraction process were translated into following steps for improvement with corresponding description of the benefits:

1. Improve functionality and responsibility definitions of each layer.
    - Promotes the design rules “Abstraction” and “Responsibilities of the functional layers” in Table 3.
2. Align child-project directory hierarchy into layers, explicitly to reflect the chosen architecture style and the responsibilities of the components within the layers.
    - Refactoring the sub-project directory hierarchy in a way that reflects the layered architecture structure.
    - Makes architecture structure explicit and therefore forces to think where new components should be placed within the layered architecture.
3. Periodic cross-referencing and tracking of perceived architecture versus actual architecture.
    - Periodic tracking of the actual architecture helps in identifying deviations from the perceived architecture in time and therefore provides a periodic feedback loop for re-factorisation efforts.
4. Removal of cyclical dependencies between components.
    - Promotes “Loose coupling” design rule of Table 3.
5. Removal of redundant component dependencies.
    - Promotes “High cohesion” and “Reusable” design rules of Table 3.

Finally, based on the identified steps for improvement stated above, the following future guidelines are suggested regarding the structure of the architecture:

- SPL directory structure should explicitly reflect the architecture structure.
- Adoption of architecture design guidelines should help to promote design principles for architecture style, see Table 3.
- Actual architecture should resemble closely the perceived architecture; otherwise there is a conflict between design and implementation.
- Layers should only depend on the services of the immediate layer below and provide events or notifications to the immediate layer above.

5.4.2 COMPONENT-BASED DEVELOPMENT PRACTICES

Component-based development practices of Visedo SPL were discussed in Chapter 5.3.2. Visedo SPL is structured into layers and component domains can be identified for each component as shown in Figure 23 in Chapter 5.3.2. However, when looking at the conflicts in component design principles listed in Table 23 it is possible that

adopting more component design principles could improve the management of diversity between existing and future products. This would enable to create products with a broader spectrum of features. As a consequence following list of steps for future improvement in regard to component-based design and development practices are suggested:

1. Adopt feasible component-based design principles and development practices.
    - Components become more autonomous and are therefore more independent from individual products and their features.
    - Enables the most fine-grained way of composing products from individual components.
2. Identify and assign a domain for each component.
    - Helps in improving mapping high-level product features to individual components belonging to a specific feature domain within the Visedo SPL.
    - Offers a medium-grained way of composing products, i.e. entire component domains can be omitted from a product or conversely included into a product.
3. Identify and assign the layer to which each component belongs to.
    - Helps in making the responsibilities and dependencies of components more explicit and e.g. helps in recognising the refactoring needs of existing components.
    - When a new component is placed into a specific domain and layer, the design requirements would be more identifiable.
    - Additional benefit is the most large-grained way of composing a product when layer scope can be utilized in the composition of the product.
    - A low-end product can consist of individual component domains or components from the lower layers versus a high-end product consisting of component domains and components from all layers.
4. Create a component database.
    - Helps in identifying new core asset development needs and management of the existing ones.
    - New product requirements can be easily cross-referenced with existing core assets of the Visedo SPL.

Should Visedo Ltd. choose to adopt the improvement steps listed above, then the following practical guidelines are provided for implementing said steps for improvement:

- Emulation of the public interface characteristics of a component (functions, properties and events) with a programming language which is not object-oriented and typically not associated with component-based development practices, either.
    - Events can be implemented as callback functions registered by the higher-level component to the lower-level component providing the events.
    - Properties can be implemented as functions that set and get specific property values through the interface provided.
    - Finally the functions that don't set or get properties of the component are regarded as the functions of the component interface.
- Reorganize the SPL directory hierarchy in a way that reflects the true distribution of component domains and their members within the domain to proper layers.
- Associate each component with a manifest file that can be scanned with proper tools to build component database containing e.g. description and dependencies of each component.

5.4.3 SOFTWARE PRODUCT LINE PRACTICES

The practices of software product line of Visedo Ltd. were discussed in Chapter 5.3.3. It was observed that according to the activities of Table 25 Visedo Ltd. has already been able to take the first steps towards SPL development activities. Currently the company may not be able to absorb all of the processes associated with the core asset and product development activities such as formal production plans and strategies, but these are something that can be constructed in time. When thinking of the next steps of improvement regarding the SPL development practices it is the author's opinion that improving the architecture and component-based development issues discussed in Chapters 5.4.1 and 5.4.2 would also enforce the SPL development practices.

Hence, only one guideline will be suggested for SPL development practices:

- Adopting further architecture and component-based development practices would also promote the SPL development practices.

Further improvements in the practice and process of SPL development can be made later as the current state is rather well established despite the fact that the software development began in 2010.

5.4.4 MANAGEMENT OF DIVERSITY

Chapter 5.3.4 discussed the last focus area of the Visedo SPL that was considered to be in the scope of this work. Currently the Visedo SPL utilizes each of the three variation abstraction levels presented in Table 12 by examining the different variation schemes and mechanisms used for constructing the product variants depicted in Figure 24. Based on the observations presented in Table 26 of Chapter 5.3.4, following improvement steps regarding the variation scheme of the Visedo SPL are suggested with their corresponding descriptions:

1. Finish decomposition efforts.
    - Ongoing component decomposition efforts should be finalized. Provides a baseline for composition development efforts.
2. Shift focus from decomposition to composition.
    - Once the decomposition efforts have been completed focus should be shifted to development of component composition practices.
3. Develop a component versioning scheme.
    - Component versioning or variation scheme should be developed in order to manage the build-time structure of different products.
4. Develop a strategy for managing component composition.
    - To fully utilize and manage component versioning a component composition strategy should be developed.

Based on these improvement steps new product variants may be constructed by composing components with a specific version of the Visedo SPL and utilize the existing mechanisms also. The product composition approach would require additional investments in component composition management and versioning tool support as maintaining the component composition and versioning configurations by hand can be time-consuming. These tools could be developed in-house but it probably makes more sense to explore existing solutions by 3rd parties to the problem first. A consequence of

the component composition scheme is that products should be defined out of the core asset tree to separate production concerns of core assets and products. Guidelines on implementing additional variation management schemes for Visedo SPL can be summarized as follows:

- Choose composition management strategy such as the *context* approach discussed in Chapter 4.3.2.2.
- Acquire or build tool support based on the chosen composition management strategy and versioning scheme.
- Reorganize the SPL directory so that the products are in a parallel tree hierarchy in regards to the core assets.
- Test and develop component composition and versioning practices. Start with a low-end product that is limited in functionality compared to the high-end products.

6 DISCUSSION AND FUTURE DEVELOPMENTS

This thesis explored the possibilities of utilizing modern practices of software engineering for developing embedded software to power converters. The work was commissioned by Visedo Ltd. The objectives for this work were:

1. To document the current state of Visedo SPL.
2. To identify the future improvement steps for Visedo SPL development practices.
3. To create guidelines on how to implement the improvement steps.

The four selected focus areas of software engineering practices covered in this work were picked in a way that would benefit the company's future efforts to develop software. Adoption of architecture styles, component-based development and SPL practices for developing power converter software has not been mainstream thus far and therefore the practices in developing mobile phone software served as an inspiration for the substance of this work. The mobile phone sector has been able to deliver products of a great diversity (low-end and high-end) for a number of years now and Visedo Ltd. is aiming to produce a range of power converter products in a similar manner. In order to address the objectives of this work the following software engineering areas were investigated:

- Architecture styles with emphasis on layered and component-based architectures.
- Component-based development practices.
- SPL development practices with emphasis on core asset and product development areas.
- Management of product diversity with emphasis on component composition and versioning.

Author of this work believes that improvements in the areas mentioned above will help in the task of producing a range of power converters for different areas of application in the future. The current state of software platform of the Visedo Ltd. was investigated in order to identify the next development steps in terms of the areas for practicing software engineering covered in this work.

The findings on the current state of software development practices in selected areas can be summarized as follows:

- Architecture style is actually layered, but the implicit dependencies between functionally different software domains can obscure this.
- Component-based design practices have been utilized, though the extent of used design principles should be reviewed.
- Many of the SPL development practices have been utilized from the beginning as core asset development has been outsourced to an external company and Visedo Ltd. has mainly been focusing on domain and product engineering practices.
- Many of the variation strategies and mechanisms presented in this work have already been utilized in the construction of product variants by Visedo Ltd.

Once the current state of the selected software engineering areas was known fields for improvement could be identified. The identified improvement fields were:

- Organizing the SPL directory structure so that it reflects the layered architecture structure. Also preferably implicit dependencies should be avoided to prevent perceived vs. actual architecture drift.
- Adoption of component-based development practices along with the reorganization of SPL directory layout. Creating a component database would improve the management of the core assets.
- SPL development practices are already being utilized, but enforcing architecture and component-based development practices would also promote SPL development practices.
- Adoption of component composition and versioning with associated management scheme and tool support. Also product variants should be placed in a parallel tree from the core assets within the SPL directory hierarchy.

Many of the improvement areas discussed above are related to the organization of the SPL structure which can be mapped into a physical, hierarchical directory. For the software development team it may not always be clear how components depend on each other or to which domain or layer they belong.

In order to make the structure of the SPL more explicit the following attributes should be considered when adding a new component or refactoring an existing one:

- What is the component's main functionality and to what domain does it belong?
    - Affects its domain placement.
- On the services of which component does it depend?
    - Affects its layer placement.

By making the developer explicitly to consider the points above it would possible to:

- Prevent architecture drift.
- Improve architecture and component design practices.
- Improve SPL practices.
- Enable component composition and versioning.

As a consequence, Figure 25 describes a combined SPL structure that maps directly into a directory structure and therefore is able to reflect the structure of the layered architecture, component domain distribution and product variants. Combining different architecture styles was discussed in Chapter 3.4 which served as an inspiration for the combined SPL structure to address following areas:

- SPL project hierarchy structure discussed in Chapter 5.3.1.1.
- Architecture issues discussed in Chapters 5.3.1.3 and 5.4.1.2.
- Component-based development issues discussed in 5.3.2 and 5.4.2.
- SPL issues discussed in Chapters 5.3.3 and 5.4.3.
- Diversity management issues discussed in Chapters 5.3.4 and 5.4.4.

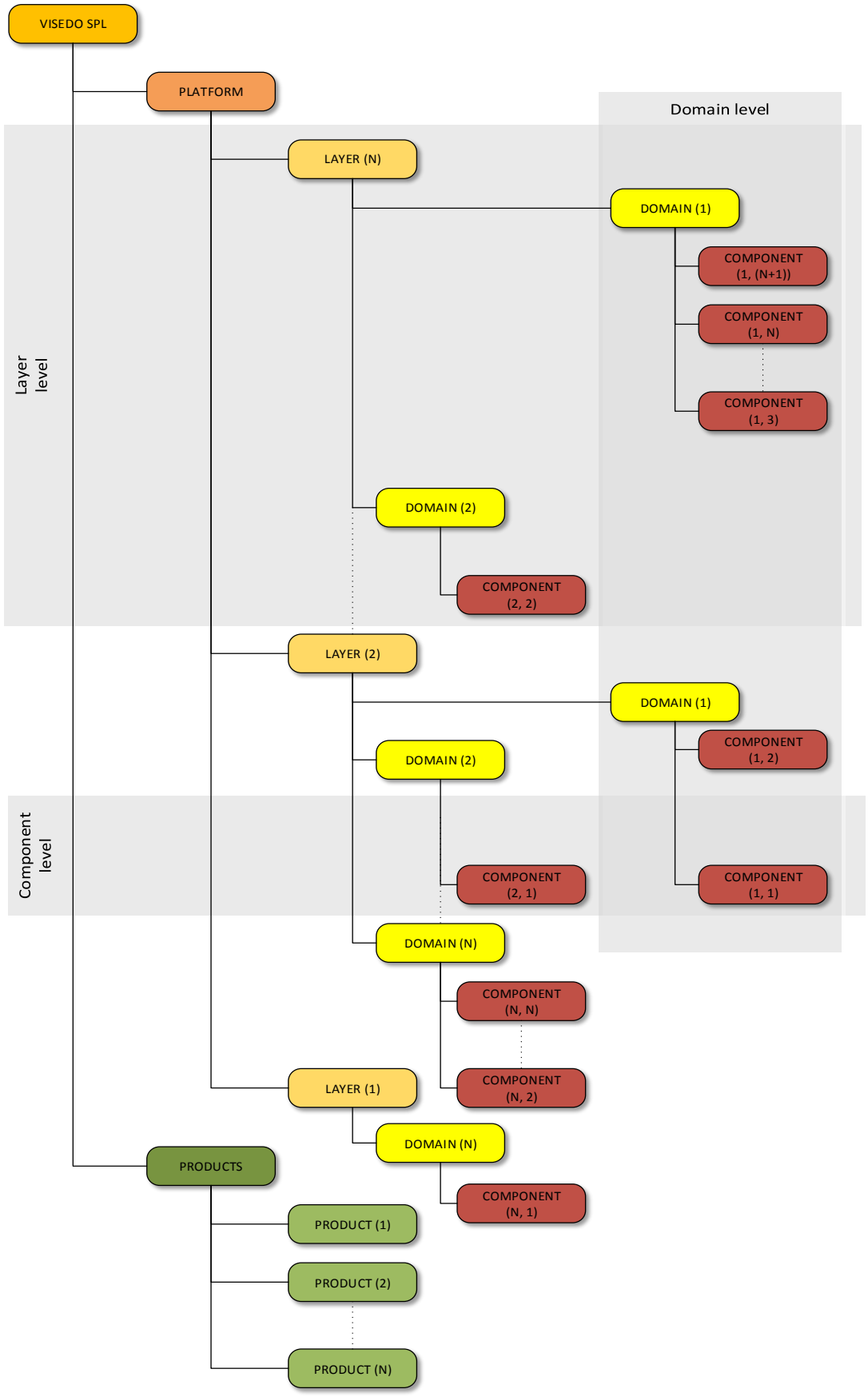

Figure 25. SPL structure for combining layered architecture with components.

The combined SPL structure presented in Figure 25 is so designed that it ties together layered architecture, component domain layout, management of diversity and the composition of actual products. This generic SPL structure has the following properties:

- Adding a new component forces the developer to consider which layer and domain it belongs to by taking into account which services are provided and required by the component.
- Discourages the usage of implicit dependencies and therefore prevents layer dependency leaps.
- Management of diversity can be performed in multiple levels
    - Layer level, large-grained variation where top layers (high-end) can be omitted (low-end) from specific product.
    - Domain level, variation where products can be composed from individual component domains and layers.
    - Component level, fine-grained variation where products can be composed from combination of individual components, component domains or layers.
- Management of products separate from the core assets.

The combined SPL structure is the most significant merit of this work along with the theoretical aspects of architecture styles, component-based development, SPL and management of diversity in software products. The subject matter of this work was demanding, due to its interdisciplinary approach. It is possible that Visedo Ltd. may not be able to absorb all of the improvement steps and guidelines presented earlier, but moving towards the combined SPL structure would probably have the most significant impact on the company's software development practices. Many of the topics discussed in this work are already being employed, in various degrees. In the future it would be interesting to gather various metrics such as:

- What was the impact of the combined SPL structure on product development time?
- How much of the code base is being reused?
- What is the size and growth rate of the code base?
- How complex is the code base?

Comparing the metrics before and after adopting the combined SPL structure would be especially interesting to quantify its impact.

7 CONCLUSIONS

Visedo Ltd. commissioned this work to investigate the current state of their software platform. Based on the findings steps for future improvement should be identified and guidelines on implementing the identified improvement steps. This work consisted of four selected focus areas from the field of software engineering which were:

- Architecture style.
- Component-based development practices.
- Software product line practices.
- Management of diversity.

Each of the selected focus areas was investigated and following observations were made:

- Architecture style is layered but is at times obscured by the implicit dependencies between components thus causing architecture drift.
- Component-based development practices have been utilized, though the extent of design principles used should be reviewed.
- Many of the software product line practices have been utilized from the beginning i.e. core asset and product development practices.
- Most of the variation management techniques presented in this work are already being utilized, with the exception of component composition and versioning.

Looking at the findings of each focus area the following improvement steps and guidelines were suggested:

- Organizing the SPL directory structure in a way that reflects the structure of the architecture and distribution of the component domains.
- Adoption of component-based development practices and creating a component database.
- Enforcing architecture and component-based development practices would promote the SPL development practices also.
- Adoption of component composition and versioning with associated tool support.

As a consequence of the suggested improvement steps and guidelines a combined SPL structure was presented which is also the most significant contribution of this work.

REFERENCES

[1] Deere & Company, “Emissions Brochure,” [Online]. Available: http://www.deere.com/en_US/docs/zmags/engines_and_drivetrain/services_and_support/engine_literature/emissions_brochure.html. [Accessed 11 April 2014].

[2] C. Ebert and C. Jones, “Embedded Software: Facts, Figures and Future,” *Software, IEEE,* vol. 26, no. 3, pp. 14 - 18, 2009.

[3] A. C. Thornton, S. Donnelly and B. Ertan, “More than Just Robust Design: Why Product Development Organizations Still Contend with Variation and its Impact on Quality,” *Research in Engineering Design,* p. 127–143, 2000.

[4] K. P. Günter Halmans, “Communicating the variability of a software-product family to customers,” *Informatik Forschung und Entwicklung,* pp. 113-131, 2004.

[5] P. Clements and L. Northrop, “Basic Ideas and Terms,” in *Software Product Lines: Practices and Patterns*, Addison Wesley, 2002, pp. 5-15.

[6] Oxford University Press, “Oxford Dictionaries,” [Online]. Available: http://oxforddictionaries.com/definition/english/inverter. [Accessed 5 July 2013].

[7] J. H. Hahn, “Modified Sine-Wave Inverter Enhanced,” August 2006. [Online]. Available: http://powerelectronics.com/mag/608PET21.pdf. [Accessed 5 July 2013].

[8] Visedo Ltd., “Visedo Heavy Duty Inverters,” 2013. [Online]. Available: http://www.visedo.com/en/products/heavy-duty-inverters. [Accessed 6 July 2013].

[9] Visedo Ltd., “Electric Drivetrains,” 2013. [Online]. Available: http://www.visedo.com/en/products/electric-drivetrains. [Accessed 6 July 2013].

[10] M. Fowler, “Introduction,” in *Patterns of Enternprise Application Architecture*, Boston, Pearson Education, 2002, pp. 1-15.

[11] Software Engineering Institute, Carnegie Mellon University, “Software Architecture - Glossary,” [Online]. Available: http://www.sei.cmu.edu/architecture/start/glossary/?location=tertiary-

nav&source=19118. [Accessed 17 November 2013].

[12] L. Bass, P. Clements and R. Kazman, Software Architecture in Practice (2nd Edition), Boston: Pearson Education, Inc., 2003.

[13] Microsoft, “What Is Software Architecture?,” in *Microsoft Application Architecture Guide 2nd Edition*, Microsoft Press, 2009, pp. 3-8.

[14] Microsoft, “Architectural Patterns And Styles,” in *Microsoft Application Architecture Guide 2nd Edition*, Microsoft Press, 2009, pp. 19-35.

[15] I. Gorton, “Understanding Software Architecture,” in *Essential Software Architecture (2nd edition)*, Berlin, Springer-Verlag, 2011, pp. 1-15.

[16] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[17] D. Garland and M. Shaw, An Introduction to Software Architecture, Pittsburgh: Carnegie Mellon University, 1994.

[18] J. Savolainen and V. Myllärniemi, “Layered Architecture Revisited - Comparison of Research and Practice,” in *European Conference on Software Architecture*, Cambridge, 2009.

[19] The Linux Foundation, “MeeGo Architecture Layer View,” 22 September 2010. [Online]. Available: https://meego.com/developers/meego-architecture/meego-architecture-layer-view/. [Accessed 17 November 2013].

[20] The Linux Foundation, “Tizen Architecture,” [Online]. Available: https://developer.tizen.org/dev-guide/2.2.1/org.tizen.gettingstarted/html/tizen_overview/tizen_architecture.htm. [Accessed 23 November 2013].

[21] ITU-T, “ITU-T Recommendation X.200,” International Telecommunication Union, 1994.

[22] The Linux Foundation, “MeeGo Architecture Domain View,” 22 September 2010. [Online]. Available: https://meego.com/developers/meego-architecture/meego-

architecture-domain-view. [Accessed 17 November 2013].

[23] S. Thiel and A. Hein, “Systematic Integration of Variability into Product Line Architecture Design,” *Lecture Notes in Computer Science 2379,* pp. 130-153, 2002.

[24] M. Staples and I. Gorton, “Software Product Lines,” in *Essential Software Architecture (2nd edition)*, Berlin, Springer-Verlag, 2011, pp. 219-237.

[25] M. Sinnema, S. Deelstra, J. Nijhuis and J. Bosch, “COVAMOF: A Framework for Modelling Variability in Software Product Families,” *Lecture Notes in Computer Science 3154,* pp. 197-213, 2004.

[26] P. Clements and L. Northrop, “Three Essential Activities,” in *Software Product Lines*, Addison-Wesley, 2002, pp. 29-54.

[27] P. Clements and L. Northrop, “Benefits,” in *Software Product Lines: Practices And Patterns*, Addison-Wesley, 2002, pp. 17-28.

[28] R. T. Vaccare Braga, O. Trindade Junior, K. R. Castelo Branco, L. De Oliveira Neris and J. Lee, “Adapting a Software Product Line Engineering Process for Certifying Safety Critical Embedded Systems,” *Lecture Notes in Computer Science 7612,* pp. 352-363, 2012.

[29] J. Bosch, “Software Product Families in Nokia,” *Lecture Notes in Computer Science 3714,* pp. 2-6, 2005.

[30] D. Durisic, M. Nilsson, M. Staron and J. Hansson, “Measuring the impact of changes to the complexity and coupling properties of automotive software systems,” *The Journal of Systems and Software 86,* pp. 1275-1293, 2013.

[31] A. Polzer, D. Merschen, G. Botteweck, A. Pleuss, J. Thomas, B. Hedenetz and S. Kowalewski, “Managing complexity and variability of model-based embedded software product line,” *Innovations in Systems and Software Engineering 8,* pp. 35-49, 2012.

[32] R. van Ommering and J. Bosch, “Widening the Scope of Software Product Lines - From Variation to Composition,” *Lecture Notes in Computer Science 2379,* pp.

328-347, 2002.

[33] B. P. Lamancha and M. P. Usaola, “Testing Product Generation in Software Product Lines Using Pairwise for Features Coverage,” *Lecture Notes in Computer Science 6435,* pp. 111-125, 2010.

[34] C. W. Krueger, “Variation Management for Software Product Lines,” *Lecture Notes in Computer Science 2379,* pp. 37-48, 2002.

[35] D. E. Perry, “Generic Architecture Descriptions for Product Lines,” in *Lecture Notes in Computer Science 1492*, Springer-Verlag, 1998, pp. 51-56.

[36] P. Clements and L. Northrop, “Requirements Engineering,” in *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002, pp. 109-117.

[37] D. E. Perry, ”Generic Architecture Descriptions for Product Lines.,” *Lecture Notes in Computer Science 1492,* pp. 51-56, 1998.

[38] M. Svahnberg, G. van Gurp and J. Bosch, “A Taxonomy of Variability Realization Techniques,” Blekinge Institute of Technology, Rönneby, 2002.

[39] P. Clements and L. Northrop, “Architecture Definition,” in *Software Product Lines: Practices And Patterns*, Addison-Wesley, 2002, pp. 57-75.

[40] T. van der Storm, “Variability and Component Composition,” CWI, 2004.

[41] R. van Ommering, “Beyond product families: Building a product population?,” in *Software Architectures for Product Families*, Berlin, Springer, 2000, pp. 187-198.

[42] C. Atkison, C. Bunse, C. Peper and H.-G. Gross, “Component-Based Software Development for Embedded Systems - An Introduction,” *Lecture Notes in Computer Science 3378,* pp. 1-7, 2005.

[43] D. Garlan, R. Allen and J. Ockerbloom, “Architectural mismatch: Why reuse is so hard.,” *IEEE Software 12,* no. 6, pp. 17-26, 1995.

[44] Visedo Ltd., “Home,” 2014. [Online]. Available: http://www.visedo.com/en/. [Accessed 28 January 2014].

[45] P. S. Foundation, “Python,” [Online]. Available: https://www.python.org/. [Accessed 22 March 2014].

[46] AT & T Research, “Graphviz - Graph Visualization Software,” [Online]. Available: www.graphviz.org. [Accessed 22 March 2014].

[47] A. R. Hambley, Electrical Engineering: Principles and Applications, sixth edition, Pearson Education, 2014.

[48] H. D. Young and R. A. Freedman, “Direct-Current Circuits,” in *Sears and Zemansky's University Physics with Modern Physics, 11th edition*, Addison Wesley, 2004, pp. 980-1008.

[49] H. D. Young and R. A. Freedman, “Current, Resistance, and Electromotive Force,” in *Sears and Zemansky's University Physics with Modern Physics, 11th edition*, Addison Wesley, 2004, pp. 942-971.

[50] H. D. Young and R. A. Freedman, “Electric Potential,” in *Sears and Zemansky's University Physics with Modern Physics, 11th edition*, Addison Wesley, 2004, pp. 869-897.

[51] H. D. Young and R. A. Freedman, “Alternating Current,” in *Sears and Zemansky's University Physics with Modern Physics, 11th edition*, Addison Wesley, 2004, pp. 1181-1206.

[52] E. Kreyszig, H. Kreyszig and E. J. Norminton, “Fourier Analysis,” in *Advanded Engineering Mathematics, 10th edition*, John Wiley & Sons, 2011, pp. 474-539.

[53] E. W. Weisstein, “Mathworld - A Wolfram Web Resource, Root-Mean-Square,” [Online]. Available: http://mathworld.wolfram.com/Root-Mean-Square.html. [Accessed 8 August 2013].

[54] MapleSoft, “Maple Online Help,” [Online]. Available: http://www.maplesoft.com/support/help/Maple/view.aspx?path=Statistics%2FQuadraticMean. [Accessed 8 August 2013].

[55] A. R. Hambley, “Steady-State Sinusoidal Analysis,” in *Electrical Engineering: Principles and Applications, sixth edition*, Pearson Education, 2014, pp. 227-295.

APPENDIX 1.

DIRECT CURRENT

Direct current (DC) can be defined as a current that does not change its polarity, in other words direction, with time. A bit more elaborate description for current can be obtained from the concept of electrical current which can be defined as the flow of electrical net charge through a circuit element per unit time. [47, 48]

Unit of the electrical charge is called the *coulomb* (C) and the unit of time is the *second* (s). Assuming that $\mathrm{q}(t)$ represents the net charge in *coulombs* in time t then current $\mathrm{i}(t)$ can be written as a ratio of net charge change $\mathrm{dq}(t)$ in unit time period $\mathrm{d}t$ (Equation 1) as:

$$\mathrm{i}(t) = \frac{\mathrm{dq}(t)}{\mathrm{d}t}, \tag{1}$$

where the unit of $\mathrm{i}(t)$ is *coulombs per second* (C/s) also known as *amperes* (A). [47, 49]

Now assuming that the change of net charge $\mathrm{dq}(t)$ is constant with time then current $\mathrm{i}(t)$ will also remain at a constant value k which is illustrated in Figure 26.

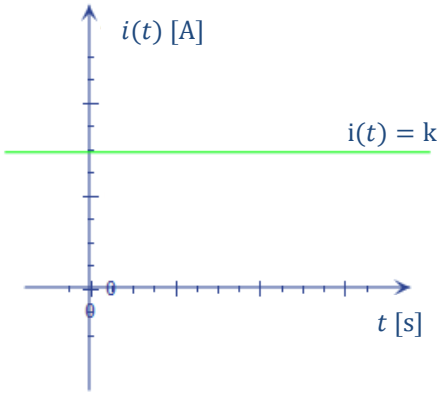

Figure 26. Constant current versus time.

There is of course an infinite number of other DC waveforms that fulfil the requirement of not changing polarity in time, but only the ideal, i.e. constant form was illustrated in Figure 26.

While a current represents how fast an electrical charge is being transferred through circuit elements or conductors, it does not indicate how much energy is being transferred in one unit of charge. To quantify how much energy is being transferred per unit of charge in a circuit from one point to another, *voltage*, also known as electrical potential energy needs to be defined [50].

Unit of voltage is called the *volt* (V) which corresponds to the amount of energy transferred in the *joule* (J) through a circuit element for each coulomb (C) flowing through it $(\mathrm{J/C})$. By examining the units of voltage and current, their product $((\mathrm{J/C}) \cdot (\mathrm{C/S}) = \mathrm{J/S})$ to be exact, it is possible to define rate of the energy transfer, in other words *power* transferred at given time t is:

$$\mathrm{p}(t) = \mathrm{v}(t) \cdot \mathrm{i}(t), \tag{2}$$

where the unit of power $(\mathrm{J/S})$ is also known as *watt* (W). From Equation 2 voltage $\mathrm{v}(t)$ can also be solved in terms of power and current at given time t

$$\mathrm{v}(t) = \frac{\mathrm{p}(t)}{\mathrm{i}(t)}, \tag{3}$$

where the unit is the volt (V). Voltage polarity indicates the direction of the energy flow. Depending on the assignment of positive and negative voltage references of the circuit element and the true direction of the current through the circuit element the direction of the energy transfer can be defined. If the current is flowing from the positive voltage reference to the negative then energy is being absorbed by the element. On the other hand, if the current is flowing in the opposite direction, from negative to positive, then the circuit element is supplying energy. In other words, result of the power calculation (Equation 2) for a circuit element correlates to whether a circuit element is supplying energy or not. [47]

For example, Figure 27 illustrates a circuit element where the positive $(+)$ voltage reference has been assigned to the top end and the negative voltage $(-)$ reference to the bottom end. The reference direction of the current $\mathrm{i}(t)$ is from positive to negative voltage. Now if the circuit element is supplying energy to the circuit through points a and b then the true direction of the current $\mathrm{i}(t)$ is the same as the reference direction (positive power). On the other hand, if the circuit element is absorbing energy then the direction of the current $\mathrm{i}(t)$ must flow opposite to the reference direction (negative power).

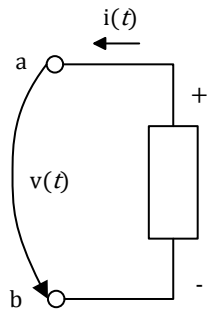

Figure 27. Circuit element with positive and negative references.

To summarise, the polarities of the voltages and the true direction of the currents are important for defining the direction of the energy flow in circuit elements [47]. To be more precise, energy in an electrical circuit may be converted to other forms of energy, such as heat, chemical or mechanical (e.g. in an electric motor) energy but not lost as such [47, 49].

DC voltages are voltages that are constant with time [47]. Figure 28 depicts an ideal constant negative voltage k as a function of time.

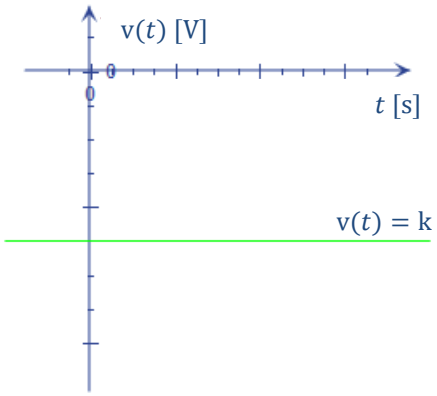

Figure 28. Constant voltage versus time.

There is also an infinite number of voltage waveforms in which the polarity of the waveform does not change that can be considered DC voltages, but only the ideal constant DC voltage waveform will be covered here.

The final element for establishing relationships between current, voltage, power and energy is the definition of energy. *Energy* is the average power multiplied by time [48]. A more general definition involves taking integral of power $\mathrm{p}(t)$ regards to time t from t_0 to t_1 (Equation 4) which is:

$$\mathrm{w}(t) = \int_{\mathrm{t}_0}^{\mathrm{t}_1} \mathrm{p}(t)\mathrm{d}t, \tag{4}$$

where the unit of energy is the joule (J) [47].

ALTERNATING CURRENT

The concept of an alternating current (AC) has been briefly covered in the previous chapter. Now the concept will be covered in more detail. Given that a direct current is constant with time, an alternating current varies with time, reversing its direction periodically. Typical waveforms of an alternating current are: sinusoidal, triangular and square. [47, 51].

This section is restricted to examining only the sinusoidal waveform as other periodic waveforms can be constructed with a Fourier series consisting of an infinite number of cosines and sines summed together as function of e.g. time [52]. A sinusoidal alternating current can be defined as the product of the current's peak value $\mathrm{I}_{\mathrm{PEAK}}$ and the periodic function of time t such as the cosine function [51]. The equation for a sinusoidal alternating current at given time t (Equation 5) is:

$$\mathrm{i}(t) = \mathrm{I}_{\mathrm{PEAK}}\cos(\omega t + \theta), \tag{5}$$

where $\mathrm{I}_{\mathrm{PEAK}}$ (A) is the maximum value of current $\mathrm{i}(t)$ (s) during period T (s), ω (rad/s) is the angular velocity and θ (rad) is the fixed angle offset i.e. phase angle [47, 51].

Since cosine and sine functions both complete one period at an angle (distance) of 2π radians, then the following (Equation 6) must hold:

$$\omega\mathrm{T} = 2\pi, \tag{6}$$

where angular velocity ω can be obtained. Thus angular velocity is the ratio of 2π radians within a period of time T (Equation 7):

$$\omega = \frac{2\pi}{\mathrm{T}}. \tag{7}$$

Angular velocity ω can also be considered as an angular frequency or speed. The equation of a sinusoidal alternating voltage can be expressed analogously with the current (Equation 8) as:

$$\mathrm{v}(t) = \mathrm{V_{PEAK}}\cos(\omega t + \theta), \tag{8}$$

where $\mathrm{V_{PEAK}}$ (V) is the peak voltage value of instantaneous voltage $\mathrm{v}(t)$ (s). [47]

An example of a sinusoidal alternating current waveform is depicted in Figure 29 which would be similar to sinusoidal AC voltage waveform, provided that scalar values of: $\mathrm{V_{PEAK}}$, ω and θ would correspond accordingly.

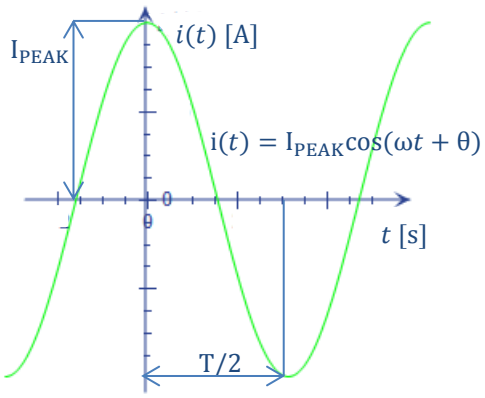

Figure 29. Sinusoidal alternating current.

Electrical component datasheets contain a number of different electrical attributes for the component such as the electrical output capabilities of the component. For example, an inverter's datasheet may contain ratings for peak AC voltage, AC current and power, but also the *Root-Mean-Square* (RMS) values of the peak values. General RMS definition of a periodic function $\mathrm{f}(t)$ over a time interval $[\mathrm{T_1}, \mathrm{T_2}]$ is presented in Equation 9 [53].

$$\mathrm{f_{RMS}} = \sqrt{\frac{1}{\mathrm{T_2}-\mathrm{T_1}}\int_{\mathrm{T_1}}^{\mathrm{T_2}}[\mathrm{f}(t)]^2\,\mathrm{d}t} \tag{9}$$

Alternatively, the RMS is referred to as quadratic mean [53, 54]. Applying Equation 9 to calculate the RMS current $\mathrm{I_{RMS}}$ over a period T would correspond (Equation 10) to:

$$\mathrm{I_{RMS}} = \sqrt{\frac{1}{\mathrm{T}}\int_0^{\mathrm{T}}[\mathrm{i}(t)]^2\mathrm{d}t} = \sqrt{\frac{1}{\mathrm{T}}\int_0^{\mathrm{T}}[\mathrm{I_{PEAK}}\cdot\cos(\omega t + \theta)]^2\mathrm{d}t}, \tag{10}$$

where $\mathrm{i}(t)$ is equivalent to Equation 5. Performing the integral calculation in Equation 10, applying the integration limits and finally reducing the terms, $\mathrm{I_{RMS}}$ current can be obtained (Equation 11) for sinusoidal current [47, 51].

$$\mathrm{I_{RMS}} = \frac{\mathrm{I_{PEAK}}}{\sqrt{2}} \tag{11}$$

In Figure 30 the RMS current is depicted in relation to the sinusoidal current from which it was calculated.

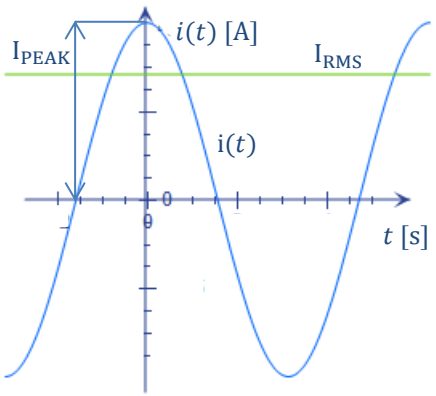

Figure 30. Sinusoidal current and equivalent RMS current.

Similarly the RMS value ($\mathrm{V_{RMS}}$) can be calculated for sinusoidal voltage which is presented in Equation 12 [47, 51].

$$\mathrm{V_{RMS}} = \frac{\mathrm{V_{PEAK}}}{\sqrt{2}} \tag{12}$$

Since RMS values for a sinusoidal current and voltage have been defined, the average power of a circuit consisting of arbitrary combination of resistors, inductors and capacitors can be examined. Looking at the phase angle θ in Equation 5 and Equation 8 it can be concluded that the phase angle θ is not always the same for instantaneous current $\mathrm{i}(t)$ and voltage $\mathrm{v}(t)$ with reference to time t [51]. So far it has been assumed that the phase angle θ is equal in both current $\mathrm{i}(t)$ and voltage $\mathrm{v}(t)$ which holds true for resistors but not for inductors and capacitors. In reality, though, the voltage phase angle θ relative to the current depends on the properties of the circuit component. Ideal circuit component properties for each type are summarized in Table 27 [51].

Table 27. The phase angles of the voltage of the circuit component voltage, relative to current.

Component	Phase angle θ	Description
Resistor	0°	Voltage and current are in phase
Inductor	+90°	Voltage leads current
Capacitor	-90°	Voltage trails current

Looking at the Table 27 one could deduce that voltage $\mathrm{v}(t)$, with a phase angle θ, across an arbitrary RLC AC circuit respect to current $\mathrm{i}(t)$, would hold as stated in Equation 8 [51]. However, since voltage $\mathrm{v}(t)$ can have any phase angle from -90° to +90° with respect to current $\mathrm{i}(t)$, Equation 5 can be simplified into the following (Equation 13) form:

$$\mathrm{i}(t) = \mathrm{I}_{\mathrm{PEAK}}\cos(\omega t). \tag{13}$$

Given Equations 8 and 13, instantaneous power $\mathrm{p}(t)$ can be calculated for an arbitrary RLC AC circuit which is presented in Equation 14 [51].

$$\mathrm{p}(t) = \mathrm{v}(t) \cdot \mathrm{i}(t) = \mathrm{V}_{\mathrm{PEAK}}\cos(\omega t + \theta) \cdot \mathrm{I}_{\mathrm{PEAK}}\cos(\omega t) \tag{14}$$

Applying trigonometric identity (Equation 15)

$$\cos(\omega t + \theta) = \cos(\omega t)\cos(\theta) - \sin(\omega t)\sin(\theta) \tag{15}$$

to Equation 14 and simplifying $\mathrm{p}(t)$ to (Equation 16):

$$\mathrm{p}(t) = \mathrm{V}_{\mathrm{PEAK}}\mathrm{I}_{\mathrm{PEAK}}[\cos^2(\omega t)\cos(\theta) - \cos(\omega t)\sin(\omega t)\sin(\theta)] \tag{16}$$

followed by the substitutions of Equation 17

$$\cos^2(\omega t) = \frac{1}{2}(1 + \cos(2\omega t)) \tag{17}$$

and Equation 18

$$\cos(\omega t)\sin(\omega t) = \frac{1}{2}\sin(2\omega t) \tag{18}$$

into Equation 16, instantaneous power $\mathrm{p}(t)$ can be expressed (Equation 19) [51, 55].

$$\mathrm{p}(t) = \frac{\mathrm{V_{PEAK}} \cdot \mathrm{I_{PEAK}}}{2} [(1 + \cos(2\omega t)) \cos(\theta) - \sin(2\omega t) \sin(\theta)] \quad (19)$$

To calculate the average power $\mathrm{P_{AVG}}$ through an arbitrary RLC AC circuit element it is sufficient to notice that the average of terms $\cos(2\omega t)$ and $\sin(2\omega t)$ over a period is zero and thus Equation 20 can be derived for the average power [51, 55].

$$\mathrm{P_{AVG}} = \frac{\mathrm{V_{PEAK}} \cdot \mathrm{I_{PEAK}}}{2} \cos(\theta) = \mathrm{V_{RMS}} \mathrm{I_{RMS}} \cdot \cos(\theta) \quad (20)$$

Figure 31 represents an example of an arbitrary AC circuit element E consisting of resistors $\mathrm{R1}$ and $\mathrm{R2}$, inductance $\mathrm{L1}$, capacitors $\mathrm{C1}$ and $\mathrm{C2}$. Average power through the circuit element E can be calculated as a product of the RMS values: current $\mathrm{i_{ab}}$, voltage $\mathrm{v_{ab}}$ and phase angle θ.

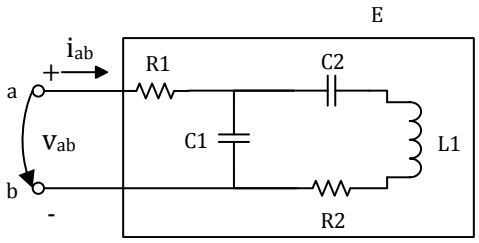

Figure 31. An AC circuit element E consisting of arbitrarily placed circuit components.

APPENDIX 2. (scan_link_info_xml.py)

```
'''
Created on 8.3.2014

@author: tommi.kankaanranta@visedo.fi
@file: scan_link_info_xml.py
'''

import sys, traceback
from optparse import OptionParser
from optparse import OptionGroup

DEFAULT_PATH = "None"
DEFAULT_FILE = "None"

def handle_path(xml_input_file):
    result = DEFAULT_PATH
    xml_path = xml_input_file.find("path")

    # Check whether input file path was available or not and act accordingly.
    if xml_path != None:
        result = xml_path.text
    return result

def handle_file(xml_input_file):
    result = DEFAULT_FILE
    xml_file = xml_input_file.find("file")
    xml_kind = xml_input_file.find("kind")

    # Check whether input file type was available or not and act accordingly.
    if xml_kind.text != "object":
        result = xml_file.text
    return result

def extract_symbols(xml_symbols):
    symbols = map(lambda x: ( x.find("name").text, { "oc-idref":
x.find("object_component_ref").get("idref") } ), xml_symbols)
    return symbols

def extract_object_components(xml_object_components):
    object_components = map(lambda x: ( x.get("id"), { "if-idref":
x.find("input_file_ref").get("idref") } ), xml_object_components)
    return dict(object_components)

def extract_input_files(xml_input_files):
    input_files = map(lambda x: ( x.get("id"), { "kind": x.find("kind").text,
"name": x.find("name").text, "file": handle_file(x), "path": handle_path(x) }
), xml_input_files)
    return dict(input_files)

def process_linking_info_xml(filepath):
    import xml.etree.ElementTree as ET

    # Parse the document at given filepath.
    tree = ET.parse(filepath)
```

(continued)

APPENDIX 2. (continued)

```
    # Get the root element.
    root = tree.getroot()

    # Read symbols from "symbol_table".
    symbols = [child for child in root.iter() if child.tag == "symbol"]

    # Find symbols that are associated with object files.
    symbols = filter(lambda x: x.find("object_component_ref") != None,
symbols)

    # Read object components from "object_component_list".
    object_components = [child for child in root.iter() if child.tag ==
"object_component"]

    # Find object components that are associated with input files.
    object_components = filter(lambda x: x.find("input_file_ref") != None,
object_components)

    # Read input files from "input_file_list".
    input_files = [child for child in root.iter() if child.tag ==
"input_file"]

    # Extract symbols.
    sym = extract_symbols(symbols)
    # Extract object_components.
    obc = extract_object_components(object_components)
    # Extract input files.
    inf = extract_input_files(input_files)

    # Add to database each symbol and corresponding component file
information
    # (i.e. compilation unit) where the symbol was defined.
    db = map(lambda x: ( x[0], obc[x[1]["oc-idref"]] ), sym)
    # Replace component file information with more detailed information
regarding
    # the actual input file.
    db = map(lambda x: ( x[0], inf[x[1]["if-idref"]] ), db)
    # Construct the final database by constructing a list of tuples
containing:
    #  (symbol, object_file_name, library_name, kind, path)
    db = map(lambda x: ( x[0], x[1]["name"], x[1]["file"], x[1]["kind"],
x[1]["path"] ), db)

    # Sort by archive name column.
    results = sorted(db, key = lambda x: ( x[2] ))

    # Print out the flat results database.
    for result in results:
        print "{0:65}, {1:45}, {2:50}, {3:10}, {4:50}".format(*result)

def main():
    parser = OptionParser(
        usage="usage: %prog [options] <path/to/linking_info_xml>",
        version="%prog 1.0")
```

(continued)

APPENDIX 2. (continued)

```
    # Supported options
    parser.add_option(
            "-d", "--default-archive", type="string", default="None",
dest="default-archive",
            help="default archive name for loose object files",
            metavar="ARCHIVE")

    # The actual "main" program.
    code = 0
    try:
        # Parse command line arguments.
        (options, args) = parser.parse_args()
        options = vars(options)

        # Default file name for archive typed input files that aré empty.
        global DEFAULT_FILE
        DEFAULT_FILE = options["default-archive"]

        # Define linker XML-file path variable.
        linking_info_xml_filepath = ""
        try:
            # Extract linker XML-file path.
            linking_info_xml_filepath = args[0]
        except IndexError:
            # Mandatory command line arguments weren't supplied => print
usage instructions.
            parser.print_help()
        else:
            # Process the actual XML-file produced by the linker.
            process_linking_info_xml(linking_info_xml_filepath)
    except:
        # Print stack trace.
        traceback.print_exc()
        code = -1
    finally:
        # Set program exit code.
        sys.exit(code)

if __name__ == '__main__':
    main()
```

APPENDIX 3. (scan_cross_reference_listings.py)

```
'''
Created on 10.3.2014

@author: tommi.kankaanranta@visedo.fi
@file: scan_cross_reference_listings.py
'''
import os, sys, traceback
from optparse import OptionParser
from optparse import OptionGroup

FILENAME_EXTENSION = ".crl"

def build_columns(symbol, access, line, column, *args):
    return (symbol, args[1], args[0], access, line, column)

def find_crl_files(path):
    crl_files = []

    # Search *.crl files from root and root's sub-folders.
    for root, _, files in os.walk(path):
        files = filter(lambda x: x[-len(FILENAME_EXTENSION):] ==
FILENAME_EXTENSION, files)

        # Add found *.crl files to list to be processed for the symbols.
        if len(files) > 0:
            map(lambda x: crl_files.append(os.path.join(root, x)), files)

    return crl_files

def process_crl_file(crl_file):
    lines = []

    # Process current crl-file per line basis.
    with open(crl_file) as fp:
        for line in fp:
            lines.append(line.split())

    # Organize each symbols data into columns based on where it has been:
Reference, Modifed or Accessed.
    lines = map(lambda x: build_columns(x[1], x[2], x[4], x[5],
*os.path.split(x[3])), filter(lambda x: x[2] in "RMA", lines))

    # Output the symbol access data to standard output.
    for line in lines:
        print "{0:65}, {1:45}, {2:65}, {3:10}, {4:5}, {5:5}".format(*line)

def process_crl_files(crl_files):
    # Process each crl-file.
    map(process_crl_file, crl_files)

def main():
    parser = OptionParser(
        usage="usage: %prog [options]
<path/to/cross_reference_listing_files_directory>",
```

(continued)

APPENDIX 3. (continued)

```
        version="%prog 1.0")
    # The actual "main" program.
    code = 0
    try:
        # Parse command line arguments.
        (_options, args) = parser.parse_args()

        # Define cross reference listings root path variable.
        crl_files_root_directory = ""
        try:
            # Extract cross reference listings root path.
            crl_files_root_directory = args[0]
        except IndexError:
            # Mandatory command line arguments weren't supplied => print
usage instructions.
            parser.print_help()
        else:
            # Process the actual XML-file produced by the linker.
            process_crl_files(find_crl_files(crl_files_root_directory))
    except:
        # Print stack trace.
        traceback.print_exc()
        code = -1
    finally:
        # Set program exit code.
        sys.exit(code)

if __name__ == '__main__':
    main()
```

APPENDIX 4. (merge_scan_results.py)

```
'''
Created on 11.3.2014

@author: tommi.kankaanranta@visedo.fi
@file: merge_scan_results.py
'''

import os, sys, traceback
from optparse import OptionParser
from optparse import OptionGroup

def read_linking_listing(linking_listing_filepath):
    db = {
        "symbols": {
        },
        "objects": {
        }
    }

# Database is organized as follows:
#     {
#         "symbols": {
#             "<symbol_nameX>": "object_nameX",
#         },
#         "objects": {
#             "<object_nameX>": "archive_nameX"
#         }
#     }

    # Read the linking info database per line basis.
    with open(linking_listing_filepath) as fp:
        for line in fp:
            # Construct a tuple of the comma separated columns.
            (sym, obj, arch, _kind, _path) = map(lambda x: x.strip(),
line.split(","))
            db["symbols"][sym] = obj
            db["objects"][obj] = arch

    return db

def read_crl_listing(crl_listing_filepath, linking_db):
    # Read the cross reference listing database per line basis.
    with open(crl_listing_filepath) as fp:
        for line in fp:
            # Extract symbol, object file, access type, row and column
information
            # regarding the symbols usage.
            (sym, obj, _path, _access, row, column) = map(lambda x:
x.strip(), line.split(","))
            dst_sym = "_" + sym
            src_obj = obj + ".obj"
            # Output the merged data to standard output.
            if dst_sym in linking_db["symbols"] and src_obj in
linking_db["objects"]:
```

(continued)

APPENDIX 4. (continued)

```
                dst_obj = linking_db["symbols"][dst_sym]
                dst_arch = linking_db["objects"][dst_obj]
                src_arch = linking_db["objects"][src_obj]
                print "{src_archive}, {src_object}, {src_row}, {src_column},
{dst_symbol}, {dst_object}, {dst_archive}".format(src_archive=src_arch,

src_object=obj,

src_row=row,

src_column=column,

dst_symbol=dst_sym,

dst_object=dst_obj,

dst_archive=dst_arch)

def merge_linking_and_crl_listings(file_paths):
    # Extract specific file paths.
    (linking_listing_filepath, crl_listing_filepath) = file_paths

    # Construct internal linking symbol database.
    linking_db = read_linking_listing(linking_listing_filepath)

    # Merge the linking symbol database with cross reference listing
database.
    read_crl_listing(crl_listing_filepath, linking_db)

def main():
    parser = OptionParser(
        usage="usage: %prog [options] <path/to/linking_info_listing_file>
<path/to/cross_reference_listing_file>",
        version="%prog 1.0")

    # The actual "main" program.
    code = 0
    try:
        # Parse command line arguments.
        (_, args) = parser.parse_args()

        # Define input file paths variable.
        file_paths = ()
        try:
            # Extract linking info and cross reference listings database
paths.
            file_paths = (args[0], args[1])
        except IndexError:
            # Mandatory command line arguments weren't supplied => print
usage instructions.
            parser.print_help()
        else:
            # Do the actual merging of linking info and cross reference
listing database files.
```

(continued)

APPENDIX 4. (continued)

```
            merge_linking_and_crl_listings(file_paths)
    except:
        # Print stack trace.
        traceback.print_exc()
        code = -1
    finally:
        # Set program exit code.
        sys.exit(code)

if __name__ == '__main__':
    main()
```

APPENDIX 5. (draw_scan_results.py)

```
'''
Created on 11.3.2014

@author: tommi.kankaanranta@visedo.fi
@file: draw_scan_results.py
'''

import sys, traceback
import networkx as nx
from optparse import OptionParser
from optparse import OptionGroup

def process_merged_reference_listing(merged_file_listing_path,
output_dot_filename):
    # Initialize directed graph.
    G = nx.DiGraph()

    # Initiliaze the graph edges list variable.
    edges = []
    with open(merged_file_listing_path) as fp:
        for line in fp:
            # Produce edge based on the symbols source usage archive and
            # its definition archive i.e. the "destination" of the
dependency.
            (src_arch, _, _, _, _, _, dst_arch) = map(lambda x: x.strip(),
line.split(","))
            edges.append((src_arch, dst_arch))

    # Filter out archive's internal references.
    edges = filter(lambda x: x[0] != x[1], edges)

    # Add nodes and edges of the graph.
    G.add_edges_from(set(edges))

    # Write Graphviz DOT-output file.
    nx.write_dot(G, output_dot_filename)

def main():
    parser = OptionParser(
        usage="usage: %prog [options]
<path/to/merged_reference_listing_file>",
        version="%prog 1.0")

    # Supported options
    parser.add_option(
            "-o", "--output-dot-filename", type="string",
default="output.dot", dest="output-dot-filename",
            help="search recursively from root directory and its sub-
directories for *.crl files to be processed",
            metavar="FILE")

    # The actual "main" program.
    code = 0
    try:
```

(continued)

APPENDIX 5. (continued)

```
        # Parse command line arguments.
        (options, args) = parser.parse_args()
        options = vars(options)

        # Initialize path and filename variables.
        merged_file_listing_path = ""
        output_dot_filename = options["output-dot-filename"]
        try:
            # Extract the merged database path.
            merged_file_listing_path = args[0]
        except IndexError:
            # Mandatory command line arguments weren't supplied => print
usage instructions.
            parser.print_help()
        else:
            # Process the actual merged database file.
            process_merged_reference_listing(merged_file_listing_path,
output_dot_filename)
    except:
        # Print stack trace.
        traceback.print_exc()
        code = -1
    finally:
        # Set program exit code.
        sys.exit(code)

if __name__ == '__main__':
    main()
```