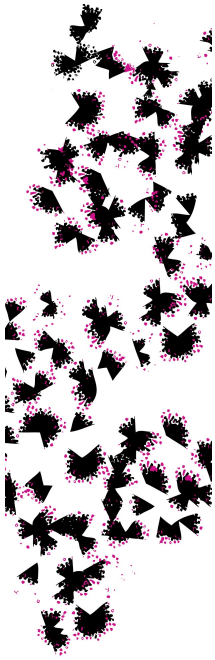
RAM ROBOTICS AND MECHATRONICS



Writing reusable code for robotics

D.H. (Dennis) Ellery

MSc Report

Committee:
Dr.ir. J.F. Broenink
Dr.ir. D. Dresscher
Dr. M. Poel

December 2017

051RAM2017
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE.

MIRA CTIT
BIOMEDICAL TECHNOLOGY AND TECHNICAL MEDICINE

Summary

In the i-Botics project about intuitive robotics, reusable software is desired to speed up and simplify robot software design. Nowadays, software is created for a specific application, reducing its reuse in multiple robotic applications. Functional source code of the implementation can be re-used in future projects by writing in a modular way. The goal of this thesis is to write reusable software for robotic applications that is language and platform independent.

Major topics for reusable software are: documentation, extensibility, intellectual property issues, modularity, packaging, portability, standards compliance, support and verification and testing. (Marshall et al., 2010). To quantify the level of reuse, the reuse readiness levels (RRL) are used. Requirements for a reusable component were defined from the nine topics and their RRL.

There are multiple characteristics of software writing that make software reusable. A selection of characteristics is made to create a paradigm for writing reusable software. A look into three main paradigms was taken: object-oriented programming, component-based software framework (CSBF) and the separation of 5 concerns.

Existing examples of reusable software using these paradigms are middleware. Middleware is software that connects individual components. Each component is a standalone object, which uses the object-oriented paradigm. The middleware defines the composition, coordination and communication. The configuration and computation is written by the user. The separation of 5 concerns in software writing allows for simple replacement of a single concern, without needing to adjust another. This allows adding pre-written computational software into generated middleware software.

From design paradigms, model-driven design is used to create a design in a high level model, which is then generated as a middleware hierarchy of "empty" components, not containing any computational and/or configurable software. After reviewing the most common CBSF middleware, it turns out that a finite state machine (FSM) is used in each component. The connection between middleware and a general component is achieved using an interface, which allows the middleware to use a general component without having to know how it is implemented. By writing the computational component in a reusable and general way, the computational and configurable software can be reused in different projects. Following the structure of the FSM, easy integration of a general component into middleware is achieved.

A demonstrator was designed, by writing a general component that achieved the requirements set by the RRL. This component was then integrated into a generated middleware and tested. The same component was integrated into a second middleware to determine if it was platform independent. Both times, the component worked as expected.

It is concluded that the design approach to create pre-written components that can be integrated into middleware is viable. These components can be reused in other projects, that use different middleware, running on different hardware.

It is recommended that the designed work flow needs to be used by other users in other applications. Feedback received from the users, should be used to improved the work flow.

Samenvatting

Voor het i-Botics project over intuïtieve robotica is het gewenst om herbruikbare code te schrijven om sneller en simpeler software te schrijven. Code wordt vaak geschreven voor een specifieke applicatie, waardoor de kans op hergebruik wordt verminderd. Functionele code kan worden hergebruikt in toekomstige projecten door te schrijven op een modulaire manier. Het doel van deze thesis is om code te schrijven voor robotische applicaties die taal en platform onafhankelijk zijn.

Hoofdaspecten voor herbruikbare code zijn: documentatie, uitbreidbaarheid, eigendoms overeenkomsten, modulariteit, verpakking, overdraagbaarheid, normen naleven, ondersteuning en testen (Marshall et al., 2010). Om het niveau van hergebruik te kwantificeren wordt gebruik gemaakt van de "reuse readiness levels"(RRL). Vanuit de hiervoor beschreven aspecten en hun RRL is een pakket van eisen opgesteld.

Er zijn meerdere kenmerken in het schrijven van code die ervoor zorgen dat code herbruikbaar is. Een selectie van deze kenmerken creëren samen een paradigma voor het schrijven van herbruikbare code. De drie meest gebruikte programmeer paradigmas zijn bekeken: object-georiënteerd programmeren, 'component-based software framework' (CSBF) en de scheiding van 5 belangen.

Bestaande voorbeelden van herbruikbare code die gebruik maken van deze paradigmas zijn middleware. Middleware is software die individuele componenten verbindt. Elke component is een op zichzelf staand object die het object-georiënteerd paradigma gebruikt. De middleware definieert de belangen compositie, coördinatie en communicatie, terwijl de belangen configuratie en computatie door de gebruiker geschreven worden. Het scheiden van de 5 belangen in code schrijven zorgt ervoor dat een belang aangepast kan worden zonder een ander belang te beïnvloeden. Dit maakt het mogelijk om vooraf geschreven computatie code in te voegen bij gegenereerde middleware code.

Van de ontwerp paradigmas wordt 'model-driven design' gebruikt om te kunnen ontwerpen in een abstract model, waarvan de middleware structuur gegenereerd wordt uit "lege" componenten. Deze componenten bevatten geen computatie of configuratie code.

Na onderzoek van de meest gebruikte CBSF middleware werd duidelijk dat een 'Finite state machine' (FSM) gebruikt wordt in elke component. De connectie tussen middleware en een generieke component wordt bereikt met een interface, welke ervoor zorgt dat een generieke component gebruikt kan worden door de middleware zonder dat deze weet hoe het geïmplementeerd is. Door het schrijven van de computatie component in een herbruikbare en generieke wijze, kunnen de computatie en de configuratie code hergebruikt worden in andere projecten. Eenvoudige integratie van de generieke component in de middleware wordt bereikt door het volgen van de FSM structuur.

Een demonstratie is ontworpen, waarbij de generieke component aan het pakket van eisen van RRL voldoet. Deze component is geÃńntegreerd in gegenereerde middleware en vervolgens getest. Dezelfde component is ook geïntegreerd in een tweede middleware om te bepalen of de component platform onafhankelijk is. In beide gevallen werkte de component zoals voorspeld was.

Hieruit wordt geconcludeerd dat de de hiervoor beschreven benadering voor het ontwerpen van vooraf geschreven componenten geïntegreerd kan worden in middleware.

Het wordt aanbevolen dat de ontworpen manier van werken door andere gebruikers in andere toepassingen moet worden gebruikt. Terugkoppeling van informatie van de gebruikers moet worden gebruikt om de workflow te verbeteren.

Contents

Acronyms

5C Separation of 5 concerns

API Application programming interface

BCM BRICS Component Model

BRICS Best Practice in Robotics

BRIDE BRICS Integrated Development Environment

CBSD Component-Based Software Development

CBSF Component-Based Software Framework

CPC Component-Port-Connector

CSP Communicating Sequential Processes

FSM Finite State Machine

GAC Generic Architecture Component

IDE Integrated Development Environment

LUNA LUNA Universal Networking Architecture

MDD Model-Driven Design

OMG Object Management Group

OOP Object-oriented programming

OROCOS Open Robot Control Software

OS Operating System

RaM Robotics and Mechatronics

RRL Reuse Readiness Level

ROS Robotic Operating System

TNO Nederlandse Organisatie voor toegepast-natuurwetenschappelijk onderzoek

TERRA Twente Embedded Real-time Robotic Application

UML Universal Modelling Language

1 Introduction

1.1 Context

Robots are getting smarter and smarter, but cannot replace humans at the moment. However, with the current technology the human mind can be combined with the robotic strength, by letting humans control robots. The robot then becomes an extension of the human operator, being able to perform tasks in dangerous environments.

i-Botics is an open innovation centre for research and development in interaction robotics, founded by Nederlandse Organisatie voor toegepast-natuurwetenschappelijk onderzoek (TNO) and the University of Twente and aims at developing knowledge and technology for value adding robotic solutions.

The two main research lines are tele-robotics and exoskeletons. This MSc project focusses on intuitive control of a tele-operated system.

In i-Botics, it is important to create software in a modular and reusable way, such that functional software of the implementation can be re-used in later projects. By standardising the implementation of modular software from the start, more time can be invested into the actual implementation instead of having to rewrite often recurring functionality.

1.2 Problem description

In robotics, no two robotic applications are the same, because they have been designed with a different purpose and goal in mind. As a result, the hardware they use and the way they are constructed are considerably different.

Due to differences in hardware, design and application, the software to control the robots differs. This results in a different software structure. With these differences it is very difficult to write a piece of software that is able to run on multiple robots.

The ideal way of creating software for a robot would be combining existing “blocks” of software together, resulting in a fully functioning system. When a piece of hardware changes, all that has to be done is to interchange the “block” of software. This is a high level modular approach to write software, known as Model-Driven Design (MDD).

Middleware is used to “glue” blocks of software together, creating a framework wherein a user can write his own software blocks. It can be defined as: “middleware are designed to manage the heterogeneity of the hardware, improve software application quality, simplify software design, and reduce development costs” (Elkady and Sobh, 2012).

Different middleware are implemented in different ways. As a result, software written in one middleware cannot be copied to another middleware without adaptation.

To simplify software development for robotics, reuse of existing software is preferable, but inherently difficult. As stated before, existing software is written for a specific robot with a specific task, making reuse of its software difficult in a different environment.

A point of interest within robotics software is the design of a software block. With different approaches in middleware and software design, the implementation of a block is different.

Different types of block implementations, and how to write generalised implementations to run within a block, are reviewed in this thesis.

1.3 Goals

To simplify the ability of writing and using reusable software in robotics, the goal is defined as:

- Write reusable software for robotic applications that is language and platform independent.

The main goal is divided into the following sub goals:

- Find out how reusable software is currently used in robotics.
- Identify the main problem/difficulty of creating reusable software in robotics.
- Identify which paradigms for reusable software exist.
- Define requirements for reusable software.
- Demonstrate that reusable software can run on different platforms.

The main result of this thesis will be a work flow to develop reusable software. This means that writing software for robotic applications is expressed in a standardised form, that is platform and language independent.

1.4 Outline

In chapter 2 the problem is analysed and dissected into smaller pieces. From the analysis information is obtained on how best to write a reusable piece of software. In chapter 3 the design and implementation of the methodology to write reusable software is discussed. The resulting software is then evaluated in chapter 4. Finally the conclusion and recommendations are given in chapter 5.

2 Analysis

For many years, the idea of creating and reusing software has been a goal in the software community. This is to reduce the time needed to "reinvent" existing implementations of software components. However, this has not been achieved yet. In this chapter, a deeper look is taken into what makes software reusable, and what methodologies and paradigms exist for developing reusable software.

2.1 Reusability of software

One of the difficulties of writing reusable software is that it is not normally part of the requirements of the project. The main focus of the project is to get this particular robot/system working, not taking into account the fact that the software is going to be reused in the future.

Brugali and Scandurra (2009) state that the reasons of using reusable software can be generally divided into two categories:

- Opportunistic – While starting a project, the team realises that there are existing components that they can use as a starting point. With similar functionality, the source code does not need to be written from scratch.
- Planned – A team strategically designs components so that they will be reusable in future projects.

For a new project one can use existing source code to speed up their initial startup time. Based on this initial source code, software for a specific application is written. If the software for a specific application is not written in a structured, modular way, a desired change could result in the necessity to rewrite code in multiple places.

A team that starts planning to reuse their software will have a longer startup time, because the structure of the reusable code needs to be defined. However, the software is wider applicable and can be reused in other projects, saving time in future projects.

When choosing to reuse software, there is a choice between:

- Internal reuse – A team reuses its own components.
- External reuse – A team may choose to use a third-party component. Using an external component will save the cost of developing it, but time must also be invested into finding, learning and integrating it into their project.

When using an external component, a decision on what form of reuse has to be made. The two forms are:

- Referenced – The external software is stand alone and will change over time, because the component is being updated as a result of use by other projects.
- Forked – The external source code is copied to a local or private location, making it “static”. Any updates or bug fixes that the original code receives are not updated.

Fork-reuse is often discouraged because it is a form of software duplication. However, the advantage of fork-reuse is that it is isolated. The project group is then able to make their own adaptations to the software. These changes are no longer restricted to the boundaries of the original component.

2.1.1 What makes software reusable?

There are multiple characteristics of software writing that make code reusable. Within software engineering, general practices have been formed to write reusable software. A list is given by Anguswamy et al. (2013):

Abstraction "Abstraction means concentrating on important essentials while temporarily ignoring the unimportant details."

Clarity and Understandability "The degree of clarity to which the module's purpose, capability, constraints, interfaces, and required resources are defined. The understandability is measured based on its self-descriptiveness: the criterion that measures how well a component explains its functions. It is provided by standard formats, prologue comments on each modules, etc."

Commonality and Variability Analysis "Classification, grouping of objects with behaviour (methods and operations) and characteristics (data) as a way of achieving commonality and variabilities."

Composition "How to connect different software components. Some guidelines include: identify and minimise import requirements (for helpers), identify and minimise interference among helpers, use layering to define complex components using simple ones, implement policy on top of mechanism."

Documentation "Documentation for software is essential for any future use or modification and critical for maintainability. Programmers are unlikely to reuse software that is not well documented or commented since it makes it harder to understand and maintain. Documentation should be self-contained, adaptable and extensible."

Encapsulation and Information Hiding "Encapsulation is a technique for minimizing interdependencies among separately written modules by defining strict external interfaces. The external interface of a module serves as a contract between the module and its clients, and thus between the designer of the module and other designers. A module is encapsulated if clients are restricted by the definition of the programming language to access the module only via its defined external interface."

Generality "The process of abstracting the commonalities and stripping away the differences (i.e. ignoring the details of how, when, where, and the constraints)."

Genericity "Genericity is the capacity for creating a package or an object class whose types are not completely defined. They maybe static (if defined before compile and run time) or dynamic (if defined during compile or run time)."

Linking of Tests to Code "Code may also be written to implement the test cases for the component part. Programmers generally would like to test code before reusing and such a design of linking test to code may encourage reuse."

Modularisation "A component should be logically partitioned into subcomponents that perform specific functions"

One Component Uses Many Helper Components "A component created for reuse may be built using many reusable components say from a library. When a component is built using other components, then the whole family of components should be considered as a single component. For example, if a component written in C uses component from a standard C library, then the written component combined with the library should be treated as one component"

Optimisation "In general, components built for reuse are usually slower than their equivalent reusable components. Organizations are more likely to use code that meets the quality standards of the organization. As a rule of thumb, if the reusable component is slower by more than 25%, it will not be used. So, optimization techniques such as profiling using profilers (profilers are language dependent) would encourage reuse of the components."

Parametrisation "Parametrization provides a controlled way of customizing a generalised component when it is reused by substituting in an allowed range of values for the parameters which are embedded "place holders" for the differences in the component."

Restrictiveness "State everything about the behaviour that is expected of a correct implementation – and nothing more ("restrictiveness"). For example, consider a component that has a functionality of performing certain operations on only the string data type. The component could be restricted to accept only the string data type, not other types such as integers or floating point numbers."

Self-documenting Code "… internal program documentation in two forms: self documenting code and program comments. They argued that self-documenting code is better than code that relies on program comments. This is because self-documenting code requires less reading. Also, the comments may not be updated when the code is updated, but this cannot occur with self-documenting code"

Separation of Concepts from Content "One of the original motivations of the object-oriented approach is to promote reuse by separating the interface of an object from its implementation. This can be achieved, for example in C++, by using abstract classes to provide the interface and subclasses of the abstract class to provide the implementation. An abstract class is a reusable object-oriented design for a component. It specifies the interface of a class and the tree of subclasses that can be derived from it. Abstract classes fully specify behaviour, not implementation. They cannot be instantiated, only subclassed from."

Variability Mechanisms "A variability mechanism is a technique by which an existing content in a component can be customised or modified to be reused. Such mechanisms and techniques are popular in product line and domain engineering where variation points (points are identified in a product line where variable implementations are possible) and variants (the variable implementations) are identified to implement the variability mechanism."

Well-defined Interface "The interface describes the boundary of the component i.e. what operations it offers, what parameters it takes, and what it demands from the environment. The distinction between the interface (the specification) and the body (the implementation) of a component plays an important part in the modularization of software, not only in object-oriented development, but also in more traditional paradigms."

At the moment, there is no general consensus on what the best characteristics for reusable software are. It is often a trade-off between writing specific and generic software. Specific software is useful when optimising for code execution time, for example with realtime systems. The drawback is that the software cannot be easily reused. If the software is written in a more general way, there is often more overhead code. This results in a longer execution time. It also requires more abstraction from the programmer to implement the generic software. The advantage is that more generic software can be easier reused.

2.1.2 Reuse Readiness Levels

Besides software being reusable, it needs to be actually used by other users in different projects. Software that is not well documented, badly structured, too complex, or that cannot be com-

Table 2.1: NASA's Reuse Readiness Levels topic areas. (Marshall et al., 2010)

Documentation	Information that describes the software asset and how to use it.
Extensibility	The ability of the asset to be grown beyond its current context.
Intellectual property issues	The legal rights for obtaining, using, modifying and distributing the asset.
Modularity	The degree of segregation and containment of an asset or components of an asset.
Packaging	The methodology and technology for assembling and encapsulating the components of a software asset.
Portability	The independence of an asset from platform-specific technologies.
Standards Compliance	The adherence of an asset to accepted technology definitions.
Support	The amount and type of assistance available to users of the asset.
Verification and testing	The degree to which the functionality and applicability of the asset has been demonstrated.

piled independently will not be reused by others (Anguswamy et al., 2013). Besides the source code, other aspects also play a role in how reusable the generic code is.

Nine topics of reusability have been identified by Marshall et al. (2010), given in table 2.1. Each topic is subdivided into nine Reuse Readiness Levels (RRLs) as shown in table 2.2 to give a quantitative value to each topic. In appendix A a table summary is given combining the nine RRLs and topics.

Looking at the levels of reusability, the goal is to create a workflow such that created software is able to reach RRL 9 in all nine topics. However, this requires reuse of the software by multiple users in multiple systems. Due to time constraints of this thesis, this is not possible. Another restriction due to time is the effort put into making software reusable for different Operating Systems (OSs). As a result of these restrictions, RRLs 7 and higher are not achievable for all topics. Within this thesis, the goal is to reach RRL 6 for all topics.

2.1.3 Combining reusable characteristics and the Reuse Readiness Levels

To get an overview on how the characteristics help create reusable software, they have been mapped to the RRLs. One characteristic can influence multiple topics and can be mentioned more than once. In table 2.3 the mapping is given.

The characteristics of the software are facilitating required aspects of code writing, needed to create the functionality defined by the RRLs. The level written in the RRLs column states that a characteristic facilitates the functions for this and all underlying RRLs. Each characteristic is given an RRL individually, but that does not state whether two characteristics are mutually exclusive or depend on each other.

2.1.4 Requirements

In table 2.3, it can be seen that the list of Anguswamy et al. (2013) covers modularity and extensibility up to RRL 6. Being two key points for the creation of small reusable parts, this was expected.

Table 2.2: Reuse Readiness Levels of reuse as given by NASA (Marshall et al., 2010)

RRL 1	Limited reusability; the software is not recommended for reuse.
RRL 2	Initial reusability; software reuse is not practical.
RRL 3	Basic reusability; the software might be reusable by skilled users at substantial effort, cost, and risk.
RRL 4	Reuse is possible; the software might be reused by most users with some effort, cost, and risk.
RRL 5	Reuse is practical; the software could be reused by most users with reasonable cost and risk.
RRL 6	Software is reusable; the software can be reused by most users although there may be some cost and risk.
RRL 7	Software is highly reusable; the software can be reused by most users with minimum cost and risk.
RRL 8	Demonstrated local reusability; the software has been reused by multiple users.
RRL 9	Demonstrated extensive reusability; the software is being reused by many classes of users over a wide range of systems.

Documentation is also addressed and can reach RRL 5 if a manual is written. On the lower scale are intellectual property issues, packaging, portability, standards compliances, support and verification and testing. For users these topics are just as important as the top three topics. The only difference is that these topics do not always directly influence the source code. However, these topics do make software more accessible for users that have not developed the code. When software is well documented, usable on different platforms, has a lot of support, and is able to be tested stand alone, it is more likely to be reused.

The list in table 2.3 shows possible characteristics to design a reusable component. However, to achieve RRL 6, using the table given in appendix A, the following points need to be addressed:

Documentation For documentation, the source code must contain comments which explain what each piece of code does (RRL 2). A basic readme file explains how the software is designed (RRL 3). Which functions are available for an external application are documented in an Application programming interface (API) document (RRL 4). Additionally a manual must be written explaining how to use the software (RRL 5). A tutorial is added explaining the user how to implement the software in an example (RRL 6).

Extensibility With parametrisation RRL 4 is reached, by being able to set configurations. All lower levels are defined as not being easily extensible. For RRL 5, the extensibility approach must be well defined and documented. RRL 6 can be reached by the separation of concerns creating multiple points that are extensible.

Intellectual property issues The developers and intellectual property rights statements must be listed in the source code, the documentation is visible during execution (RRL 5). In all lower RRLs no consensus has been made on the intellectual property rights statements. For RRL 6 a recommended citation must be stated.

Modularity The generic and specific functionality need to be clearly separated. For RRL 6, a clear separation of specific and reusable software is required. This is done by encapsulating the software and restricting it to do only a specific task. With well defined interfaces, the data and functions are properly accessible from the outside with general interfaces, while the internal source code can be implemented in several different ways. Lower RRLs are defined as source code that cannot be separated into individual generic functions.

Table 2.3: RRLs of code characteristics

Topic	Characteristic	RRL
Documentation	Clarity and Understandability	3
	Documentation (manual)	5
	Self documenting code	2
Extensibility	Composition	4
	Commonality and Variability Analysis	4
	Parametrisation	4
	Separation of concepts from content	6
Intellectual property issues		1
Modularity	Commonality and Variability Analysis	5
	Encapsulation and Information Hiding	7
	Generality	5
	Genericity	5
	Modularisation	3
	One Component Uses Many Helper Components	3
	Restrictiveness	7
	Separation of concepts from content	7
	Variability mechanisms	7
	Well defined interface	9
Packaging	One Component Use Many Helper Components	3
	Variability mechanisms	3
Portability	Abstraction	3
Standards Compliance	Encapsulation and Information Hiding	4
	Generality	3
	Optimisation	3
	Parametrisation	4
	Restrictiveness	4
Support	Linking of Test to code	2
	Self documenting code	2
Verification and testing	Linking of Test to code	2

Packaging The software should be able to auto build (RRL 3). Locations of resources are configurable and all configurable information is centralised (RRL 5). An OS detection system is available and the component can auto build for supported platforms (RRL 7).

Portability The documentation explains steps required to adapt when porting. Porting software to a different OS should be easily done.

With abstraction, the computation can be generalised and not be dependent on the environments the software is working within. The more generalised and abstract the software, the easier it is to port to different environments.

To achieve RRL 6, the software should be ported to most major systems. The documentation should explain any modifications required to change based on the OS. All lower RRLs are defined as the software being more difficult to port, based on dependencies.

Standards Compliance software development follows best practices (RRL 2). Standards are defined to be followed but not verified (RRL 4). When following standards, but verification is incomplete, RRL 5 is achieved. If following and verification of standards is done, RRL 6 is achieved.

Support The developers should be reachable and be willing to give support (RRL 3). Updates and patches are given at intermittent intervals, to fix bugs and update functionality(RRL 4). A central source of information (website) contains useful resources and answers to FAQ (RRL 6).

Verification and testing Unit testing has been implemented (RRL 2). Testing has been done for known and unexpected input(RRL 3). Error condition have been tested and requirements have been finalised. A simple simulation are able to run(RRL 4). The software should be tested in a laboratory context(RRL 5). The software should be tested in a relevant context(RRL 6).

A summary of the requirements to create a reusable piece of software is given in table 2.4.

2.2 Paradigms

In the previous section an overview was presented of characteristics of reusable software. Unfortunately it is impossible to incorporate all of them at the same time, because they may be contradicting. Based on experience and current practices, characteristics are put together to create a coding paradigm. Current practices in the Robotics and Mechatronics (RaM) group define three major paradigms in use, being Object-oriented programming (OOP), Component-Based Software Development (CBSD) and the Separation of 5 concernss (5Cs)s. These will now be discussed in more detail.

2.2.1 Object-oriented programming

Object-oriented programming (OOP) is a programming paradigm based on the concept of “objects”. These objects may contain data, in the form of fields, often known as attributes; and source code, in the form of procedures, often known as methods. The idea behind objects is that the only way to change internal data is by use of its procedures, so external code cannot directly manipulate data. There are different OOP languages, however the majority of them are class-based. This means that each object is created as a class.

The main focus points of OOP are:

Encapsulation Encapsulation is used to make an object independent of other parts of the software. This hides the internal implementation and only shows the defined interfaces to the rest of the code.

Table 2.4: Deliverable requirements based on RRLs.

1. Documentation
   - 1.1. Code commenting
   - 1.2. Reference manual
   - 1.3. User manual
   - 1.4. Tutorial
2. Extensibility
   - 2.1. Program with extensibility in mind
   - 2.2. Documentation on extensibility and future plans
3. Intellectual property issues
   - 3.1. Recommended citation
   - 3.2. Intellectual property rights statement, ownership and/or copyright is written in source code and documentation, and is visualised during execution.
4. Modularity
   - 4.1. Clear separation of specific and reusable components
   - 4.2. Organisation of all components into libraries or service registries
5. Packaging
   - 5.1. Auto build function
   - 5.2. OS detection
   - 5.3. Centralised reconfigurable
6. Portability
   - 6.1. Be able to port code to major OSs without modification
   - 6.2. Documentation on modifications required when porting to non-major OS
7. Standards Compliance
   - 7.1. Comply with specific and proprietary standards (to be defined by the user)
   - 7.2. Verify compliance through testing.
8. Support
   - 8.1. Give centralised support via a website
   - 8.2. Give updates/patches at regular intervals
9. Verification and testing
   - 9.1. Test individual component inputs and outputs
   - 9.2. 'White box' testing
   - 9.3. Test through simulation
   - 9.4. Test within a fully integrated environment

2

Inheritance Inheritance is another way of building an object by using other existing objects. In section 2.1.1 this characteristic is mentioned as “One Component Uses Many Helper Components”.

Polymorphism Polymorphism is when there are multiple implementations of the software. Dependent on the 'higher' object calling is, a different behaviour can be expected. This is a form of “Parametrisation”.

2.2.2 Component-Based Software Development

A successor to OOP is Component-Based Software Development (CBSD). Here the software is broken down into logical components. However, instead of combining these at source code level, each component is built into a piece of binary software that is able to interact with other components.

A comparison is made between OOP and CBSD in figure 2.1. On the left is a simplified model of OOP, where at the bottom the software is constructed by combining objects to create bigger and more specific objects. Eventually it is compiled and can be executed. On the right is the CBSD model, where individual components are compiled into binary form and have predifined interfaces with other executables. However, an overhead system, called middleware, is required to “glue” the components together and facilitate the data transfer between components.

The main focus points of CBSD's characteristics are:

- Reusable – Components are usually designed to be reused in different situations and different applications. However, some components may be designed for a specific task.
- Replaceable – Components may be freely substituted with other similar components.
- Not context specific – Components are designed to operate in different environments and contexts.
- Extensible – A component can be extended from existing components to provide new behaviour.
- Encapsulated – A component depicts the interfaces, which allows the caller to use its functionality, and does not expose details of the internal processes or any internal variables or state.
- Independent – Components are designed to have minimal dependencies on other components.

In the current development of robotics, a robot is designed as modular as possible. For example, for a robot to pick up an object, it needs an arm and a tool. These two parts are designed and controlled independently. On a higher level, these to parts are controlled by one controller. Above this controller there is another controller that may control the base where the arm and tool are mounted on. Each module is designed to do a specific task. The individual modules are able to interact with each other by means of an overhead framework called a middleware. Such a framework is called a Component-Based Software Framework (CBSF).

2.2.3 5Cs

Another paradigm towards reusable software is the Separation of 5 concerns (5C) defined by Bruyninckx et al. (2013). A separation of concerns isolates certain aspects needed within a component. It is then possible to change one of these concerns within a module without having to adapt other concerns.

In figure 2.2 the decomposition of a module is given as explained by Bruyninckx et al. (2013). A summary of the concerns is:

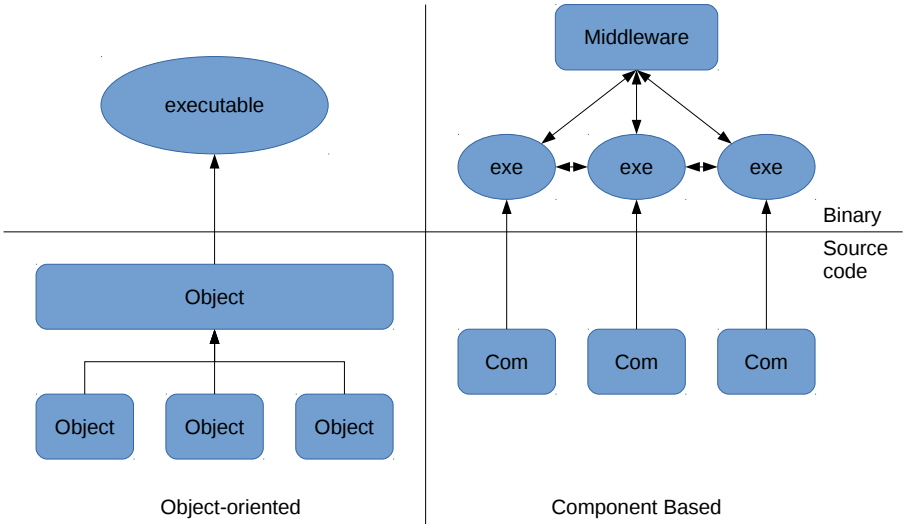


Figure 2.1: A comparison between Object-oriented programming and Component-Based Software Development

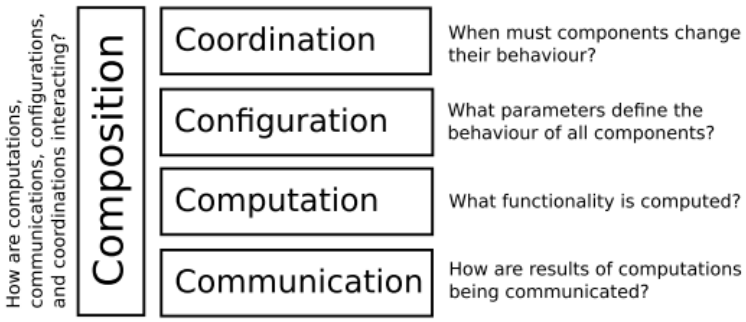


Figure 2.2: The 5 concerns as described in the BRICS overview Bruyninckx et al. (2013)

Computation: This is the core of the component. Here the source code is written for a specific function.

Communication: This is the input and output for the data for the component.

Coordination: This controls how the component works within the whole system, for example in which state it should be.

Configuration: This allows users to give values to parameters of different functions within the component.

Composition: While the four 'C's' given above will help separate concerns, the way they are designed will be a trade-off between specific and general software.

Keeping the 5Cs in mind while writing software can help distinguish the function/goal of the code within the component. Once the function is clear, it must be written in such a way that the writer tries to minimise the amount it overlaps with the other concerns. If this can be achieved, one of the first four C's can be changed without the need to modify any of the others.

A CBSF implements the communication and coordination aspects as discussed by the 5Cs, while facilitating a means to also send and change the configuration of a component. This means the composition is fixed by the CBSF, leaving the user free to implement the computation of the component.

The drawback is that often the component is written in such a way that it is highly dependent on how the CBSF handles the individual decoupled parts of the software. This results in the software being reusable within the same CBSF, but not outside of it.

2.3 Paradigm models

From the design paradigms, a model is created that envelopes the "ideal" component model. Using MDD, the goal is to express the behaviour in a model that is built from smaller reusable models. A high level system is created from smaller building blocks.

There are many models and concepts designed on the explained paradigms. More detail will be given to two models used extensively in robotic applications within the RaM group. These two models are the BRICS Component Model (BCM) and the Generic Architecture Component (GAC).

2.3.1 BRICS Component Model

The BCM has been designed in the Best Practice in Robotics (BRICS) project (Bruyninckx et al., 2013) and is a combination between the CBSD and 5Cs paradigms. The BCM itself is a design paradigm, in that it introduces a methodology. It is based upon the meta-modelling paradigm design by the Object Management Group (OMG) (Object Management Group, 1999). In figure 2.3 the structure of the BCM is given. It is divided into four layers:

- M3 –The highest level of abstraction, containing all constraints or restriction that a model needs to follow, without defining any domain.
- M2 –The meta-models. An abstract model is given in the form of a Component-Port-Connector (CPC). Here the model is divided into logical components that are able to connect with each other through ports. The CPC can be specialised to a specific software framework. Examples are Robotic Operating System (ROS) and Open Robot Control Software (OROCOS).
- M1 –The specific models. The big difference between M1 compared to the higher levels, is that the user will develop the implementation in this environment.
- M0 –The physical source code. For the implementation, binary code is generated that will run on the system.

The idea is that an abstract model is built at the M3 level without taking into account any platform or language dependencies. From this model, more platform specific models can be defined in M2, followed by the language dependent model in M1. Ideally this process from abstraction to practical software should be automated as much as possible. BRICS has started to create this automated toolchain with de creation of BRICS Integrated Development Environment (BRIDE). BRIDE is able to generate a CPC model for ROS and OROCOS from M3 to M1. The limitation is that at M1, only the "shell" is created, wherein the user then needs to implement the computation. With a "shell", generated source code is meant that connects at level M2 of figure 2.3. These shells do not hold any software or source code containing their specific application behaviour.

Following the 5Cs, BRICS has been able to separate concerns in the following way: Composition and Configuration is defined by the user's CPC model. Communication and Coordination is handled by the specific middleware and Computation is left for the user to fill in.

2.3.2 Generic Architecture Component

In previous work done by Bezemer (2013) a GAC has been designed. The goal is to provide a template for designing component implementation. After reviewing existing CBSFs, Bezemer

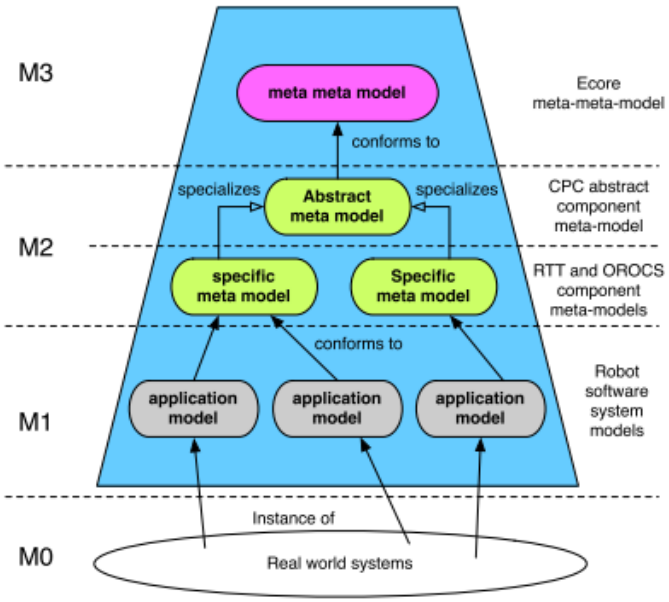


Figure 2.3: Four layers of the BCM as described by Bruyninckx et al. (2013)

(2013) concluded that both the component models and the execution framework are lacking. The new proposed GAC template is given in figure 2.4.

The main focus of GAC is on embedded systems and therefore has been kept simple and lightweight. A GAC component is split into three execution layers:

- Run once layer –Run once during initialisation.
- Synchronous layer –Periodic execution layer.
- Asynchronous layer –Executed when an event is received and needs processing.

The 5Cs is a major design point of the GAC added with a safety layer. Communication and Coordination are not directly visible in figure 2.4. However, the addition of ports allow for communication between different GACs. Indirectly, the composition aspect is implemented by an architectural network of GACs. The safety layer allows monitoring incoming and outgoing signals. Should an error occur, an error event can be sent out through the Coordination block. It also allows the receiving of a error event that has been signalled externally so that appropriate action can be taken.

The GAC model has been implemented into a Communicating Sequential Processes (CSP) execution framework called LUNA Universal Networking Architecture (LUNA). LUNA has been designed to interact with the Twente Embedded Real-time Robotic Application (TERRA) tool. This is a graphical editor that allows a MDD toolchain.

2.4 Designing using a paradigm

In the previous section design paradigms to write reusable software have been discussed. To help work within a chosen paradigm, a toolchain can be constructed that goes from an abstract model to the specific modular software. Using MDD, software is designed as an abstract model that is converted into source code, as seen in figure 2.3. However, the generation of the source code for the computational part of a component is not possible. Therefore, the goal is to create computational software components that are designed to be inserted into the generated software. These computational software components are designed from the bottom-up to be reusable.

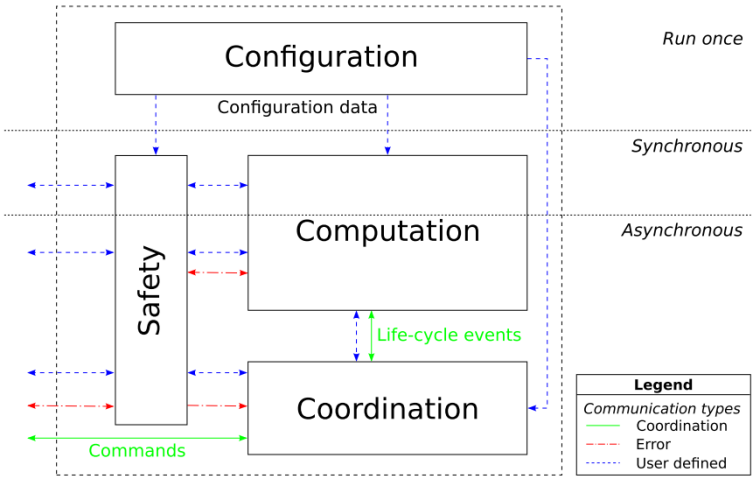


Figure 2.4: Design overview of the Generic Architecture Component (Bezemer, 2013)

2.4.1 Top-Down overview

The best practice for designing a robotic set-up would be by programming it in a platform and language independent way. A possible way of writing platform and language independent is by using models. Universal Modelling Language (UML) models designed by OMG (2015) are often used to show the software structure. This software structure can be multi level, such as the high level abstraction to the low level specific implementation. Ideal systems that have been design in UML models should be able to generate the underlying models and eventually the lowest layer of source code. These steps from high level design to more specific models down to the actual source code generation is called a toolchain.

Currently, a toolchain that converts abstract models to specific software does not exist. There are some partial toolchains that do certain steps of the chain, but not the entirety, the behaviour need to be implemented by the user.

For ROS, the BRIDE tool has been developed within the BRICS project (BRICS, 2010). This is an Integrated Development Environment (IDE) that allows the user to create a ROS generated system from UML models. This ROS generated software is only a “shell” containing the high level interaction between ROS “nodes”. In the UML models the nodes with their interfaces, variables, data types and Finite State Machine (FSM)s are set, which are generated in the shell. This means that the communication, configuration, composition and coordination is already implemented, leaving the user free to focus on writing the computational software.

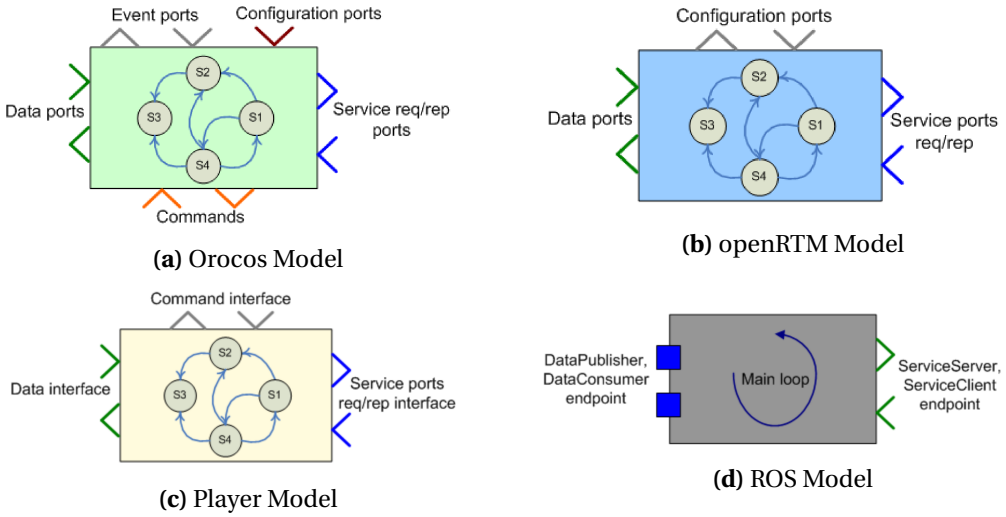
A different example of an existing toolchain is TERRA. This IDE has been developed together with LUNA to allow the user to design a model using simple blocks. Similar to BRIDE, TERRA only generates a “shell” topology of LUNA components. The computational software of each component still has to be written by the user.

2.5 Middleware

To use a CBSD, an additional piece of software is needed to “glue” the components together. This “glue” is often called “middleware”.

The role of middleware is not clearly defined and is different for each implementation. For this thesis a middleware is defined that uses a CBSF paradigm to connect individual components. In this section the middleware is reviewed using a top-down approach.

The BRICS (2010) project looked into different middleware and tried to compare them. In their deliverable by Shakhimardanov et al. (2010), a comparison is made between four component



(a) Orocos Model

(b) openRTM Model

(c) Player Model

(d) ROS Model

Figure 2.5: Component models used in robotics software as found by Shakhimardanov et al. (2010)

based middleware and how they work. In figure 2.5 a generalised model is given for the four CBSD middleware. In figure 2.6 the FSM is given for the LUNA framework. LUNA is interesting due to the fact that it is a realtime middleware, while the others are not.

In figure 2.5 it is visible that a component can have several ports for data, services, events and configuration, while the user software is encapsulated internally within the FSM. The statemachine controls the life cycle management, for example the creation, running and termination of the node.

The general FSM structure for ROS, OROCOS, Player, openRTM and LUNA is the same. A general structure for software writing would make the code usable in all five models.

A major difference between the middleware is the way data is stored and sent. The structure of the data is middleware dependent, such that data cannot be sent from one middleware to another.

An addition to the discussed middleware of figure 2.5 is LUNA (Bezemer, 2013). "The LUNA framework is a hard real-time, multi-threaded, multi-platform, CSP capable and a component based framework" (Wilterdink, 2011). This realtime software has been developed at the University of Twente. It has the ability to interact with a ROS network, so that certain parameters and values can be read, set and written in a non-realtime way, while locally the control software is running realtime loops. The design of the internal, embedded source code can be done externally and is seen by ROS as another (black box) node. On the other hand, LUNA is also a CBSF, so any generalised software could also be added and run as a realtime node.

Of the five mentioned middleware, within RaM the most well known are ROS and LUNA. These two middleware are evaluated using the RRLs in table 2.5, to see which topics have achieved the desired level.

Reviewing table 2.5,it can be seen that both ROS and LUNA focus on modularity and packaging of the components. Both are lacking Standards Compliance. The difficulty here is defining to which standards they must comply. If no standard is clear, it is impossible to test for it. Support for LUNA is lower than ROS due to the fact that ROS is a widely used open-source platform with far more users than LUNA. Furthermore the RRL rating of 1 for LUNA on Intellectual property issues is because no license of use was found.

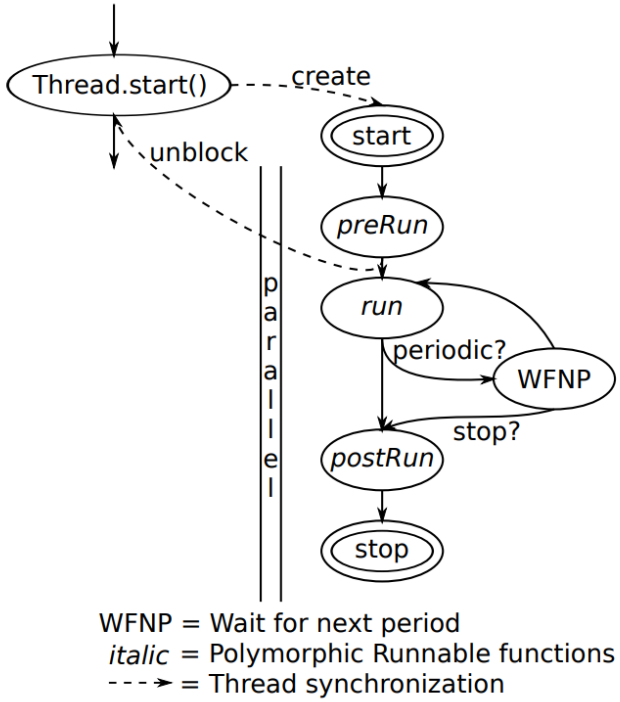


Figure 2.6: LUNA FSM Model (Wilterdink, 2011)

From table 2.5 it is visible that ROS and LUNA have high levels of reusability. However, this does not state anything about the software that is encapsulated within the components that the middleware is running.

2.5.1 Bottom-Up Approach

As discussed in the previous section, both BRIDE and TERRA generate a high-level topology of component "shells" that need to be filled. Instead of having to redesign the behaviour of each component, it would be more convenient if there are existing implementations of these computational components that can be integrated.

In figure 2.7 a generic toolchain is visualised. At the top a user can express the application in an abstract model. The abstract model is then structured in such a way that it is applicable for

Table 2.5: Using the RRLs as described in appendix A, current middleware are compared. The focus is on what functionality the middleware adds. The level of reusability of an individual component is discussed in section 2.5.1

Topics	ROS	LUNA
Documentation	6	6
Extensibility	7	5
Intellectual property issues	7	1
Modularity	9	9
Packaging	7	7
Portability	5	7
Standards Compliance	3	3
Support	5	2
Verification and testing	7	6

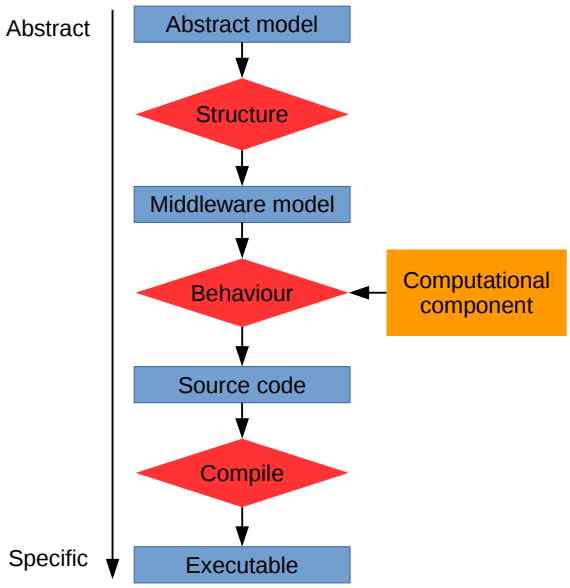


Figure 2.7: A generic toolchain from abstract model to compiled executable.

a middleware. The middleware models are then generated and the user needs to implement the behaviour of each component. Instead of writing the behaviour, an existing computational component is integrated. Once the source code has been written, it is compiled and can be executed.

The goal is to create a general template for the behaviour software within a component, in such a way that it can be used in different middleware.

2.6 Approach

In section 2.1, it has become clear that to design a reusable component, more than just the code needs to be taken into consideration. Using the RRLs in section 2.1.2, a set of requirements for a reusable component is defined in section 2.1.4. The following approach to design reusable software is suggested, based on the knowledge gained in the previous sections.

To standardise the way of implementing software components, OOP and CBSD paradigms should be used as discussed in section 2.2. Software for a robot is designed as an abstract model. Based on existing CBSD toolchains, a middleware structure is generated from the model containing component shells, as discussed in section 2.4. By using middleware, the coordination, communications, composition and configuration are standardised, leaving computation to be done by the user. This approach follows the 5Cs within the middleware. The middleware that are used during implementation are ROS and LUNA, because of the knowledge about these middleware within the RaM group. The computation part of a middleware shell is filled in by integrating computational software components. A wrapper is used as an adapter to decouple specific middleware data types form the computational software component data types.

The computational software components should meet the requirements set in section 2.1.4. This will partially be achieved by following the OOP paradigm. The use of additional tools is suggested to automate steps during design of the computational software component and thus to facilitate achieving the full list of requirements. Following these ideas, the proposed methodology is described in chapter 3.

3 Design and Implementation

In this chapter the design of the methodology to create a computational software component is discussed.

Starting from the compiled component as a library, its structure and layout are explained, followed by more detail on how the source code is written. Emphasis is given on documentation. In section 3.3, the steps and tools required to compile and test the component are explained. Sections 3.4 and 3.5 discuss more topics that are not directly linked to the source code itself, namely intellectual property and support.

A demo component is created to show that the discussed toolchain is viable. This demo component is a general SISO PID controller, which are often used in robotics. It receives a setpoint and a feedback value, to calculated the next steervalue.

3.1 Design of a computational software component

For software design of the computational code, a structured way of working is desired. The software must be extensible without breaking existing uses. To achieve this an object-oriented approach has been chosen. This means that the computation is written as a standalone, encapsulated piece of software with a predefined interface for external code. To achieve this, the software is written using a “universal” language that is used in all five described middleware in section 2.5. The language is C++. This language was chosen since it is used by almost all OSs and is seen as a “mature” language. Furthermore, it is still being supported and updated. However, the way that data is transported within the middleware is dependent on their local implementation. For the computational software to receive data, the data needs to be converted from middleware specific to generic. Computations can then be done by the generalised software. After the computation is done, the data is converted back to a middleware specific data type. These conversions also need to be done when setting parameters during initialisation, configuration and destruction of the node.

In C++, functions and/or classes can be created externally and included into a different file. Due to the fact that the computation is already defined in an external class, it can be included into the shell code. A class is a piece of encapsulated software that contains its own data variables and functions. Through a fixed interface, external code can call certain functions within a class. The class can be created by external software, in this case the shell.

Functions can be called at the moment that the shell FSM is in a particulate state, calling the external functions as the shell needs them, see figure 3.1. For example, the looping computation state of the node is then able to call the external class’s functions to do the computations.

When the compiler then creates the final executable software, it will add the ROS shell, the wrapper and the computational class together as one executable node.

3.2 Writing source code for a computational software component

The general structure of writing source code for a software component is discussed.

To help the programmer design a component, tools can be used to automate certain aspects. The tools mentioned here are suitable for C++, but tools with an equivalent functionality exist for different programming languages. The main IDE used in this thesis is Visual Studio Code (Microsoft, 2016). This IDE was chosen for its easily integratable tools and integrated terminal. Other IDEs are also applicable, however integration of external tools cannot be assured. All tools can be run as an external command in a terminal and are cross platform.

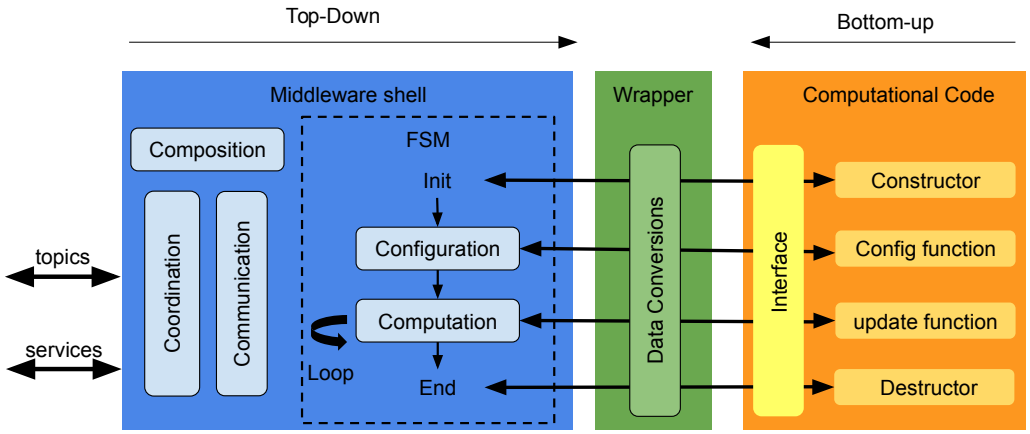


Figure 3.1: A possible approach towards generalised code, using a generated ROS shell that uses function calls from generalised code. The data is transferred through a wrapper, to convert ROS specific data types to general data types.

3.2.1 Structure of a component directory

A standardised layout helps future users to find specific files, since their location is the same for all components. This layout also standardises the paths written in the compiler configuration. In figure 3.2, an overview is given of how to structure a component directory.

The main two subdirectories are the *src* (source) and *include* folders. These hold the source and header files of the component. This follows common C++ practices. As a result, all the header files are collected, which are required when exporting a compiled library.

Under the *doc* folder, a doxyfile is stored. This contains the configuration to generate formatted documentation in its subdirectories. More information is given in section 3.2.5.

All test related files are collected in the *test* folder. In figure 3.2 two testsuites are defined. This can be expanded into as many testsuites the user wants. More information on testing is given in section 3.3.2.

The last listed folder is the *build* folder. Here the compiler is able to put any created and compiled files. Furthermore, the file *CMakelist.txt* is given in every level. This file is a part of the compiler. More information about compiling a component is given in section 3.3.1.

Another advantage of using a standard structure, is that it is easy to use a version control system. By using subdirectories, source code is isolated from compiled or generated code. Folders containing generated or compiled content are in general not stored in a repository.

3.2.2 Compiled libraries

The compiled component needs to be available to be used in other applications in the form of a library. This library can then be included into different applications and its functions can be called.

A well known example in C++ is the include of *stdio* to use the *cin* and *cout* functions in software, and the same approach can be used to include a compiled library into a wrapper that has been generated by middleware.

By using this approach to separate middleware shells from the reusable computational part of components, modularity requirement 4.1 from table 2.4 has been achieved.

```
/Component
├── /build
├── /doc
│   ├── /documentation
│   │   ├── /latex
│   │   └── /html
│   └── doxyfile
├── /include
│   ├── interface.h
│   ├── implementation.h
│   └── **.h
├── /src
│   ├── implementation.cpp
│   └── **.cpp
├── /test
│   ├── testsuite1
│   │   ├── implementationTest1.cpp
│   │   └── CMakelist.txt
│   ├── testsuite2
│   │   ├── implementationTest2.cpp
│   │   └── CMakelist.txt
│   └── CMakelist.txt
└── CMakelist.txt
```

Figure 3.2: Directory of a component

3.2.3 Interfaces

An interface needs to be defined to standardise the way external software can use a library. Within the library an array of implementations can be available. However, any function or information that needs to be exchanged with an external application has to go through the interface. An example of an interface with several implementations is given in figure 3.3.

By keeping the interface the same, a change in implementation can be done without breaking any external software that depends on the component. A new interface can be added, if the current interface is not adequate. This assures that existing applications that use the original interface do not break.

Within C++ an interface object is not defined. To implement an interface, a virtual abstract class is used. In figure 3.4 the interface is given that is used for the demo component.

An interface can be defined as follows:

- All functions should be virtual, because an interface cannot be initiated. Only the inheriting class that implements the functions can be created.
- Because the interface itself can never be initialised, no constructor should be defined. The appropriate constructor of the desired implementation must be called to initialise the object.
- The destructor should be defined to ensure that if the interface is deleted, all inheriting classes are properly destructed to prevent memory leak.
- A printConfig function should be defined in an interface. This function prints a string to the terminal, that states an implementation's current settings.
- A set Parameter function is defined to allow an external application to set internal parameters, without directly accessing the parameter. The parameter to change and the new

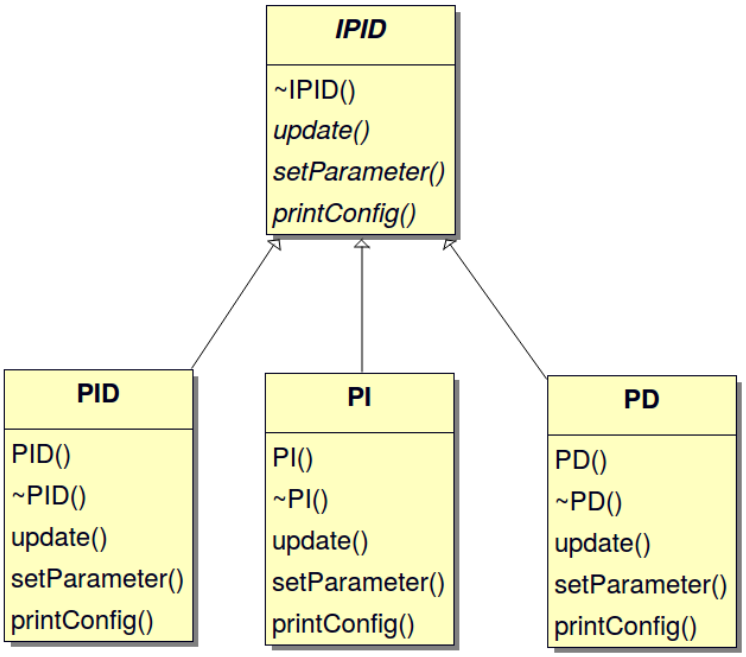


Figure 3.3: A UML diagram of an interface that is inherited by different classes

```
#include <string>

class IPID {
    public:
    virtual ~IPID(){};
    virtual void printConfig() = 0;
    virtual int update(double setpoint, double previousValue,
        double & steerValue) = 0;
    virtual int setParameter(std::string parameter,
        double value) = 0;
};
```

Figure 3.4: IPID interface header without documentation comments

value are given as arguments. The implementation of the software then uses these arguments to set the internal parameter. The functiuon should return a warning or error signal if setting a parameter has failed.

- All necessary input and output data that functions require will pass through its arguments.
- Any errors and/or warnings that have been detected in the implementation are signalled back through the return value of the function. A function returning a 0 value is successful. A positive number stands for a warning. A negative value is an error. A list of warning and error values must be documented in the API documentation.

By using the OOP approach of interfaces, it ensures that software is extensible in future use without breaking existing code. Extensibility requirement 2.2 of table 2.4 is then achieved.

Classes A class is an implementation of the interface it inherits, as shown in figure 3.5. The functions of the interface must be overridden, to make sure that the virtual function in the interface is replaced by the actual implementation within the class. A class may have private

```
1 class PID : public IPID {
2 public:
3     PID();   ///< Constructor
4     ~PID();  ///< Destructor
5     int update(double setpoint, double previousValue,
6     double & steerValue) override;
7     int setParameter(std::string parameter, double value) override;
8     void printConfig() override;
9
10 private:
11     double _dt;          ///< loop interval time
12     double _Kp;          ///< proportional gain
13     double _Ki;          ///< Integral gain
14     double _Kd;          ///< derivative gain
15     double _max;         ///< maximum value of manipulated variable
16     double _min;         ///< minimum value of manipulated variable
17     double _pre_error;   ///< Previous error value
18     double _integral;    ///< Integral value
19 };
```

Figure 3.5: PID implementation header file

functions that it uses internally, but any connection with external applications must go through the interface. This is done to comply with the OOP paradigm of encapsulation.

The PID class inherits the IPID class as seen in figure 3.5, line 1. All functions must be defined the same as in the interface, see figure 3.4, where the word *override* is appended to show that this function overrides the virtual function.

All internal parameters are defined in the header as private attributes. To change a parameter, the external application can call the setParameter function. In the implementation a string has been linked to a parameter, such that if the input string matches, the new value is set. Since the parameters for setParameter are “open", it is up to the writer of the implementation to define what parameters are able to be set by the external application. These parameters must be well documented in the API documentation.

By not defining the parameters in the interface, but allowing a variable string to select the parameter, this function is easily extensible. A warning must occur if a non-existing parameter has been selected. This extensible interface achieves requirement 2.1 from table 2.4.

3.2.4 Uniform source code

To make the source code clean and readable, a standard way of writing has to be defined. Google C++ Style Guide (Google, 2017) was chosen, since it has extensive documentation on its design choices and it is widely used.

To make sure source code follows this style, a linter tool is used. This tool will go over the source code and inform the writer when the source code does not follow the style. The writer can also choose to let the linter auto-format the source file when saving or closing the file. The tool used by the writer is Cpplinter. This integrates with VSCode and formats according to the Google C++ style.

By writing uniform source code and using a linter tool to follow the style, requirement 7.1 and 7.2 from table 2.4 on Standards Compliance have been achieved.

3.2.5 API Documentation

Now that writing the source code has been explained, the documentation for future users of the component are discussed.

To help generate API documentation, the *de-facto* standard tool *Doxygen* (van Heesch, 1997) is used. By properly commenting the source code, Doxygen can extract the information required to generate a website or latex document that holds the minimum required API information. An example of commented source code is given in appendix B.

As a bare minimum, all header files need to be fully documented. At the top of the file a description explaining the file's function, the authors name and email and the copyright markings should be described.

Every class and function needs the following commenting:

- A description containing a one line brief description, followed by a longer description. In the longer description, the general workings of the object must be explained and how to use the object.
- Any input or output parameters must be defined and also what their default values are.
- The return values of the class must be given and all possible error and warning codes must be listed.

At a minimum, this documentation should be done for functions and classes that are defined in the interface.

To generate the documentation from the source code, Doxygen reads a doxyfile, which is located in the *doc* folder. In the doxyfile settings can be adjusted to create the desired documentation output. The Doxygen tool is called in the *doc* folder. It finds the doxyfile automatically and starts generating the documentation for the software. The Doxygen tool is also linked to the compiler, such that when the source code is being compiled, it also calls Doxygen to generate the documentation. Because the documentation is isolated in the *doc* folder, cluttering of other folders is prevented.

At the moment of writing this thesis, the generated documentation is either a latex document or a HTML local page. In the future, the html documentation can automatically be uploaded to a central website to keep documentation up to date for other users.

With Doxygen the API Documentation is generated and documentation requirements 1.1 and 1.2 from table 2.4 are achieved.

3.2.6 Manuals and tutorials

To achieve requirements 1.3 and 1.4 for documentation, a manual and tutorial are still required.

This manual contains:

- How to pull source code from GIT
- A list of tools required to compile
- Locations to download and install these tools
- Commands needed to compile the source code
- Explanation of how the component is constructed and how to use it

The manual for the component is written as a concise readme file that is readable in GIT and is linked with the API Documentation.

A tutorial gives a user insight on how to use a component by explaining which steps are needed to use this component.

At the time of writing this report, no tutorial was written, due to time limitations.

For a tutorial, the following step by step explanations are required:

- How to include files / libraries
- How to use interfaces
- How to initiate implementations
- How to interact with functions

With the API Documentation and the concise manual, documentation requirements 1.1, 1.2 and 1.3 of table 2.4 have been achieved. Requirement 1.4 is achieved when a tutorial is written.

3.3 Compiling and testing software components

The documented source code that is written needs to be compiled before it can be run. After the source code has been compiled, it needs to tested. In the next section the compiler will be explained, followed by the testing tool.

3.3.1 CMake

To compile source code a compiler is required. There are many compilers that are specialised for a certain architecture. However, the goal is to write the software once for multiple platforms.

To do this CMake (Cedilnik et al., 2000) is used. It is a free and open-source tools designed to build, test and package software. It allows to make compiler independent configuration files, that then can be used by native compilers. It works by defining a configuration file called *CMakelist.txt*, as visible in the directory shown in figure 3.2. The top level CMakelist defines all necessary variables in the main project. Underlying Cmakelists can be added to the main project as sub projects. For example, separate CMake files can be used for the main project and testing suites.

Compiling with CMake consists of two steps. First, local build files are generated from the CMakelist files. Then the platform's local compilers are used to build the library. For a Unix operating system compiling C++ files, CMake will generate Make files. The Make files are used by GCC, the local C++ compiler, to create the final component.

The toplevel project builds a library from the source code.

For the library to be readable by a different OS, for example an ARM chip, the compiler flags of the main project need to be adjusted. All subdirectories use the same flags, requiring the user only to change the flags in one location.

Besides compiling the source code, CMake is also able to call external programs, such a Doxygen and Google Test. Google Test is described in section 3.3.2. By including these tools into the toolchain, the creation of a component is automated.

CMake automatically compiles and builds the final component into a library, so packaging requirement 5.1 and modularity requirement 4.2 from table 2.4 are achieved. With CMake the compiler configuration can easily be changed to build components for different hardware and OSs. This achieves portability requirement 6.1 and packaging requirement 5.3. Portability requirement 6.2 is achieved when the steps required to configure the compiler for different OSs are documented.

3.3.2 Unit Tests

After a component is built, it needs to be tested. The most basic test is to check how it responds to predefined inputs and comparing the output values to expected values. To simplify the process for the designer, it can be automated into the toolchain. Due to easy integration with CMake and familiarity by the writer, Google Test (Google, 2015) was chosen. Any other testing tools that automates unit testing is valid.

Google test is a C++ test framework. It allows the programmer to define input values and expected output values, but also tests if a wrong input is handled correctly. From the software the output is checked with the expected values. If they are not the same, there is a problem in the software and this will be signalled to the terminal.

The Google Test framework can help debug problems, however the user must write all the tests. Multiple tests can be defined which run in sequence.

Tests need to be written for a component. Examples of possible tests are: If a set function is given with an unknown parameter, will the function return the correct errorcode? Does the set function with a correct parameter actually set the new value? With known input and parameters, is the output of the user defined functions as expected? These tests can be written using Google Test to automate testing of the component.

Another advantage of Google Test is that it integrates with CMake. CMake can automatically download the newest version of Google Test and compile itself using the compiler flags defined in the main project. Once all components are built, the test will automatically run and return any test failure back to the user. With this information, the user can then start debugging any problems that have been found.

With Google Test framework, testing requirement 9.1 from table 2.4 is achieved.

3.3.3 Integration tests

To achieve the other testing requirements, additional tests need to be performed.

A higher-level test compared to unit testing is “Whitebox" testing. This is giving a known system a known input, so the output can be predicted and the system verified. A stand alone piece of C++ code was written to include the external PID component. The PID component received a static setpoint from an offset starting point. A known mass component was also created for the PID controller to interact with. The software runs 100 loops to simulate 100 time steps. The output was recorded, plotted and manually verified to be working.

The next step is to integrate the component into a simulated environment. Here, unexpected inputs can be generated to see how the component responds. Furthermore, timing tests can be addressed.

The final level of testing is to connect the component to the external system in a controlled environment. Here the interaction with other components can be observed. A PID coputational component was included into a ROS node, generated by BRIDE, and a LUNA node, generated by TERRA. In both middleware, the component is connected to a complete setup with hardware input and output and able to run. This shows tha the PID component is reusable in different middleware.

With these tests, requirements 9.2, 9.3 and 9.4 from table 2.4 of Verification and testing can be achieved.

3.4 Intellectual Property issues

The main issue for Intellectual property is the selection of the type of license. For open source projects there are three main types of licenses:

- MIT License
- Apache License 2.0
- GNU GPLv3

However, more licenses are available than stated above. To help choose see GitHub (2017).

When a license has been chosen, it must be visible at the following location:

- In the source code (every file)
- In the documentation
- On screen when the software is executed

Often at the bottom of a license there is a boilerplate or example of the copyright statement that needs to be visible.

If an article or academic paper is the base or result of the component, a recommended citation should be added. This can be stated in the readme file of the manual.

If all this has been done, requirements 3.1 and 3.2 from table 2.4 for Intellectual property issues have been achieved.

3.5 Support

Another topic that has not yet been addressed is support. Support mostly focusses on how users can get help and information about the component.

To achieve support requirement 8.1 and 8.2 from table 2.4, formal support should be available for users. However, this is outside the scope of this thesis.

To give systematic support, an issue tracker can be used. A user can mention a problem and/or bug. The writer or the user can fix the problem and the update can be merged into the current version of the software. However, only RRL 4 is achieved.

4 Evaluation

Based on the requirements set up in section 2.1.4, the design and implementation is evaluated. First the toolchain as a workflow is evaluated followed by the evaluation of the demos.

4.1 Review of the implementation

In table 4.1 the requirements are written down with the outcome given in the toolchain column. Here, a review is given of how the general implementation meets these requirements.

Documentation The documentation requirements, source code commenting and reference manual, are achieved by the use of Doxygen. Guidelines on how to write and where to place a manual and tutorial are also given.

Extensibility Programming with extensibility has been achieved by using interfaces. New interfaces can be created for new features, while retaining existing interfaces, to prevent breaking of external applications. Documentation on the extensibility and future plans are taken into account. The way of designing an interface is mentioned in the manual.

Intellectual property issues In the manual an example of a citation is stated, which completes this requirement. To display an ownership/copyright statement in the running implementation, source code and manual, a license was chosen. Choosing this license is up to the reader.

Modularity A clear separation of specific and reusable components has been achieved, by using an OOP approach to write the source code. A general interface is defined for use by other applications, while the implementation can be changed. Any change of the implementation will not break the external application, because all calls to the component go through an interface. This results in a reusable component. The requirement on organising all components into libraries has been achieved by using the directory structure and CMake to build individual components.

Packaging An auto build function is achieved by using CMake to compile and create a library of components. It is possible to automatically detect the operating system, however this has not been implemented. Another point is that a user may want to compile for a different operating system then is being used to build the component. This should be configurable in the main *CMakelists.txt*. In this file, all build configurations can be adjusted to the users wishes. This creates a central configuration point.

Portability The requirement of porting the component is achieved, by using a cross compiler that is able to compile for multiple operating systems. Should any adjustments need to be made in CMake, this can be written down in the manual.

Standards Compliance The designed components should comply to the guidelines explained in this thesis. While most requirements are followed, there are still requirements that need to be addressed. This is discussed in section 4.3. Furthermore, all source code must be written to follow the Google C++ Style Guide and all files structured in a directory as predefined in figure 3.2. To validate this, a linter program is used to format the source code. CMake will give an error on compile time if the directory structure is not followed.

Support None of the support requirements have been achieved. This is because support is given after a component has been designed and is in use. This is outside the scope of this thesis. Possible steps to set up this support have been mentioned in section 3.5.

Table 4.1: Requirements status overview. ✓ = achieved, o = partially achieved, x = not achieved

Toolchain	ROS	LUNA	Requirement
			Documentation
✓	✓	✓	1.1 Source code commenting
✓	✓	✓	1.2 Reference manual
o	✓	✓	1.3 User manual
x	x	x	1.4 Tutorial
			Extensibility
✓	✓	✓	2.1 Program with extendibility in mind.
o	o	o	2.2 Documentation on extendibility and future plans
			Intellectual property issues
✓	✓	✓	3.1 recommended citation
o	✓	✓	3.2 intellectual property rights statement, ownership and/or copyright is written in source code, documentation and visualised during run.
			Modularity
✓	✓	✓	4.1 Clear separation of specific and reusable components
✓	✓	✓	4.2 Organisation of all components into libraries or service registries.
			Packaging
✓	✓	x	5.1 Auto build function.
x	x	x	5.2 operating system detection.
✓	✓	✓	5.3 Centralised reconfigurable.
			Portability
✓	✓	✓	6.1 Be able to port software to major OSs without modification.
o	o	✓	6.2 Documentation on modification required when porting to non-major operating system.
			Standards Compliance
✓	✓	✓	7.1 Comply with specific and proprietary standards (to be defined by the user)
o	✓	✓	7.2 Verify compliance through testing.
			Support
x	x	x	8.1 Give centralised support via a website
x	x	x	8.2 Give updates/patches at regular intervals.
			Verification and testing
✓	✓	✓	9.1 Test individual component in and outputs.
o	✓	✓	9.2 'White box' testing
o	x	x	9.3 Test through simulation
x	✓	✓	9.4 Test within a full integrated environment.

Verification and testing A test environment has been added to the toolchain. With this testing tool the unit tests can be automated. However, this only tests the components in an isolated manner. To test a component in a higher application level, the user will have to do simulations, white box testing or a similar test. Finally the component needs to be tested in a fully integrated environment. These tests cannot be automated within this toolchain, since they are not part of the components' build system.

4.2 Review demos

As mentioned before in chapter 3, a demo implementation was created. The simple SISO PID controller was written once and compiled for two different middleware, these being ROS and LUNA. The models designed in BRIDE and TERRA and source code of the demo can be found in appendix C.

The component was written using the toolchain designed in chapter 3. Additionally, a simple manual was written and a copyright license was chosen. These were added in the designated locations of the component. Furthermore a simple 'whitebox' test was written to test if the software works properly over time. From the 'whitebox' test it was manually verified that the PID software computation works as expected.

The ROS demo was designed to take a joystick position input and move a drivable output to that position using the PID component. A wrapper node was written that included the PID component and was able to read the input setpoint and give it to the component. The component then calculated the desired steervalue which was sent to the output. This resulted in the hardware reacting as expected.

To test if the component is reusable, the same component as used in the ROS demo is also used in the LUNA demo. The general idea is the same. However, since ROS runs on Linux, no changes where made in the compile step. LUNA runs on a RaMStix (ARM embedded OS). This requires the cross compiler to be set up differently. A compiling script specific for the RaMstixs was available as a Make (not CMake) file. This was used to build the component of the ARM embedded OS. Normally, CMake generates a Make file to compile the component. Due to time constraints, the reverse engineering step of generating the Make file from a CMake file was not performed. As a result, the build steps required to make the component had to be done manually. The built component was included into the LUNA software that was generated by TERRA. The system was able to control the position of the hardware based on the input setpoint using the PID component.

Reviewing the requirements once again in table 4.1, the ROS and LUNA columns show if the requirements have been achieved. As these are specific implementations, documentation has been written to support the components. The only difference between the ROS and LUNA demos is that the LUNA compiler's Make file is hand written, whereas the Make file is generated by CMake from the toolchain. Because of this difference, the manual has a section on how to compile code for a LUNA environment to achieve requirement 6.2 of table 2.4.

4.3 Discussion

In this section, requirements that have not been achieved are discussed, their reason of failure and how they could be fixed. Furthermore, issues found during implementation that need to addressed are given.

The toolchain as a whole is able to aid the writer with the general design of the component, by using the OOP approach and adding additional tools that can be integrated into one build function. However, the most requirements that have not been achieved are requirements that cannot be fully automated, such as writing manuals, tutorials, testing at a higher level than unit tests and setting up a support structure for users.

For documentation, it is required that the writer steps away from the software and dedicates time to writing. If time is limited, one of the first things to be neglected is documentation, as is also seen in the demo implementation of this thesis. Due to time constraints, no tutorials were written for the demo PID component.

Additional work needs to be done on integrating tools better into the toolchain. Doxygen is able to generate the API documentation, but has to be run as an external tool. CMake is able to integrate Doxygen into its build function to automatically generate the API documentation. Another requirement that has to be reviewed is the operating system detection. This is a feature of CMake, but has not been used. The question arises if this is recommended. There are circumstances where a user would like to compile the component for a different operating system than the one used to design it. An example is a component designed and compiled on a Linux PC, but runs on an ARM embedded chip. An alternative solution for operating system detection is operating system selection by giving CMake a variable. The variable sets which operating system it should compile the component for.

Peer review is required to test reusability. The component has informally been peer reviewed by two people from the RaM group. However, two reviews are insufficient to determine if a component is interesting for reuse. Furthermore, the point of interest is not just the reusable component itself, but the entire workflow and toolchain. More quality control is needed to identify limitations of this approach with reusable components and their design paradigm.

The entire toolchain is designed around C++ code, so is it language independent? The concept of interfaces and objects are not limited to C++ and are found in other languages. All tools used during implementation are available for different languages. So if a different language is desired, the same concepts of component design hold, as do the requirements. Tools aiding the writer may need to be swapped, but the function of the tool is still the same.

5 Conclusions and recommendations

In robotics multiple types of hardware, software and middleware are used. To overcome coding issues for different types of robots, the main goal of this thesis is creating software in a modular and reusable way.

5.1 Conclusions

This will be answered by first reviewing the sub goals.

Find out how reusable software is currently used in robotics Within robotics, often software is reused by creating components that can be reused with a middleware. Examples of these middleware are ROS and LUNA.

Identify the main problem/difficulty of creating reusable software in robotics. When writing reusable software, the main focus is on the source code, however there are more aspects to take into account to have a good reusable component. The source code must be easily expendable and modular. Documentation must be available at different levels, from source code until full tutorials. The component should be easily used on different platforms and packaged in a standardized way, for example a library, support for bug handling and feature extensions, and include testing possibilities.

Identify what kind paradigms of reusable software exist There are many different paradigms on how to write reusable software. Within robotics, the main paradigm is the use of middleware. The two main discussed middleware are ROS and LUNA. ROS follows the paradigm of CBSD. LUNA follows the paradigm of GAC. Other middleware have be looked at and a similar component structure has been identified. They all have a FSM within each component.

Define requirements for reusable software. Using the RRLs, a set of requirement has been constructed. These requirements should help the writer write a highly reusable component. To aid the writer a set of tools are given to help build reusable components.

Demonstrate that reusable software can run on different platforms. Using the knowledge of how a component has to interface with a middleware, and what the requirements are for a highly reusable component. A PID component was implemented and integrated into both a ROS and LUNA environment.

The main goal for this thesis is:

Write reusable software for robotic applications that is language and platform independent.

By first defining what is necessary to design a highly reusable component, a set of requirements is given. After reviewing in what way software is already being reused, namely middleware, the new component can be added to the existing paradigm of reuse. The middleware have similar FSM structures within each of their nodes, that could interface with a reusable component. A well defined interface simplifies adding a reusable component to existing middleware. A single reusable component that is able to be include into both a ROS and LUNA middleware is created and usable.

5.2 Recommendations

In this thesis a way of working has been described on how to write reusable software. However, it also needs to be adopted and tested by other people in other applications. Over time more feedback will then be obtained. This can lead to updates in the way of working, resulting in a more robust work flow.

Reviewing the implementation of the toolchain, the following points need further investigation:

- Add operating system selection into CMake.
- Integrate doxygen into CMake.
- Configure CMake to generate Make files for LUNA.

From the evaluation of the methodology as a whole, future steps are required to verify the reusability. These steps are:

- Write a component in a different language and run it in multiple middlewares.
- Run the demo component in a third middleware.
- Define a standard structure for manuals and tutorials.
- The proposed methodology needs to be peer reviewed and tested by people with more knowledge on reusability.

The proposed methodology of designing computational software components is mainly based on an OOP paradigm. However, more methods used within OOP, could be added to the proposed toolchain. Examples are the use of "templates" to allow a function to accept different types of arguments, such as integers, floats, strings etc. Another is the use of "factory builds" to simplify the creation of computational objects with different initial parameters by using a 'factory' object. This helps creating computational software components that are more general.

A RRL Topic area levels summary

REUSE READINESS LEVELS VERSION 1.0 (APRIL 30, 2010)

Table 1 – Reuse Readiness Level (RRL) Topic Area Levels Summary

	Documentation	Extensibility	Intellectual Property Issues	Modularity	Packaging	Portability	Standards Compliance	Support	Verification and Testing
Level 1	Little or no internal or external documentation available	No ability to extend or modify program behavior	Product developers have been identified, but no rights have been determined.	Not designed with modularity	Software or executable available only, no packaging	The software is not portable	No standards compliance	No support available	No testing performed
Level 2	Partially to fully commented source code available	Very difficult to extend the software system, even for application contexts similar to the original application domain	Developers are discussing rights that comply with their organizational policies.			Some parts of the software may be portable	No standards compliance beyond best practices	Minimal support available	Software application formulated and unit testing performed
Level 3	Basic external documentation for sophisticated users available	Extending the software is difficult, even for application contexts similar to the original application domain	Rights agreements have been proposed to developers.	Modularity at major system or subsystem level only	Detailed installation instructions available	The software is only portable with significant costs	Some compliance with local standards and best practices	Some support available	Testing includes testing for error conditions and proof of handling of unknown input
Level 4	Reference manual available	Some extensibility is possible through configuration changes and/or moderate software modification	Developers have negotiated on rights agreements.			The software may be portable at a reasonable cost	Standards compliance, but incomplete and untested	Moderate systematic support is available	Software application demonstrated in a laboratory context
Level 5	User manual available	Consideration for future extensibility designed into the system for a moderate range of application contexts; extensibility approach defined and at least partially documented	Agreement on ownership, limited reuse rights, and recommended citation.	Partial segregation of generic and specific functionality	Software is easily configurable for different contexts	The software is moderately portable	Standards compliance with some testing	Support provided by an informal user community	Software application tested and validated in a laboratory context
Level 6	Tutorials available	Designed to allow extensibility across a moderate to broad range of application contexts, provides many points of extensibility, and a thorough and detailed extensibility plan exists	Developer list, recommended citation, and rights statements have been drafted.			The software is portable	Verified standards compliance with proprietary standards	Formal support available	Software application demonstrated in a relevant context
Level 7	Interface guide available	Demonstrated to be extensible by an external development team in a similar context	Developer list and limited rights statement included in product prototype.	Clear delineations of specific and reusable components	OS detect and auto-build for supported platforms	The software is highly portable	Verified standards compliance with open standards	Organized/defined support by developer available	Software application tested and validated in a relevant context
Level 8	Extension guide and/or design/developers guide available	Demonstrated extensibility on an external program, clear approach for modifying and extending features across a broad range of application domains	Recommended citation and intellectual property rights statement included in product.				Verified standards compliance with recognized standards	Support available by the organization that developed the asset	Software application "qualified" through test and demonstration (meets requirements) and successfully delivered
Level 9	Documentation on design, customization, testing, use, and reuse is available	Demonstrated extensibility in multiple scenarios, provides specific documentation and features to build extensions which are used across a range of domains by multiple user groups	Statements describing unrestricted rights, recommended citation, and developers embedded into product.	All functions and data encapsulated into objects or accessible through web service interfaces	Installation user interface provided	The software is completely portable	Independently verified standards compliance with recognized standards	Large user community with well-defined support available	Actual software application tested and validated through successful use of application output

10

Figure A.1: Reuse readiness levels topic areas Marshall et al. (2010)

B IPID headerfile

```
1  #ifndef INCLUDE_IPID_H_
2  #define INCLUDE_IPID_H_
3
4  #include <string>
5
6  /** @brief Reusable IPID Interface
7   *
8   * This interface class is used to have one interface for different
9   * implenetations
10  *  - PID
11  *  - PI
12  *  - PD
13  *
14  * @author Dennis Ellery <Dennisellery@gmail.com>
15  *
16  * @copyright Copyright (C) 2017 Ibotics
17  *
18  *   This program is free software: you can redistribute it and/or modify
19  *   it under the terms of the GNU General Public License as published by
20  *   the Free Software Foundation, either version 3 of the License, or
21  *   (at your option) any later version.
22  *
23  *   This program is distributed in the hope that it will be useful,
24  *   but WITHOUT ANY WARRANTY; without even the implied warranty of
25  *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    See the
26  *   GNU General Public License for more details.
27  *
28  *   You should have received a copy of the GNU General Public License
29  *   along with this program.  If not, see <http://www.gnu.org/licenses/>.
30  */
31 class IPID {
32  public:
33   //! Destructor
34   virtual ~IPID(){};
35
36   /** @brief Calculate next setpoint based on desired setpoint and current
37    * state.
38    *
39    *  @param[in] setpoint Desire setpoint
40    *  @param[in] previousValue Previous value of the plant.
41    *  @param[out] steerValue Steering value of plant.
42    *  @return Error code.
43    */
44   virtual int update(double setpoint, double previousValue,
45                      double &steerValue) = 0;
46
47   /**@brief set individual parameter
48    * @param[in] parameter string value of the parameter to be set
49    * @param[in] value The desired value to set.
50    * @return Error Code.
51    */
52   virtual int setParameter(std::string parameter, double value) = 0;
53
```

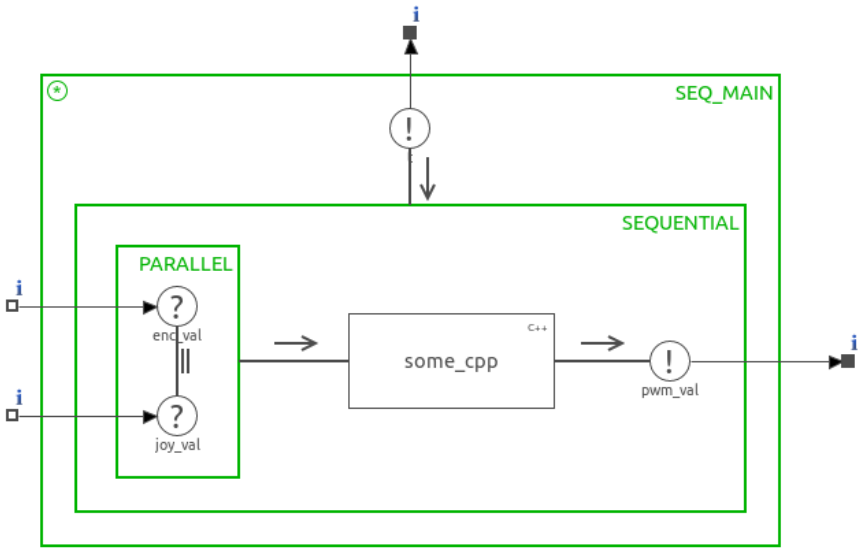
```
54   /**@brief Get current settings.
55    * @return A string containing all parameters and their values.
56    */
57   virtual void printConfig() = 0;
58 };
59
60 #endif  // INCLUDE_IPID_H_
```

Listing B.1: IPID header file

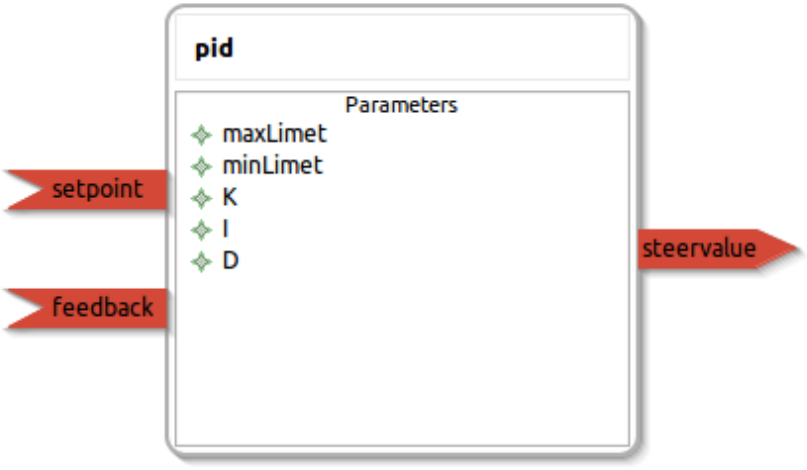
C Demo

A SISO PID controller was designed and integrated into two different middleware. The middleware models are given in figure C.1. The internal code is given in C.2.1.

C.1 Models



(a) TERRA model of PID controller



(b) BRIDE model of PID controller

Figure C.1: Models designed in middleware specific IDEs

C.2 Source code

C.2.1 TERRA

```
/**
 * Source file for the some_cpp model
 * Generated by the TERRA CSPm2LUNA generator version 1.1.3
 *
 * protected region document description on begin
 *
 * protected region document description end
 */

#include "some_cpp.h"
// protected region additional headers on begin
// Each additional header should get a corresponding dependency in the Makefile
#include "pid.h"
#include "IPID.h"
#include <strings.h>

// protected region additional headers end

namespace SomeModel { namespace some_cpp {

some_cpp::some_cpp(int &enc_val, double &joy_val, double &pwm_val) :
    CodeBlock(), enc_val(enc_val), joy_val(joy_val), pwm_val(pwm_val){
  SETNAME(this, "some_cpp");

  // protected region constructor on begin
  pwm_val = 0.0;
  enc_val = 0;
  joy_val = 0.0;
  PidInstance = new PID;
  // protected region constructor end
}

some_cpp::~some_cpp()
{
  // protected region destructor on begin
    delete PidInstance;
  // protected region destructor end
}

void some_cpp::execute()
{
  // protected region execute code on begin
//  set values
      // dt -  loop interval time
      // max - maximum value of manipulated variable
      // min - minimum value of manipulated variable
      // Kp -  proportional gain
      // Ki -  Integral gain
      // Kd -  derivative gain
    PidInstance->setParameter("dt", 0.01);
    PidInstance->setParameter("kp", 0.00003);
    PidInstance->setParameter("ki", 0);
    PidInstance->setParameter("kd", 0);
    PidInstance->setParameter("max", 0.3);
    PidInstance->setParameter("min", -0.3);
```

```
56
57      PidInstance->printConfig();
58      PidInstance->update(joy_val * 10000.0, enc_val, pwm_val);
59      printf("Hi there! Encoder value is: %d. Input joy value: %f. Output pwm is: %
60    // protected region execute code end
61  }
62
63  // protected region additional functions on begin
64  // protected region additional functions end
65
66  // Close namespace(s)
67  } }
```

Listing C.1: TERRA code some_cpp.cpp

*

C.2.2 ROS

```
1  // ROS message includes
2  #include "ros/ros.h"
3  #include <std_msgs/Float64.h>
4  #include <std_msgs/Float64.h>
5  #include <std_msgs/Float64.h>
6
7  /* protected region user include files on begin */
8  #include "pid.h"
9  #include "IPID.h"
10 /* protected region user include files end */
11
12 class pid_config
13 {
14 public:
15     int maxLimet;
16     int minLimet;
17     int K;
18     int I;
19     int D;
20 };
21
22 class pid_data
23 {
24 // autogenerated: don't touch this class
25 public:
26     //input data
27     std_msgs::Float64 in_setpoint;
28     std_msgs::Float64 in_feedback;
29     //output data
30     std_msgs::Float64 out_steervalue;
31     bool out_steervalue_active;
32 };
33
34 class pid_impl
35 {
36     /* protected region user member variables on begin */
37     IPID* controller;
```

```
38      double _setpoint;
39      double _feedback;
40      double _steervalue;
41      /* protected region user member variables end */
42
43  public:
44      pid_impl()
45      {
46          /* protected region user constructor on begin */
47          controller = new PID;
48          /* protected region user constructor end */
49      }
50
51      void configure(pid_config config)
52      {
53          /* protected region user configure on begin */
54          controller->setParameter("max",config.maxLimet);
55          controller->setParameter("min",config.minLimet);
56          controller->setParameter("kp",config.K);
57          controller->setParameter("ki",config.I);
58          controller->setParameter("kd",config.D);
59          /* protected region user configure end */
60      }
61
62      void update(pid_data &data, pid_config config)
63      {
64          /* protected region user update on begin */
65          double _setpoint = (double)data.in_setpoint;
66          double _feedback = (double)data.in_feedback;
67
68          controller->update(_setpoint, _feedback, _steervalue);
69
70          data.out_steervalue = (float) _steervalue;
71          /* protected region user update end */
72      }
73
74
75      /* protected region user additional functions on begin */
76      /* protected region user additional functions end */
77  };
```

Listing C.2: BRIDE code pid_common.cpp

Bibliography

Anguswamy, R., W. B. Frakes, G. M. Belli, I.-R. Chen, G. W. Kulczycki and O. Yilmaz (2013), *Factors Affecting the Design and Use of Reusable Components*, Ph.D. thesis, Virginia Polytechnic Institute and State University.
https://vtechworks.lib.vt.edu/handle/10919/23674

Bezemer, M. (2013), *Cyber-physical systems software development : way of working and tool suite*, Ph.D. thesis, University of Twente, Enschede, The Netherlands, doi:10.3990/1.9789036518796.
http://doc.utwente.nl/87731/

BRICS (2010), BRICS - Best practice in robotics.
http://www.best-of-robotics.org/home

Brugali, D. and P. Scandurra (2009), Component-based robotic engineering (Part I) [Tutorial], *IEEE Robotics & Automation Magazine*, **vol. 16**, no. 4, pp. 84–96, ISSN 1070-9932, doi:10.1109/MRA.2009.934837.
http://ieeexplore.ieee.org/document/5306930/

Bruyninckx, H., M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi and D. Brugali (2013), The BRICS component model: a model-based development paradigm for complex robotics software systems, *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pp. 1758–1764, doi:10.1145/2480362.2480693.
http://dl.acm.org/citation.cfm?doid=2480362.2480693

Cedilnik, A., B. Hoffman, B. King, K. Martin and A. Neundorf (2000), CMake.
https://cmake.org/

Elkady, A. and T. Sobh (2012), Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography, *Journal of Robotics*, **vol. 2012**, pp. 1–15, ISSN 1687-9600, doi:10.1155/2012/959013.
http://www.hindawi.com/journals/jr/2012/959013/

GitHub (2017), Choose an open source license.
https://choosealicense.com/

Google (2015), Google Test.
https://github.com/google/googletest

Google (2017), Google C++ Style Guide.
https://google.github.io/styleguide/cppguide.html

van Heesch, D. (1997), Doxygen.
www.doxygen.org

Marshall, J., S. Berrick and A. Bertolli (2010), Reuse Readiness Levels (RRLs), Technical report, NASA.
https://wiki.earthdata.nasa.gov/download/attachments/49446977/RRLs_v1.0.pdfhttps://wiki.earthdata.nasa.gov/pages/viewpage.action?pageId=49446977

Microsoft (2016), Visual Studio Code.
http://code.visualstudio.com

Object Management Group (1999), Object Management Group.
http://www.omg.org

OMG (2015), OMG Unified Modeling Language Version 2.5.
http://www.omg.org/spec/UML/2.5

Shakhimardanov, A., J. Paulus, N. Hochgeschwender, M. Reckhaus and G. K. Kraetzschmar (2010), Best Practice Assessment of Software Technologies for Robotics. http://www.best-of-robotics.org/pages/publications/BRICS_Deliverable_D2.1.pdf

Wilterdink, R. J. W. (2011), Design of a hard real-time, multi-threaded and CSP-capable execution framework, Technical Report 009, University of Twente, Enschede. http://essay.utwente.nl/61066/1/MSc_R_Wilterdink.pdf