The Functional Paradigm in Embedded Real-Time Systems

A study in the problems and opportunities the functional programming paradigm entails to embedded real-time systems

Emil Bergström    Shiliang Tong
Packsize Technologies AB, Mälardalen University
Innovation, Design and Engineering

Company supervisor: Stefan Karlsson
Examiner: Björn Lisper

October 21, 2014
Abstract

This thesis explores the possibility of the functional programming paradigm in the domain of hard embedded real-time systems. The implementation consists of re-implementing an already developed system that is written with the imperative and object oriented paradigms. The functional implementation of the system in question is compared with the original implementation and a study of code complexity, timing properties, CPU utilization and memory usage is performed. The implementation of this thesis consists of re-developing three of the periodic tasks of the original system and the whole development process is facilitated with the Test-driven development (TDD) development cycle. The programming language used in this thesis is C but with a functional approach to the problem. We conclusions of this thesis is that the functional implementation will give a more stable, reliable and readable system but some code volume, memory usage and CPU utilization overhead is present. The main benefit of using the functional paradigm in this type of system is the ability of using the TDD development cycle. The main con of this type of implementation is that it relies heavily on garbage collection due to the enforcement of data immutability. We find in conclusion that one can only use the functional paradigm if one has an over dimensioned system when it comes to hardware, mainly when it comes to memory size and CPU power. When developing small systems with scarce resources one should choose another paradigm.

Keywords. Functional paradigm, Embedded real-time systems, PLC, Memory management, Test-driven development.
Contents

1 Introduction 7

2 Problem Formulation 9

3 The Art of State 10

3.1 Why is software hard? 10

3.1.1 Complexity caused by state 10

3.1.2 Complexity caused by executional order 11

3.1.3 Complexity caused by code volume 11

3.1.4 Power corrupts 12

3.1.5 Breakthroughs that simplifies complexity 12

3.2 The concept of a programming paradigm 12

3.2.1 The imperative paradigm 13

3.2.1.1 Challenges 14

3.2.1.2 Advantages 16

3.2.1.3 Summary 16

3.2.2 The object oriented paradigm 17

3.2.2.1 Overview 17

3.2.2.2 Challenges 19

3.2.2.3 Advantages 20

3.2.2.4 Summary 21

3.2.3 The functional paradigm 21

3.2.3.1 Pure and Impure 21

3.2.3.2 Challenges 22

3.2.3.3 Advantages 23

3.2.3.4 Summary 25

3.3 Real-time Systems 25

3.3.1 Overview 25

3.3.2 Essential properties of real-time languages 26

3.3.2.1 Execution time measurability 26

3.3.2.2 Determinism 26

3.3.2.3 Memory management 27

3.3.2.4 Concurrency 27

3.3.3 Common practice paradigm in real-time systems 28

3.3.3.1 What makes imperative code suitable 28

3.3.4 The functional language paradigm in Real-time Systems 29

3.3.4.1 The unreasonable barriers 29

3.3.4.2 The real barriers 30

3.3.4.3 Why should we care? 31

3.3.5 Functional languages for the Real-time domain 32

3.3.5.1 Hume 32

3.3.5.2 Safe 34

3.3.5.3 Nitro 34
### Core features of functional languages

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 State and Side Effects</td>
<td>37</td>
</tr>
<tr>
<td>4.1.1 Immutable Data</td>
<td>37</td>
</tr>
<tr>
<td>4.1.2 Monads</td>
<td>38</td>
</tr>
<tr>
<td>4.2 Programs as data: Understanding the function pointer</td>
<td>39</td>
</tr>
<tr>
<td>4.3 Higher order functions</td>
<td>40</td>
</tr>
<tr>
<td>4.3.1 Cons, Car and Cdr</td>
<td>41</td>
</tr>
<tr>
<td>4.3.2 Map</td>
<td>42</td>
</tr>
<tr>
<td>4.3.3 Filter</td>
<td>43</td>
</tr>
<tr>
<td>4.3.4 Fold</td>
<td>44</td>
</tr>
</tbody>
</table>

### Case Study

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Goals</td>
<td>46</td>
</tr>
<tr>
<td>5.2 Expected outcome</td>
<td>46</td>
</tr>
<tr>
<td>5.3 Current system</td>
<td>47</td>
</tr>
<tr>
<td>5.3.1 Software</td>
<td>47</td>
</tr>
<tr>
<td>5.3.1.1 Platform</td>
<td>47</td>
</tr>
<tr>
<td>5.3.1.2 Memory model</td>
<td>47</td>
</tr>
<tr>
<td>5.3.1.3 System</td>
<td>47</td>
</tr>
<tr>
<td>5.3.2 Hardware</td>
<td>48</td>
</tr>
</tbody>
</table>

### Implementation

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Introduction</td>
<td>51</td>
</tr>
<tr>
<td>6.2 Implementation Rules</td>
<td>51</td>
</tr>
<tr>
<td>6.3 Services and Facilities</td>
<td>51</td>
</tr>
<tr>
<td>6.3.1 Unit testing</td>
<td>51</td>
</tr>
<tr>
<td>6.3.2 Immutable Data</td>
<td>52</td>
</tr>
<tr>
<td>6.3.3 Garbage collector</td>
<td>52</td>
</tr>
<tr>
<td>6.4 Implemented tasks</td>
<td>56</td>
</tr>
<tr>
<td>6.4.1 General implementation approach</td>
<td>56</td>
</tr>
<tr>
<td>6.4.2 KnifeCyclic</td>
<td>57</td>
</tr>
<tr>
<td>6.4.3 IOControllerCyclic</td>
<td>57</td>
</tr>
<tr>
<td>6.4.4 HttpClientCyclic</td>
<td>59</td>
</tr>
</tbody>
</table>

### Testing

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Introduction</td>
<td>62</td>
</tr>
<tr>
<td>7.1.1 Test cases</td>
<td>62</td>
</tr>
<tr>
<td>7.1.1.1 Code metrics</td>
<td>62</td>
</tr>
<tr>
<td>7.1.1.2 CPU utilization</td>
<td>65</td>
</tr>
<tr>
<td>7.1.1.3 Memory usage</td>
<td>65</td>
</tr>
<tr>
<td>7.1.1.4 Response and Execution time</td>
<td>66</td>
</tr>
<tr>
<td>7.1.1.5 User evaluation</td>
<td>67</td>
</tr>
<tr>
<td>7.2 Testing strategy</td>
<td>67</td>
</tr>
</tbody>
</table>
H  Code metrics for the original version of KnifeCyclic  112
I  Code metrics for the functional version of KnifeCyclic  113
J  Code metrics of functional KnifeCyclic without libraries  114
K  Code metrics for original IOController  115
L  Code metrics for functional IOController  116
M  Code metrics of functional IOController without libraries  117
N  Code metrics for HttpClientCyclic  118
O  Code metrics of functional HttpClient without libraries  119
1 Introduction

The functional language paradigm is gaining attention in the software market [89] due to the benefits it imposes on handling of state and referential transparency. Those two benefits yield more stable systems and systems that are easier to reason about as a programmer and or as a compiler [70].

According to M.Wallace [87], functional languages are generally used in workstations with large memory, big disks and fast processors. He claims that specialized hardware for functional languages is not viable and therefore better implementations of functional languages are needed on existing hardware. He also points out that the domain of embedded systems has been incompatible with the functional paradigm so far. But why is that so? In this M.Sc. thesis, the authors will try to answer this question.

The purpose behind this thesis is to investigate what the functional programming paradigm could yield to embedded computing and industrial systems in the hard Real-Time Operating System (RTOS) domain. How will a change in paradigm and approach to problems suitable for embedded systems affect the finalized products when it comes to testability and maintainability. The work in this thesis is performed by two master level students studying embedded systems and applied software engineering at Mälardalen University, Sweden.

The authors has in this thesis used an imperative language, C, and tired to impose the functional approach to computational problems. An already built system was used as a reference. The authors redeveloped this system with a functional frame of mind and the whole development process was facilitated with the TDD development cycle. This investigated whether or not it is possible to use TDD when it comes to systems that are very close to its underlying hardware.

The evaluation of this thesis consisted of static code analysis where code metrics were calculated with the developed source and the original system. Runtime analysis was performed where CPU usage, memory usage and timing properties was some of the topics of interest. A user study was also preformed where experienced software developers looked at the code base and grade it on a scale from 1 to 10 based on I.e. readability, testability, quality etc. The two implementations was then compared with these metrics in mind.

This paper is organized in the following way, the thesis proposal with a list of questions to answer in this thesis is given in Section 2. Section 3 focuses on trying to explain the problems that software developers are faced with when it comes to handling of imperative state and what type of consequences these issues can impose on the source written. Section 3.2 gives a brief introduction to a programming paradigm as a concept and three common paradigms are briefly explained. In Section 4, a brief introduction is given on how to perform some of the core concepts of the functional programming paradigm in the imperative language C. In Section 6.2, in order to enforce the functional programming paradigm when using C, the rules we restrictly observed in implementation are presented. In Section 5 is the case study of this thesis presented. This de-
scription contains both a description of the hardware used and the software of the original system. This section also contains case study specific question. In Section 6 is the functional implementation of thesis explained. In Section 7, a description of the different metrics that is used in this thesis to measure the difference in quality between the original implementation and the functional one is given. Section 8 contains the data collected and comparisons between the functional system and the original system. At the end is our conclusions including answers to the question posed in Section 2 and Section 5, the limitations of this thesis, its implications and a description of the future work that needs to be done.
2 Problem Formulation

This thesis is a study of what pros and cons the functional programming paradigm entails to a hard RTOS. Questions to answer in this thesis include:

1. Is the functional programming paradigm suitable for embedded/real-time systems?

2. What are the trade offs for the functional language paradigm compared with the imperative or object oriented language paradigms in the real-time domain?

3. Will a functional implementation have a higher or lower complexity? Where would the complexity lie?

4. Will the functional paradigm yield more readable, stable and reliable systems, compared with similar systems?

5. How will the code metrics differ for a system written with the functional paradigm compared with a similar system in either the object oriented or imperative paradigm?

6. Is it beneficial to combine the functional paradigm with other paradigms?

To answer these questions a state of the art survey is done in Section 3 and a case study, which is explained in Section 5.
3 The Art of State

3.1 Why is software hard?

According to F P. Brooks [55] there are four reasons for why building software is hard, those are complexity, conformity, changeability and invisibility.

Complexity is the problem that emerges in software when it becomes hard to understand and reason about.

Conformity is the challenge with software being written by humans and that developers solve problems in their own way and there is nothing governing how one should solve a specific problem.

Software is always under the pressure of change and how to handle changes. Without good changeability in software customers will not be satisfied with the time overhead imposed for new functionality requests. Software also has to handle environment changes such as new disks, new processors, new screens and so on.

The problem with invisibility in software comes from the natural way of how people structure a problem. One draws a floor plan for a house or maps over an area. This is not easy in software since there are so many ways to draw these overviews. One could picture for instance a chart for data flow, dependencies, executional flow, name-space and others. This makes software hard to model.

According to B. Mosely and P. Marks [70] complexity is the only significant reason to why building software is hard. Being able to think and reason about a system is crucial to be able to deliver robust and reliable software.

There are several causes of complexity. Some of them are:

1. State
2. Executional order
3. Code volume
4. Power corrupts

They all reside in the domain of state but are different versions of the consequence of state. In todays modern world, with concurrent systems and multi core computers, state causes more and more problems to software [70]. Incoherent state, deadlocks, race conditions, mutual exclusion are some of the problems that one has to deal with when building a concurrent system.

3.1.1 Complexity caused by state

What is state? State is the current contents of each memory cell in a computer, it is the current configuration of all the information that is available to the computer at a point in time. The reason why this makes a software system complex is that the outcome of a system is often based upon said state. Executional paths and calculations are based upon state. Since each bit of state added to
the system doubles the amount of possible states of the system it is very hard to keep track of all possible states and their effect on the outcome of the system.

A system that uses 1 kilobyte of data memory in total will have \(2^{8192}\) states in total that can affect the final outcome of the system. This makes testing a state dependent system comprehensively very hard and/or time consuming. In general when you are testing a system you supply input, observe the final output and verify that it is the expected outcome. When defining if a system has good test coverage one can measure the percentage of lines of code run through your test suite. But what does this really tell you about the test coverage of system as a whole? You can have 100% code coverage but only testing in worst case one state in the system. Is this a fully tested system? One could argue that this is a topic of major concern.

3.1.2 Complexity caused by executional order

Most common programming languages, i.e. C, Java, PHP and Javascript [77], puts importance in the executional order of a system. It is controlled by what order the instruction statements in the system were written. This order is then modified with conditional statements and subroutines. The problem with executional order is that a programmer has to understand the consequences each executional order impose on the system even in the parts of the system that are unrelated to the work he is currently doing. B. Moseley and P. Marks explained the consequence of this perfectly in [70, p.9].

When a programmer is forced (through use of a language with implicit control flow) to specify the control, he or she is being forced to specify an aspect of how the system should work rather than simply what is desired.

I.e specifying the what instead of the how is the point of declarative approaches i.e. XAML, Datalog, Logic programming etc.

This is the simplest part of the problem with executional order. If we add the problem of concurrency we can easily see that the problem with executional order increases. Concurrency exponentially increases the importance of executional order due to data being shared between threads and therefore can the state changes in thread A affect the outcome of thread B. This is therefore the root of some hard to find bugs where the state of the system is altered to a incoherent one because of a complex sequence of operations where timing is very important.

3.1.3 Complexity caused by code volume

This one is pretty self explanatory. A bigger system will be harder to understand compared to a small system. Code volume will also increase the complexity caused by state management and complexity caused by executional order. More code means more state management and more ways a program can walk
down its execution path and therefore it makes the system inherently harder to understand.

3.1.4 Power corrupts

This problem has mainly to do with the nature of people and project management. In the context of software development there are always deadlines and time pressure to be able to hit the next release. People under time pressure will often cut corners to be able to fit the time schedule imposed by management. Therefore the more a language permits the more it will be abused. If a language does not enforce rules upon its users that discourage mistakes there will be potential for their making these mistakes. If we take garbage collection as an example. With manual memory management in a language one gives the responsibility and power of cleaning up after himself to the programmer instead of having language enforced rules that do this automatically. Programmers can forget to free allocated memory blocks since they are human, but a garbage collector will not. The more a language permits when it comes to corner-cutting the more mistakes will be made potentially.

3.1.5 Breakthroughs that simplifies complexity

Over the years some breakthroughs have been made when it comes to the handling of complexity. High level languages, that enforce rules upon the programmer so he or she can’t mutate state without good reason, have been developed. For instance languages with garbage collection like Java and C#. Time sharing where the time for the compilation to deployment cycle has been severely reduced so the programmer can keep his train of thought when he is testing out his changes, instead of having to remember each incremental change. Unified programming environments have reduced the complexity caused by the management of dependencies in software.

3.2 The concept of a programming paradigm

The concept of a programming paradigm might not be of common knowledge therefore it is explained in this section.

A programming paradigm is the way of building the structure and elements of a computer program. One could say that the paradigm defines the style of the code and the way that problems are solved. Different programming languages support different paradigms. Some support just one while others support a collection of paradigms i.e. Haskell is a pure functional language and therefore only supports that paradigm, C# can support both the object-oriented paradigm and parts of the functional paradigm with the introduction of Lambda and Higher-order functions, in C# v3.0 [60, p. 400], into the language.

There are six main programming paradigms:

1. Imperative
Three of these are briefly explained in sections: 3.2.1, 3.2.2 and 3.2.3.

1. Imperative
2. Object oriented
3. Functional

These three are explained since the basis system of the case study uses the two first ones and the last one is the focus of this thesis.

3.2.1 The imperative paradigm

The imperative programming paradigm can be summarized the same way a normal cake recipe works, a sequential execution of steps as a function of time. The word imperative is derived from the Latin word "imparare" which means to command in this sense a command corresponds to each step of the cake recipe [8].

The imperative programming paradigm has its basis on the von-Neumann architecture of a computer [45].

The von-Neumann computer architecture consists of an instruction processor and a memory, where the processor and the memory are connected through a bus. These components can then perform four actions, read, write, compute and branch. With a sequential combination of these operations software is built. Each location in the memory can store one value and one can only save information to memory by overwriting already existing information with a destructive write [97].

When solving a task with the imperative programming paradigm three characteristics can be seen:

1. State
2. Sequential order
3. Assignments

In this sense, state corresponds to the content in each memory cell. Some instructions are based on the current memory contents or state and some instructions change the current state by overwriting the contents of a particular memory cell.
All instructions in a von-Neumann computer are performed in sequential order. Since a von-Neumann computer generally only has one instruction processing unit or core it can only perform one instruction at a time. Since some instructions can change or are based upon state the order in which the instructions occur is very important, an example of this can be seen in Figure 1. In this example we can see that if the order would not be enforced and the branch instruction would occur before the addition of 1 to $t0 than the return address in $ra would not be set to 100 which was the desired outcome and we would end up in the wrong part of the program.

Assignment statements are used to change the state of the system by overwriting the current state with new information. As a consequence these operations also destroy the old state.

The most notable fact about the imperative paradigm is that all other paradigms in any high level language will still be compiled down to a low level imperative language that can be read by the processor. Due to the fact that processing units can only execute the assembly instructions designed for their architecture and these assembly languages are always imperative languages. So all other paradigms are just abstractions on top of the imperative paradigm that handle the problems of the imperative paradigm in different ways, and these problems are described in Section 3.2.1.1.

The imperative programming paradigm does not group memory into blocks that define different states. In the imperative paradigm objects do not exist, memory is just memory with stored data. If one has these kinds of groupings: objects, classes, encapsulation or something similar one does not belong to the imperative paradigm.

3.2.1.1 Challenges The challenges of the imperative programming paradigm are many but in this paper five are presented.
1. State
2. Protection
3. Readability
4. Re-usability
5. Abstraction

In the succeeding paragraphs an explanation of each challenge is given.

**State** Since the different instruction types in the imperative programming paradigm can be based on or alter state it is a state-full paradigm. The challenge with this is that testing a state-full system is hard. Each binary bit of state doubles the amounts of states that are possible in the system. This makes state-full systems almost impossible to fully test in all areas. As mentioned in Section 3.1.1 one has to ask the question about what kind of metrics one should use when analyzing how well tested ones system is. Code coverage is one, but what does that say about your system really? 100% code coverage tells you that every line is run in your test suite but it does not tell you anything about how many states that have been tested. One can have full code coverage with only a small fraction of the possible states of the system run.

Since a specific state can be the product of a complex sequence of actions, that are rarely performed, a state-full system will often have hard to find bugs. The usual solution to these bugs is a restart of the system to get to a known state.

As soon as someone recommends a reboot you are dealing with the side effects of a state-full system.

**Limited protection** The imperative programming paradigm has the least amount of protection of its state compared to other paradigms. The imperative paradigm does not impose any restrictions on manipulating state this creates an environment which is prone to accidental manipulation. This will further increase the downside of it being a state-full paradigm. Unconditional branching and global variables are free to dominate the system. Other paradigms can put restriction on this, i.e. in the structured programming paradigm goto statements are removed. The purpose of these restrictions in other paradigms are to help with unintentional state mutation.

**Limited readability** Readability has never been important to the imperative programming paradigm [97]. Recognition of the software crisis caused engineers to reevaluate the need for readable code since this could reduce the amount of errors during development. Spaghetti-like goto statements will cause the system to be hard to reason about and therefore very hard to understand. An example of this problem can be seen in Figure 2
Limited re-usability  Since the imperative paradigm does not contain any functions, procedures or subroutines code is hard to reuse. The only way to reuse code is a branch statement that moves the program counter back or forward to the instruction sequence you would like to perform on the current state. Other paradigms are much better at re-usability i.e. the object oriented paradigm have classes and objects that can include methods and procedures to be called upon a passed state.

Limited abstraction  Since the imperative paradigm does not group state into objects almost no abstractions are used. No inheritance, no procedures, no locks, no subroutines and so on. All of these abstractions can be found in other paradigms.

3.2.1.2 Advantages  The advantages of the imperative paradigm are few and in this paper two are presented.

1. Efficient 

2. Close to the hardware

The imperative paradigm can be a very efficient paradigm. Since all of the code is written very close to the hardware itself a lot of optimizations that a high level compiler would not find can be made. This is the upside with giving total control to the development team. If one has a very good team, it can produce very efficient software. But this goes both ways, if the team is not that good the software produced will not be any good.

3.2.1.3 Summary  As said in the beginning of this section. The imperative programming paradigm is equivalent to a cake recipe. It is a sequential execution
of steps as a function of time. Where each step can alter the current state of the
system i.e. adding more flour will change the state of the cake mix in the same
way as writing a value to a memory cell. In this sense all state modifications are
also destructive. One can not easily go back to an earlier state. This holds both
for cake mixtures because it is hard to remove already added flour and also for
memory since one cell can only hold one value and it does not store old values.

The main advantages with the imperative paradigm are that it is efficient
and a lot of optimizations can be made since the code is really close to the
hardware. Main disadvantages include limited protection of state and limited re-usability.

3.2.2 The object oriented paradigm

The term ”objects oriented”, in the context of computing systems, first appeared
at MIT in the late 1950s and early 1960s. The first programming language
that introduced objects in the notion of classes was Simula 67 in the 1960s,
which was designed for discrete event simulation. Simula, as the pioneer of pro-
gramming languages that introduced the object oriented paradigm, influenced
the later members of the object oriented language family, including C++ and
Smalltalk [74, p. 123].

In the early and mid 1990s, the object oriented paradigm started to grow
to become an outstanding programming methodology [74, p. 122] and contin-
ued during the beginning of the 21st century. One of the main factors that
pushed the object oriented paradigm to the programming mainstream was the
increasing popularity of Graphical User Interfaces(GUI), which rely heavily on
object oriented ideas. For instance, Cocoa frameworks, which is Apple’s native
application programming interface for the OS X operating system, introduced
a dynamic GUI library and an object oriented language written in Objective-C.
With the success of the object oriented paradigm, more and more previously
existing languages like BASIC were improved with object oriented features [74,
p. 132]. This led to the significant position of the object oriented paradigm as
seen in today’s systems.

3.2.2.1 Overview The object oriented paradigm, general speaking, is a
prominent software development paradigm where the focus lies on objects which
are usually instanced as classes. The objects consist of data fields which are at-
tributes used to describe the object, and associated operations known as meth-
ods used to manipulate the attributes of the object. Amongst current popular
languages, Java, C++ and Python are examples of object oriented programming
languages.

Compared with imperative systems, rather than to structure programs as
code and data as in the imperative paradigm, an object-oriented system treats
the two as one entity and builds the system with a collection of interacting
entities known as objects.

To facilitate the implementation of objects, the object oriented paradigm has four main fundamentals compared with the imperative paradigm which are
encapsulation, abstraction, inheritance and polymorphism.

**Encapsulation**  Is the feature that the internal representation of an object is protected by a black box. External access to the object can be achieved only through a set of public methods, which is generally called a class interface. The methods are a set of functions designed to ensure correct usage of the corresponding data inside of the object. That is to say the protected data fields defined in an object can only be modified or manipulated by its own methods. With the implementation of this methodology, it improves the security of data fields in the objects by preventing the internal state from being modified by accident [74, p. 133].

In addition, encapsulation makes the system more modularized and reusable, thus enhancing the robustness of the system.

**Data abstraction**  The definition of data abstraction is to only show the necessary parts of an object to the user. Through development of object interfaces, abstraction denotes a representational model of an actual item that could be found in the real world. I.e. when we turn on a computer, we are only interested in the actuation and the outcome. We are not aware nor do we care about the inner changes of the computer as long as we get the expected outcome. As the definition from Grady Booch in his book Object-Oriented Analysis and Design With Applications [31, p. 38]:

"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer."

However data abstraction is not a one way street towards the Holy Grail. Advocates of the object oriented paradigm speaks of objects as a nice and complete description of the real world. That is true for actual objects in the world, like a chair, but when one is forced by the environment to place all logic inside of objects one has to think about objects that only hold logic. So the mapping towards the real world is somewhat fuzzy. Its easy to describe a chair in an object with color and weight as attributes, which makes it easy to visualize, but a manager that keeps track of relationships between objects is hard to get a visual grasp on.

**Inheritance**  Was designed to facilitate the development of objects that share some common features or behaviors. The point of inheritance is to allow software developers to not implement common features and behaviors in every object from scratch. Instead of duplicating the developing processes, inheritance allows a developer to inherit data and functions from other classes, which is also called parent, super or base classes.

The parent-child relationship can be used to develop a hierarchical class structure, like a genealogical tree. The biggest benefit from this relationship is
that much effort will be saved when creating a new object which shares some aspects, i.e. attributes or methods, with other objects [74, p. 133].

**Polymorphism** Can be understood as multiple implementations of an object interface. General speaking, in the real world many behaviors consisting of different procedures are named identically. However, the users of this type of behavior don’t care about the details of its implementation, they simply care about the result from this behavior. For instance, a method in a language class named Translate2English, the implementation and operations of this method are not identical between different language classes, but it is feasible and convenient for management to assign those methods the same name. For the code reader, readability will be improved after implementing this principle. For the programmer who uses objects from others, it eases the process of learning. At the same time, polymorphism helps to hide the implementation details.

These four fundamentals of the object oriented paradigm are closely tied with each other. Abstraction together with encapsulation hides the internals and allows programmers to work through abstract interfaces. Inheritance makes it possible to inherit members from other objects. Polymorphism allows working with objects through their parent interface and invoke abstract actions [38].

### 3.2.2.2 Challenges

The characteristics of the object oriented paradigm generates some challenges. In this paper three are presented.

1. **State**
2. **Control**
3. **Hierarchy**

**State in object oriented systems** Object oriented systems suffer the same problems when it comes to state as a system developed with the imperative paradigm. The benefit in the object oriented world is encapsulation. In the imperative paradigm one has no way to protect state, since it’s just data in memory. With the object oriented approach one can hide parts of the state in a system by encapsulating them in objects with a strict interface of methods that is the only way to mutate the protected state of the object. Protected state is an option though, and not enforced by common object oriented languages, i.e Java or C#, as a default. This relieves some parts of the problems with accidental state mutation since the interface enforces rules upon how state can be mutated but it does not remove the problems completely. Since the problem is not solved with encapsulation tools to help with state dependent testing are available such as Pex for the .Net platform [79].

**Executional control in object oriented systems** Executional order is equally important in an object oriented system as it is in an imperative systems. They suffer the same consequences, as seen in Figure 1, if executional order is
not enforced in the same sequential way the program was written for. Object oriented systems also suffer when it comes to concurrency since data or objects can be shared between threads. Therefore concurrent object oriented systems often have hard-to-find bugs that are a product of a complex sequence of actions as a function of time. These bugs often reside latent in code for a long time until the right sequence of actions are performed, as mentioned in Section 3.1.2. There are tools to help with the testing of concurrent systems though, one of them are Microsoft's Chess [78].

**Hierarchy** Inheritance is created to simplify the building process of objects. Nevertheless the implementation of inheritance on multiple levels will increase the challenges of maintenance. As a system grows there is a possibility that a complex hierarchy of inheritance will emerge. This will directly affect the maintainability of the system since engineers will need to sort out the trail of inheritance to be able to effectively maintain the system [56, p. 115].

### 3.2.2.3 Advantages

There is no doubt that the object oriented paradigm relieves some of the challenges to software, when it is implemented in a right way. This is also the reason why the object oriented paradigm is so well accepted in the software development community. Some of the benefits of the object oriented approach are presented within the three aspects below:

1. Maintenance
2. Code reuse
3. Quality

As mentioned in Section 3.2.2.1 an object instanced as a class has defined behaviors that we can use to access the state of the object, without caring about the inner details of the implementation. It provides a good framework for building code libraries where supplied objects can share the signature of methods but the inner operations are different. Thus it facilitates simplification on maintenance since the programmer only needs to code with already existing objects.

Abstraction together with the inheritance principle will simplify code reuse. Adding additional features to an existing class can be achieved by inheriting from a superclass without the need for modification. Properly written classes eases the modularization of programs through the division of complexity into small parts.

Because of the great amount of re-usability that the object oriented approach brings to software one could argue that the development time and costs associated with new functionality should decrease over time during a project. By reusing existing well-tested programs the quality of new software is increased. It also reduces some of the effort needed in testing new software. As mentioned above, lower development cost and faster development due to code reuse allows more time and effort to be assigned to quality-oriented processes, i.e. system
architecture or design. Without doubt, software written in the object oriented paradigm will produce a higher-quality of software compared to the imperative paradigm [74, p. 132-139].

3.2.2.4 Summary The essence of the object oriented paradigm can be summarized in one word, encapsulation. With the usage of objects one can encapsulate and hide state from user and therefore control how it is being used. This will constrict users in their usage of objects and inherently prevent them from abusing the state and control of the system. For deducing software complexity, the object oriented paradigm does not make the program simpler, but it makes the usage of state simpler. As a fact, object oriented programs still suffer greatly from both state-derived and control-derived complexity.

3.2.3 The functional paradigm

The functional programming paradigm has been popular in the scientific domain for decades but it’s just recently that it has become popular amongst commercial programmers [51]. Functional languages in the scientific domain have mainly been used for symbol processing. The most popular application is probably Mathematica.

The fundamental principle in functional programming is that computation is realized by composing functions in the mathematical sense. In the mathematical sense of functions inputs are mapped to an output. The most important part is that functions always produce the same output given a specific set of input parameters and that they have no side-effects, they do not affect the state of the system, they do not write to files, they do not change memory. This corresponds to the mathematical definition of a function.

Since functions can’t have side-effects on state in a pure functional languages one can’t use loops as the primary iterative process since they have the need for changes in state to be able to terminate. Therefore recursion is the basis in all iterative processes in a pure functional language.

In a functional language everything is considered data, even functions. It is common in functional languages to write functions that take functions as arguments and returns a function as the output. These are called higher order functions. This is a benefit given by the usage of referential transparency which is explained in Section 3.2.3.3.

3.2.3.1 Pure and Impure Functional languages can be classified as pure or impure languages. The difference between pure and impure is the languages attitude towards mutable state. Amongst pure languages we have Haskell and Miranda. In Haskell mutable state is only permitted within special language constructs permitting the compiler to verify the absence of state mutation in other parts of the system. If we look towards impure languages such as Scheme or Clojure, two lisp dialects, one has functions that permit mutations of state. These functions are signified with an exclamation mark to notify the programmer to watch out so accidental state mutation does not occur. This puts the
responsibility of managing imperative state on the developer since it is not enforced by the language.

3.2.3.2 Challenges  The challenges of the functional paradigm and functional languages resides mostly in people not being used to, in the context of programming, the functional way of thinking. The concept of not mutating state in ones functions seems strange to an imperative programmer, which is the majority. How does one really do anything without changing the state of the machine which is running the program? The answer is that state mutation is not completely abolished but put under large restrictions. Below three challenges for functional languages are presented.

1. The Blub paradox  [46]
2. Readability  [73]
3. Modularity  [80]

The Blub paradox  The Blub paradox states that a programmer looking at other languages or paradigms than the one he is mostly familiar with has a hard time to see the benefits of the said languages or paradigms.

Programming languages can be arranged in a power continuum. Languages of low power i.e. assembly is placed at the bottom of the continuum and high-level languages like lisp are placed at the top of the continuum. If a programmer is familiar with a hypothetical language, Blub, at the middle of the continuum. If he looks downwards in the continuum he will see languages with missing features that he depends on. If he looks upwards to the top of the continuum our Blub programmer will not realize that he is looking up. All he will see is weird languages with added functionality that is unnecessary to him. This is one of the problems with functional languages. Imperative programmers have a hard time seeing the benefit of the restrictions put upon them when it comes to solving a certain problem.

Readability  As Martin Fowler famously said [43],

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

One of the disadvantages of functional programming presented by most naysayers is that functional languages are not mapped to human language constructs but to mathematical constructs. This entails code that is harder to read at a glance due to the extensive usage of recursion and expressive code. One will encounter denser code that does more in each line instead of the almost novel like code in object oriented systems where each intricate detail is explained with easy to understand keywords.
Modularity  The problems with modularity in pure functional languages has to do with named state. When it comes to modularity we look at the ability to change one module of a system without the need to change other modules because of the initial change.

If we take a system with three modules, A, B and C, as an example, each is developed independently and they resides in independent object files. Module A and C uses functionality in module B.

After module A and B are finished a request to add a call count is issued to one of the functions in module B because module C needs that count for its logical operations.

In an object oriented system one would simply add that variable to the object containing the method which needed the call count and add corresponding getters to add the ability to retrieve the information. Module C would later on call those getters to retrieve the information needed. This would not require any changes in module A since the signatures on the methods in module B did not change.

In a pure functional language one would need to pass down a counter from module C to module B and increase that counter every time the specific method is called. With this solution a change to module A would also be necessary since the signatures on the methods in module B changed when the counter was added.

In this example we can see the effects of not having named state and its effects on modularity in a language. This could be a big problem if several modules need updates due to a change in a single module. One could argue that this would be a design flaw from the beginning but nonetheless the effects of not having named state affects modularity.

3.2.3.3 Advantages  The advantages of the functional language paradigm are many but in this paper four are presented. These are:

1. Restrictions on mutable state
2. Concurrency
3. Referential transparency
4. Testability

Restrictions on mutable state  Since pure functional languages do not permit functions with side-effects one has put large restrictions on state mutation. This abolishes the occurrence of accidental mutation since in pure languages like Haskell mutation is only permitted in certain areas and in impure languages like Scheme one has identifiers in the call signature that serve as a warning for state mutation. This relieves some of the problems caused by complexity explained in Section 3.1.1.
**Concurrency**  Since the performance of a single processor is not increasing the way it did a couple of years ago computers are incorporating more processing power by adding more processing units or cores to the same computer. To be able to utilize this technology shift in the correct way software has to be able to fire away several executional threads that walk down different paths in the source at the same time, this is called concurrency [32, p. 11].

The big problem with concurrency is the ability to maintain data in a usable state. Since several threads are accessing the same memory blocks at the same time problems with reading and writing can occur, i.e. if thread A reads a value from a cell while thread B writes to the same block. The main issue with these kind of problems are that they are a product of timing. They can exist for a very long time in the system before they cause any problems, and inherently they are very hard to find.

Since pure functional languages have no variables there are no problems with how data is handled in memory. This also yields a low importance of the order of execution, so two threads working on the solution to the same problem will not affect each other since all the data used to solve the problem in question is exclusive in the context of each thread. Therefore functional languages are a promising paradigm as they are able to better utilize a computer with several processing units.

**Referential transparency**  The definition of referential transparency is stated as: "An expression is said to be referentially transparent if it can be replaced with its value without changing the behavior of a program" [82]. In mathematics all functions are referentially transparent and since functional programming has its basis in lambda calculus all functions in a pure functional language are also referentially transparent. This is due to the fact that functions in a functional language are first class values, exactly in the same way as value-types such as integers or boolean, therefore the substitution needed to be defined as referentially transparent is possible.

There is no difference in passing the function (+ 1 2) or passing the value 3 in a functional language. In some languages, i.e. Clojure, there is also an option to buffer input arguments with their corresponding result, called memoization, so lengthy computations are not needed to be performed more than once for each set of input parameters.

The benefit of referentially transparency is that it allows the programmer and the compiler to reason about program behavior since all the information available to a function is in its arguments. As a programmer reading referentially transparent functions allows one to disregard stuff outside of the scope function since the functions themselves have no side effects or dependencies to the system as a whole.

**Testability**  Due to the benefits of referential transparency presented in Section 3.2.3.3 one can easily see how the testability aspect of a system would increase with the usage of the functional programming paradigm. Since func-
tions have no side effects one does not have to create test cases for unusual consequences of state. Since errors due to a incoherent state can be the product of a complex set of actions they are inherently very hard to test. In the functional paradigm one only need to test functions in the "given these parameters do I get this result" method. One does not need to worry about fringe cases due to hidden state in the same way as in an object-oriented or imperative system.

When it comes to more concrete ways of testing functional systems there are lightweight tools for random testing such as QuickCheck [35] for Haskell and test.check [44] for Clojure. Random testing is the act of generating input for a function and defining properties that the output must fulfill to pass the test. Pass criteria could for instance be no value of the output can be bigger than 42 or this string "whats up doc?" can’t be within the output of the function.

As an example one could generate a random set of integers, i.e. [5, 4, 87, 12, -5, 23], and send that set to a function that sorts the values in ascending order. The pass criteria for this test would be that the first item of the list is equal to the lowest value of the list. This could then be recursively called over and over while removing the first element with each call and therefore verify that the list is sorted accordingly. The benefit of this type of random testing is that over time one will test more and more of the input range and hopefully hit all fringe elements.

3.2.3.4 Summary The functional paradigm is a mathematical based model with emphasis on referential transparency, restrictions on mutable state and concurrency. Since state mutation is only permitted in certain areas in pure functional languages and signified in the function signatures in impure functional languages it is easy for a programmer to guard himself and his software from accidental state mutation.

Due to referential transparency in a functional language testing is made easier. Since state mutation is not allowed one only needs to test the input and the output of a function and verify that each function behaves according to the way it was designed.

More on the functional paradigm and some important features of functional languages is given in Section 4.

3.3 Real-time Systems

3.3.1 Overview

There is a sub-category of embedded systems called real-time systems. In this category not only is the outcome of the system important but also the timeliness of it. Embedded systems are widely used all over the world, they are small computer systems designed for a specific purpose. They range from small portable devices like digital watches or IR remote controls, to large appliances such as televisions or washing machines, they are everywhere. An embedded system is a computer that does not look like a normal computer.
If we look towards real-time systems there are a couple of properties that are important for an embedded system to be classified as a real-time system. As mentioned above one important aspect of a real-time system is time or more precisely timeliness. To a real-time system the time a result is delivered is equally important as the result itself. If we take a mp3 player for instance if the signal to the headphones aren’t delivered in the right time the music will not sound as it should.

Real-time systems can be divided into two categories, hard and soft real-time systems. The difference between them is their attitude towards the importance of timing constraints [34, p. 1-7].

In a soft real-time system timing constraints are only a matter of quality. A soft real-time system can tolerate large jitter [59, p. 14] and even deadline misses. Deadline misses in a soft real-time systems will only have an effect on the quality of the output i.e. the shutter on a digital camera, if it is slower than it should it will affect the picture but it will not have catastrophic consequences. However, soft real-time systems are a sub category embedded systems and there are embedded systems which does not implement the timeliness and deadline principles at all.

A hard real-time system puts more importance on timeliness. They can tolerate less jitter than a soft real-time system. Delays or deadline misses might have catastrophic consequences, i.e. an air-bag in a car. If that system doesn’t deliver in a timely manner it might be the difference between life and death of a passenger. In other words a hard real-time system is stricter when it comes to time and scheduling to ensure all the tasks in the system must be finished in a timely manner.

3.3.2 Essential properties of real-time languages

As discussed above in Section 3.3.1, the features of real-time systems determine some properties for programming languages to make them suitable to be used in real-time systems. Four essential properties are explained below.

3.3.2.1 Execution time measurability  The definition of the correctness of the outcome from a real-time system is not only about the result itself but also about timeliness of a task’s completion. That is to say, for the languages used in real-time systems, compared with normal embedded systems, execution time analysis is an important factor. Languages that incorporate the imperative paradigm, explained in Section 3.2.1, is advantageous for execution time analysis due to the simplicity of the constructs in combination with the code being relatively close to the resulting machine code in imperative languages. This contributes to imperative languages popularity in real-time applications.

3.3.2.2 Determinism  For all the software applications, stability is the most important and basic requirement. Stability in the context of software is that the same set of inputs will generate the same output every time. State is the main issue that affects stability, as explained in Section 3.1. Accidental state change
can affect the final outcome of a system in an undesirable manner. Therefore, it is desirable that programming languages used in real-time systems decrease the negative effects of poor state management in a system. Since state management is restricted in functional languages this might make functional languages suitable for the real-time domain.

3.3.2.3 Memory management Since the memory resources in real-time systems are restricted, a sound support for memory management is a good property for the programming languages of real-time systems. As a fact, more and more high-level programming languages nowadays, i.e. Java or Haskell, support automatic memory management in the form of garbage collection [53]. Two criteria for evaluating the performance of language provided memory management tools are memory usage and the execution time of the garbage collector. A decent memory managed language should have a resource usage that is close to the actual need of the system and a fast garbage collector. The problem with memory managed languages in the real-time domain is the unpredictability of the garbage collector. This can be solved by disabling the automatic garbage collector and have a specific task in the system for deallocation of unused memory. With this strategy one can have garbage collection with the ability to do schedulability analysis on the system.

3.3.2.4 Concurrency Is the ability to do computations in parallel. It can either be true parallelism where different calculations are executed on different cores, or pseudo parallelism where all calculations are made on one core but the switching frequency of calculations on that core are so fast that from a human perspective it looks like it is doing several tasks at once. For the system to be able to achieve pseudo parallelism one must incorporate some kind of scheduling algorithm like Round Robin [75, p. 222]. As a real-time system programming language, one must support the communication between independent tasks. As said by Kevin Hammond [47]:

"the language must allow the construction of systems as communicating units of independent computation."

In conclusion, the requirements for languages suitable for real-time systems are somewhat in conflict. Garbage collection is terrific since it simplifies memory management for programmers and decreases the usage of memory as well, but it will have a negative consequence to the execution time- and schedulability analysis of the system. Functional languages have restrictions on accidental state mutation which will increase the stability of the system but they are, memory managed and they are some times compiled down to an intermediate language, i.e. Clojure is run on the JVM and F# is compiled down to CLR, therefor the efficiency of the translation to the intermediate language also is a topic for major concern.
3.3.3 Common practice paradigm in real-time systems

C is according to langpop.com [77] the most common programming language in today’s software and in the wikibook Embedded Systems/C Programming [90] they say that C is probably the most popular language for programming embedded systems. Therefore one can deduce that the most commonly used paradigm is the imperative paradigm. If we look towards the IEC 61131-3 standard for Programmable Logic Controller (PLC) we can see that out of those five languages we have three graphical ones and two imperative ones, Instruction List (IL) and Structured Text (ST). This is also an indication that the imperative paradigm is the most commonly used paradigm in the industry of hard embedded real-time systems.

But why is that so? The object oriented paradigm relieved some of the problems when it came to state in the imperative paradigm. The functional paradigm helped with state in the form of referential transparency. But why is not this being widely used in the real-time domain? These are questions the authors will try to answer in this thesis.

The benefits of the imperative paradigm is its simplicity. When it comes to setting timing constraints on imperative code one does not have to worry about the effects of a garbage collector, or the translation into an intermediate language or any of the other fancy tools that exist in languages higher up in the power continuum, explained in Section 3.2.3.2.

3.3.3.1 What makes imperative code suitable

If we look towards the essential properties of a language suitable for real-time applications.

In the imperative paradigm it is easier to calculate the execution time of a program with static analysis compared to other paradigms since there are no fancy tools, i.e. garbage collection, available to the developer.

When it comes to stability it seems like the imperative paradigm is not suitable due to the consequences that state imposes on software written in the paradigm in question. No restrictions are put upon how state can be mutated by programmers as explained in Section 3.2.1, and therefore it might point towards the unsuitability for the imperative paradigm in systems with a high demand on stability and reliability.

Since imperative languages generally do not implement automatic memory management but leave the hassle of allocating and freeing memory to the programmer, it demands a good development team to be able to sort this out. This is also a performance vs resource usage question. Do I want a garbage collector and save resources at the cost of performance or do I prefer control.

Imperative languages do not enforce any restrictions on the management of state and suffer greatly when it comes to accidental state mutation as explained in Section 3.2.1. Since concurrent systems are getting more and more common one could argue that the imperative paradigm is not suitable for those type of applications due to effects of state as a product of timing.

In conclusion one could argue that the imperative paradigm is not suitable for real-time systems since it is only fulfilling one out of four essential properties.
3.3.4 The functional language paradigm in Real-time Systems

It is commonly believed that the functional paradigm has not generated much interest amongst the real-time systems community even though some functional languages have shown prevailing advantages in academia [67, p. 9-11]. One reason might be that, like the other high-level languages, languages in the functional language family are often designed to put layers of abstraction above the underlying hardware and restrict direct control over said hardware [36]. However, recent decades have witnessed the blossom of the functional language family. Some dialects that target real-time embedded systems such as Hume [49] and Nitro [36] have emerged in academia.

This section focuses on the limitations and challenges of the functional paradigm in embedded real-time systems. This section is split into two parts, the first part explains the challenges for functional languages in the real-time domain and the second part gives a brief explanation of functional languages aimed at real-time applications.

Including the challenges mentioned in Section 3.2.3, the barriers that hinder the success of most functional languages in the real-time domain can be split into two categories: unreasonable barriers and reasonable barriers [86].

3.3.4.1 The unreasonable barriers Functional programming has been disputed for decades. A lot of reasons put forward by opponents of the functional paradigm aimed at real-time applications has to do with the performance of languages in said paradigm. The acceptance of that argument, and therefore denying functional languages in the real-time domain because of its claimed performance, is asinine. It might have been true a decade ago but it is doubtful nowadays due to the advances in processor performance and memory capabilities of target systems, this can be seen in Figure 4 and Figure 3.

Like we mentioned in Section 3.2.3.2, imperative or object oriented programmers have a hard time rationally evaluating languages that are not their forte. Recursion, that functional languages rely heavily on, is according to J. Altiere [25] something that developers fear. One of the reason for that is they are already used to the iterative structure of the loop statements in imperative or object oriented languages. Another reason might be the increased usage of stack memory in recursive calls [62, p. 511-521].

With a lot of functional dialects emerging and improvements to profiling systems for functional languages, performance of functional programming languages has improved vastly. The code-measure-improve cycle has been applied in functional languages to boost the performance in time and space consumption [81]. At the same time, tail call optimization is standard for some functional languages such as Scheme, which allows the tail recursion to be performed without increased stack usage, thus improving the space and execution efficiency significantly [96, p. 5-6]. In fact, as it is indicated in Pseudoknot benchmark [39] some functional languages in some test cases achieved comparable or even better performance compared to C++ [27] which is becoming one of the mainstream languages of real-time systems.
Performance is rare to be the essential property for the success of a programming language. For example, Java has become extraordinarily popular and successful in the development community despite its significantly poorer performance compared with C. On the contrary, assembly which has high performance is not widely used because of its poor understandability as explained in Section 3.2.1. Functional programming languages emphasize properties like stability and reliability which is very important to real-time systems.

If we also look towards development time, languages of higher power and abstraction, as explained in Section 3.2.3, tend to increase productivity of each developer. People generally do not write huge systems in pure assembly since it would take a lot of time and money to complete the project. Functional languages will decrease the development time due to more abstractions so each developer does not have to worry about i.e. memory management.

In conclusion, performance is not a good reason for denying functional languages in the real-time domain. Some functional languages, if done right, can even have better performance than C++ [27].

3.3.4.2 The real barriers There are some undeniable factors that hinder the adoption of functional languages in the development of real-time system. Below, three of them are explained.
1. Compatibility

2. Tools

3. Supports

As P. Wadler said [86, p 10]:

"Computing has matured to the point where systems are often assembled from components rather than built from scratch."

As an effect, the languages that is used in building or compatible with most software components will gain a competitive advantage. So far, most real-time system components, such as hardware interfaces, are written in C or C++ [86], this makes it essential for a language to have a way of interfacing these libraries. Functional languages have significant differences compared to languages in the imperative or object oriented programming paradigm. Overcoming the isolation of functional languages is an urgent task for functional programmers to make it compatible with the already existing libraries.

The tools available for functional languages are far from enough, compared to C or C++. For developing a real-time systems, a sound debugger and profiler are compulsory [86]. Generally speaking, development environments often have integrated debuggers and profilers which facilitates the development process. However, functional languages do not have enough tools that can evaluate time and space usage which is critically needed for a language to be successful in the real-time domain.

Support for functional languages from the commercial software community has been underwhelming. When you search for software related work most of the results are for positions with an emphasis on C, C++ or Java. The profit-oriented commercial companies are hesitant to dip their toes into the water of functional languages, since the first experiences tend to be expensive. However, Ericsson’s Erlang[26] is growing up to be an industrial-grade language with a solid user environment. Without a doubt, in a decade, a lot of successful functional systems and corresponding tools will emerge. This will surely facilitate some attractive functional languages.

3.3.4.3 Why should we care?

According to Moore's Law [69]:

"the number of transistors on integrated circuits doubles approximately every two years."

and Intel states that [83]:

"We will no longer see significant increases in the clock speed of processors. The power consumed by the fastest possible processors generates too much heat to dissipate effectively in known technologies. Instead processor manufacturers are adding multiple processors cores to each chip."
As we can see in figure 4 the trend of transistors per integrated circuit is still following Moore’s Law. But the majority of the processors after the Itanium 2 are multi-core processors. So with this information one could deduce that the reason for the fulfillment of Moore’s Law is more and more cores on each chip instead of more and more transistors in one core. This puts more emphasis on the problems concurrent systems have to deal with since more and more cores are incorporated in the hardware of high end systems.

It is obvious that parallel computing is the future of software engineering. one of the great advantages of functional languages is that operations are entirely independent, hence perfect for parallel implementations [84]. It is reasonable to deduce that in the near future, multi-core processors will become the mainstream micro-controller in embedded systems. Therefore the stability and easy-to-parallel properties of functional languages will be appreciated by developers.

3.3.5 Functional languages for the Real-time domain

Even though there are difficulties for the functional paradigm in the embedded real-time domain there are some efforts in academia to overcome these difficulties. In this paper four are presented:

1. HUME
2. SAFE
3. Nitro
4. Timber
5. Erlang

3.3.5.1 Hume Higher-order Unified Meta-Environment is a strongly typed, mostly functional language with an integrated tool set for developing, proving and assessing concurrent, safety-critical systems. The Hume language was developed by Kevin Hammond at the School of Computer Science, University of St Andrews, Greg Michaelson and Robert Pointon at the School of Mathematical and Computer Sciences, Heriot-Watt University.

The goal of the Hume language is an expressive language with strong guarantees on execution time and space usages. This is achieved through a careful language design based upon Leveson’s guidelines for software intended for safety critical applications [58].

In general, safety critical systems must meet both strong correctness criteria and strict performance criteria. The latter are most easily attained by working at a low level, whereas the former are most easily attained by working at a high level. Hume is therefore designed as a three layered language. The outermost layer is a static declaration language that provides definitions of types, streams,
Figure 4: Transistor Count and Moore’s Law - 2011 [91]
etc. The innermost layer is a conventional expression language using pure functional expressions. The middle layer is a coordination layer that links functions into processes.

The functional expression language in Hume is intended for the description of single one-shot non-reentrant processes. It is deterministic and has statically bounded time and space behavior. Expressions that are not the target of dynamic timeouts are restricted to statically checkable primitive recursive forms to provide the ability of static analysis. The functional part of the Hume language has no imperative state, it is encapsulated in the coordination language within Hume.

The coordination language is a finite state language for the description of multiple interacting reentrant processes built of the expression layer. It has been designed to incorporate safety properties as the absence of deadlocks, livelocks and resource starvation. The coordination language is responsible for the interaction with imperative state such as ports and streams to external devices.

In summation, Hume is a partly functional language built with security, correctness and concurrency in mind. It provides a sound border between the functional expression language layer and imperative state encapsulated in the coordination layer within. [48, 49]

3.3.5.2 Safe

Safe is a first-order functional language, with a different approach to memory management compared with other functional languages. Safe is developed as a research platform for optimizing, certifying and analyzing functional languages in the terms of memory usage by following a Proof Carrying Code approach [42], which is a critical property for programming in embedded systems with limited memory resources. That is to say, Safe is a step towards functional implementations in embedded systems.

Generally most functional languages use garbage collection to manage memory resources. This method relieves the programmer from the hassle of low-level memory management during development. However, on the other hand, garbage collectors incur difficulties in the evaluation of memory usage and unpredictability when it comes to execution time. Safe’s memory model is based upon heap regions, and therefore negating the necessity of a garbage collector for memory management. By introducing four rules [68, p.5-6] for memory allocation and deallocation in functions, the compiler distributes or removes data fields within certain memory regions and cells on the heap in a more wise and predictable manner. This improves the memory recycling mechanism and therefore reduces the requirement of memory. In addition, by introducing type reference algorithms [68, p.6-17] systems developed in Safe guarantees the absence of dangling pointers and memory bounds when done correctly.

In summation, the commonly believed drawback that functional languages are bad and unstable in performance are conquered or at least conquerable and the memory management method used in Safe is a good example.
3.3.5.3 Nitro  Nitro is a low level functional language that gives the developer access to the underlying hardware. It combines the convenience and safety properties of the functional language paradigm and the hardware interaction of lower level languages.

Nitro was introduced in the dissertation of A. Clark [36] at the Laboratory for Foundations of Computer Science, University of Edinburgh. Since Nitro is a functional language, systems written in Nitro will gain the advantages mentioned in Section 3.2.3.

There are two main features of Nitro that makes it worth mentioning in the domain of real-time systems.

1. low-level
2. Interfaces to foreign language data

The low-level aspect of Nitro empowers the developer with the ability to inspect and control the underlying hardware of the system. This enables Nitro, as a low-level language, to be used in development of operating systems, hardware drivers and run times.

As discussed in Section 3.3.4, one of the barriers that blocks functional languages from being prevalent in the real-time domain is compatibility. Nitro is designed with interfaces to data from foreign languages, this enables Nitro to interface with libraries or other existing coder resources written in other languages without translation [37].

In conclusion, Nitro shows the right direction for relieving some of the barriers for functional languages in real-time systems. It also dictates that more and more research efforts are dedicated to facilitate and improve the implementations of functional languages in the real-time domain.

3.3.5.4 Timber  Timber is a descendant of O’Haskell which is designed as a reactive object-oriented concurrent functional language. As fact, Timber is a purely functional programming language and as A P. Black [30] says:

"The 'naked' view of Timber is a language that consists of only the Haskell kernel and a set of primitive monadic constants."

By using monads, Timber makes it possible for a purely functional language to have stateful objects. In addition, by integrating reactive objects, polymorphic subtyping and usual imperative commands, Timber can be characterized as an imperative object-oriented programming language. At last, Timber supports high-order functions, recursive definitions, and monads, which is in line with the functional paradigm and therefore can be counted as a purely functional language. As said by A P. Black [30]:

"Indeed, Timber attempts to combine the best features of three different programming paradigms."
The reason why Timber is suitable for embedded real-time system development is that Timber adopts a creative model to construct a real-time system. This model is to build up a referential transparent system by objects and reactions, which are endowed with platform-independent timing properties. Therefore, the real-time reactive objects or actions can be configured within a timing constraint such as a deadline and a period in real-time systems.

Reactive object [72, p. 5-24], is one of the main features added to Timber. According to the thesis of Nordlander [72], reactive objects is actually a monadic implementation of stateful objects, and it is intended as a replacement for Haskell’s standard IO model. The reactive objects in the setting of the functional paradigm can provide concurrent objects and assignable state variables. That is to say, in a view of embedded systems development, it offers chances for concurrent events with timing constraints while still honoring the principles of referential transparency. Therefore, to build up a real-time functional system, one only need to build the system with real-time dynamic behaviors or events which are the composition of time sensitive, concurrent executable reactive objects [30].

3.3.5.5 Erlang

In 1986 Joe Armstrong developed the first version of Erlang at Ericsson [93]. A proprietary language that was later released as open source in 1998. Erlang is a general purpose functional language influenced by Prolog and Smalltalk, and a language that supports soft-real-time non-stop applications where hot swapping of code base is possible.

The main strength of Erlang is its support for concurrency which is based on the Actor model [23]. In the actor model is the actor considered the primitive for concurrent computation. In Erlang, an actor is called a process. Actors react to messages and when a message is received by an actor it can be concurrently executed:

1. Make local decisions
2. Send a finite number of messages to other actors
3. Create a finite number of new actors
4. Designate the behavior to be used for the next message it receives

This concurrency model is a good fit for real-time applications and one that comes to mind is the control of concurrent physical processes. One downside though is that Erlang is only geared towards soft real-time systems and there is nothing within the language that helps with hard real-time constraints.

For Erlang to be proficient in the hard real-time domain V. Nicosia mentions two pieces of the puzzle that have to be solved. They are Real time message passing, which will be introduced in a future version of Erlang, and unpredictable behavior of garbage collector [71].
4 Core features of functional languages

In this section a brief introduction to some of the core features of functional languages is given and how to perform these in the imperative language C. All examples of functional code are given in the impure functional language Clojure [41, p. 1-3] or in the pure functional language Haskell [61].

The examples highlights the concept of the function and disregards other aspects such as memory management. Those services are provided by underlying libraries which can be seen in Appendix B. The datastructures used in the examples can be seen in Appendix A.

4.1 State and Side Effects

In a pure functional language functions have no side effects, this is how functional languages achieve referential transparency. However, in an industrial system which is using sensors and actuators to interpret and alter the world around it one needs side effects to mutate the state of the system to be able to use the actuators. How to deal with state mutation, which is heavily used and easy to implement in an imperative system, is one of the challenges when changing paradigm to a functional one.

To relieve complexity caused by mutable state, most of functional languages adopt the following two techniques:

1. Immutable Data
2. Monads

4.1.1 Immutable Data

Immutable data is data that can not be altered after it has been created [88, p. 8]. This concept is deeply rooted in functional programming and often enforced by the language itself. This is the reason to why functional languages rely heavily on recursion since the only time one can alter data is to change the arguments to the next call of the same function. This can be seen in figure 5, in this example items are prepended to the beginning of a list. The cons function returns a new list with the value of x appended to the list y which is then passed into the same function again to allow for more additions before the final list is returned out of the first call to the function. The benefit of immutable data is that since you are always working with copies you do not have to worry about what your changes impose on the data available to other parts of the system since they only have identical data not the same. Your changes will not affect other threads or processes since the data is immutable. The downside is that immutable data demands a bigger chunk of memory to facilitate all copies.

Since immutable data relies heavily on copies a good library to manage memory to allocate space for all the copies and a garbage collector is needed and a basic version of a garbage collector has been implemented for this thesis.
((fn create-list [x y] (if (zero? x) y (create-list (- x 1) (cons x y)))) 5 '())
⇒ (1 2 3 4 5)

Figure 5: Clojure example of recursive creation of a list

An explanation of the garbage collector written for this thesis can be seen in Appendix B.

4.1.2 Monads

A monad is a computation builder that creates a first order value that represents a stepwise computation. Monads are heavily used in pure functional languages such as Haskell [61, p. 267]. The benefit of monads is that one can represent I/O operations as pure values and through that keep it referentially transparent. There are a couple of monads and in this section the I/O monad and the maybe monad are explained.

Since a monad is a representation of a stepwise computation one can compare it to a recipe. A monad itself performs no computations it only represents how a computation is made. This allows for combination modification and manipulation of monads without affecting the system in any way and therefore keep it referentially transparent.

A monad itself does not perform anything. The computation built inside of the monad is not applied to the system until a monad evaluation function is called with the monad as an argument, such as `domonad` in Clojure or `do` in Haskell. It is during the evaluation of the monad side effects occur and the state of the system is altered.

The maybe monad [61, p. 269] is a monad where each step in the sequence of actions is performed only if the preceding step did not return nil. If one of the steps in the sequence gets a nil value during execution the whole monad stops and returns nil as an output. An example of this can be seen in figure 6. In this example we can see two usages of the maybe-m monad in Clojure. In the first example both placeholders `a` and `b` are given an integer value and therefore the computation (`*` a b) is performed and 12 is returned as a result from the monad. In the second example `a` is given nil as a value and therefore the execution stops there and the whole monad returns nil. If the monad would not stop its execution at `a` one would get a null pointer exception in the multiplication. The benefit of the maybe monad is that it allows execution to stop when something goes wrong to avoid lengthy computations or unnecessary errors.

The maybe monad is not of great importance for this thesis but it is an example on how monads can be used. The I/O monad [61, p. 298] on the other hand is of great importance. Haskell is a pure functional language and it relies
(\texttt{domonad} maybe\textendash m [ a 3 \\
  b 4 ] \\
 (\ast{} a \ b))
\Rightarrow 12

(\texttt{domonad} maybe\textendash m [ a \ \texttt{nil} \\
  b 4 ] \\
 (\ast{} a \ b))
\Rightarrow \texttt{nil}

Figure 6: Clojure example of the maybe monad

\texttt{ioActions} :: [\texttt{IO ()}]
\texttt{ioActions} = [(\texttt{print \ "Hello!" )}, \\
 (\texttt{putStr \ "just kidding" )}, \\
 (\texttt{getChar >> return ())}]

Figure 7: Haskell example of the I/O monad

heavily on the usage of monads to perform all of its I/O operations. The I/O model in Haskell is the inspiration for this thesis. The I/O monad is a way of representing I/O as a pure value. Its a first class value that represents a sequence I/O operations such as putStr or getChar. An example of the I/O monad can be seen in figure 7 this example is given in Haskell. In this example we have represented the sequence: Print "Hello!" to standard out, then put the string "just kidding" to standard out, then wait for a character input and after that return. This monad does not perform these actions until it is executed with for instance the \texttt{do} function. This can be seen in figure 8.

An example of how the maybe monad could be implemented in C is shown in Appendix C and an example of how the I/O monad could be implemented in C is shown in Appendix D.

4.2 Programs as data: Understanding the function pointer

The function pointer in the C language enables the programmer to pass a behavior to functions and through that effect the outcome of functions during runtime [85, p. 113]. In figure 9 is an example of the type signature of a function pointer in C. In this example we have a function pointer called \texttt{foo} which

\texttt{main = do head ioActions}

Figure 8: Haskell example of the execution of the I/O monad defined in figure 7
int ExampleFunction (int a, char b) {
    put (c);
    return a;
}

int (*foo)(int, char) = ExampleFunction;

Figure 9: Example of a function pointer

void qsort (void *base,
    size_t nitems,
    size_t size,
    int (*compar)(const void *, const void *))

Figure 10: Signature of the qsort function

can hold the address to a function that takes an integer and a character as arguments and returns an integer as an output. This function pointer is assigned the address to the function ExampleFunction so the statement ExampleFunction(1,'b') is equivalent to foo(1,'b').

This is already used in a lot of standard libraries for the C language. I.e the qsort function [11] that is defined in stdlib.h that implements the quick sort algorithm, as seen in figure 10. The last argument in that function’s signature is a function pointer to a compare function. This gives the programmer the ability to compare any data type as long as he/she defines how to compare the data within the passed compare function.

With the usage of function pointers one can, in the C language, implement some of the necessary concepts of functional languages such as higher order functions explained in Section 4.3.

4.3 Higher order functions

A higher order function is a function that either takes a function as an argument or returns a function as an output, or both. In that sense any functions that takes a function pointer as argument or returns a function pointer are higher order functions [50, p. 78-79].

Common higher order functions in functional languages include [88, p. 13]:

1. Cons
2. Map
(defn cons [x y]
  #(cond
    (= % 0) x
    (= % 1) y
    :else (print 'Error: argument outside range')))

(defn car [z] (z 0))

(defn cdr [z] (z 1))

(car (cons 1 2))
⇒ 1

cdr (cons 1 2))
⇒ 2

Figure 11: Example of how Cons, Car and Cdr can be built in Clojure

3. Filter
4. Fold

These three functions and an example of the same function implemented in C are explained in succeeding subsections.

4.3.1 Cons, Car and Cdr

One way of creating lists in functional languages is with the help of the cons function [24]. The cons function takes two arguments and returns a list where these two arguments are the items of the list in question.

Cons is tightly coupled with the two functions car and cdr. Car takes a list produced by cons and returns the first element of the list in question. Cdr also takes a list produced by cons and returns everything except the first element in the list.

An example of how the cons function can be built in Clojure can be seen in Figure 11. This example shows one way of building lists with the help of functional pointers. A function pointer is constructed where it will evaluate either to the first or the second argument of the cons function. This evaluation is based upon an argument passed to the function pointer. The car and cdr functions are then built by only passing the correct argument to the function pointer returned by cons. This is an easy way of implementing lists in functional languages.

In C on the other hand one can not really utilize function pointers in the same way as in functional languages. Functional languages does not differentiate between function pointers and values but C does. This forces one to build the
list* cons(void* Item, size_t size, list* listHeader)
{
    if(listHeader == NULL)
        return newNode(Item, size);

    return prepend(lst_newNode(Item, size), listHeader);
}

list* car(list* listHeader)
{
    return copy(listHeader);
}

list* cdr(list* listHeader)
{
    if(listHeader == NULL || listHeader->next == NULL)
        return NULL;

    return copyList(listHeader->next);
}

Figure 12: Example of Cons, Car and Cdr in C

cons function in another way. An example of this can be seen in Figure 12. Here we can see that the cons function prepends a new list node to a passed list pointer and the car and cdr will either copy the first item and return that or copy the rest of the list and return that.

4.3.2 Map

Map [50, p. 83-84] is a classic higher order function. It is a transformation function that applies a supplied function to all items in a collection. It is often called apply-to-all in the functional form [57]. In figure 13 two examples of how the map function can be used are shown. In the first example a function, double which doubles a value, is supplied with a list of integers. The output from that call is a new list with all items in the original list multiplied by two.

In the second example a function called validate-solution and a list of proposed solutions are supplied. The output from this example is a list of messages that indicates if the given solution is valid for an arbitrary problem.

An implementation of the map function in C can be seen in figure 14. In this example a pointer to a list, l, and a function, fn, is passed. The desired outcome is a new list with the function, fn, applied to all items in the list, l. To keep the data immutable a new list pointer, o, is created and new items are appended to that list after the item in question has been evaluated by l. When the function fn has been applied to all items in l we return o. This will supply
\[
\text{map}(\text{fn} \ [x] \ (\ast \ x \ 2)) \ ' (1 \ 2 \ 3 \ 4 \ 5)) \\
\Rightarrow (2 \ 4 \ 6 \ 8 \ 10)
\]

\[
\text{map} \ \text{validate-solution} \ ' (\text{solution1} \ \text{solution2} \ \text{solution3})) \\
\Rightarrow ("\text{ok}" \ "\text{ok}" \ "\text{not a solution}")
\]

Figure 13: Clojure example of the map function

\[
\text{list} * \text{map}(\text{list} \ *1, \ \text{void} *(*\text{fn})(\text{void} *, \ \text{void} *), \ \text{void} *\text{args}) \\
\{
\text{list} *\text{o} = \text{NULL} ; \\
\text{list} *\text{curr} ; \\

\text{for} (\text{curr} = 1 ; \text{curr} != \text{NULL} ; \text{curr} = \text{curr}\rightarrow\text{next}) \\
\{
\text{o} = \text{append}(\text{o}, (*\text{fn})(\text{curr}\rightarrow\text{val}, \ \text{args})) ;
\}

\text{return} \ \text{o} ;
\}
\]

Figure 14: Example of the map function in C

the needed behavior for the map function.

Since C is a statically typed language [92], a language where the types of
everything have to be known by the compiler at compile time, this solution
requires that type casting before the logic in \text{fn} is done within \text{fn}. This solution
also relies on that memory management is preformed within either \text{fn} or within
the \text{append} function.

4.3.3 Filter

Filter [50, p. 83-84] is exactly what it sounds like, a filter, a net that only
captures items that fulfill a certain criteria. An evaluator function, with a
boolean result, evaluates each item in a collection and creates a new collection
with only the items for which the evaluator function returns true.

In figure 15, two examples of the usage of the filter function in Clojure are
shown. In the first example we filter out all the items that are even in the the
passed list and in the second example we filter out all items that is not equal to
true.

In figure 16 a filter function in the C language is shown. In this solution only
the items that evaluates to true by the \text{fn} function are appended to the output
\text{list} \ \text{o}. This solution relies on that type casting is done within the passed func-
tion, \text{fn}, and that memory management is done within the \text{copyitem} function.

43
(filter even? '(1 2 3 4 5 8 2))
⇒ (2 4 8 2)
(filter (fn [x] (not (true? x))) '(true false 1 2 true "foo"))
⇒ (false 1 2 "foo")

Figure 15: Clojure example of the filter function

(list * filter (list *1, bool (*fn)(void *, void *), void *args) {
  list *o = NULL;
  list *curr;
  for (curr = l; curr != NULL; curr = curr−>next) {
    if (((*fn)(curr−>val, args)) {
      o = append(o, copyitem(curr−>val));
    }
  }
  return o;
}

Figure 16: Example of the filter function in C

4.3.4 Fold

Folding [50, p. 84-86] is the act of taking a list and combining all the values in that list with an accumulation function. It generally takes a function, a start value, which is often the identity value of the function i.e. 0 for addition 1 for multiplication and so on, and a sequence of elements to apply the accumulation function upon. Fold is also called an aggregate function.

There are two version of fold called foldl and foldr. The difference is whether it is left or right associative. Foldl is left associative and foldr is right associative as indicated by the last letter in the function name.

In figure 17 we have two examples of a fold. One for foldl, and one for foldr. The example is given in the pure functional language Haskell. In these examples can we see what a fold does. First we create a list containing the two strings "1" and "2" these two are then passed with the aggregator "0" into a concatenation function where they are successively added into a string with the pattern ( Current-accumulator + new-value ) until all of the items have been accumulated.

In figure 18, an example on how the foldl function could be implemented in the C language. This solution requires that all memory management for the accumulator, o, is done within the passed aggregate function fn.
\texttt{foldl \((\times \times \rightarrow \text{concat} \ ["\times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \time
5 Case Study

Packsize® [10] is the world leader in on demand packaging®. It is a company that builds and distributes packaging machines with corresponding software that are specialized for lean manufacturing.

In this case study the authors redesigned the software implementation of their newest machine the iQ Fusion [9]. The iQ Fusion has been designed for the fulfillment industry and is currently under development. The authors have with this machine investigated whether an implementation of the control unit is possible in the functional language paradigm. If so, this would yield these benefits:

1. Simpler reasoning about the system
2. Simpler testing of the system

The authors tried to find what trade-offs one has to take into account when choosing a development paradigm when designing a real-time embedded system in the industrial domain.

5.1 Goals

Besides the questions posed in Section 2, the questions answered in this case study included:

1. Would a functional implementation allow for thorough unit testing that would increase maintainability and stability of the system?
2. Will the functional implementation prove to be better than the current implementation, with regard to readability, stability, reliability and reasonability
3. How will the code metrics on both implementations differ?

5.2 Expected outcome

The expected outcome of this study was to answer the questions posed in Section 5.1 and with these answers answer the questions in Section 2. In an ideal situation the product of this case study would be a fully functional implementation of the iQ Fusion control unit using the functional language paradigm. If this would not be possible an investigate if its possible to use a combination of paradigms including the functional paradigm was performed.

The ultimate goal of the study was to get a good comparison between the current system and the redeveloped functional version of the system.
5.3 Current system

5.3.1 Software

5.3.1.1 Platform  The current system is running on a hard RTOS called Automation Runtime™ [2]. This RTOS is used for all devices supplied by Bernecker & Rainer (B&R). Some properties of Automation runtime™ include:

- Configurable jitter tolerance for all tasks
- A cyclic system that guarantees deterministic behavior
- Guaranteed highest possible performance for the hardware being used
- Task cycles are synchronized with I/O cycles
- Identical behavior for the entire B&R I/O system
- Services like FTP or Web services can easily be enabled and used
- Deterministic cycles with minimum jitter

5.3.1.2 Memory model  The memory for tasks within Automation runtime are split in two parts, one stack and one heap.

The sizes of these two are set before runtime to a maximum value. One stack is shared between all tasks in one task category. Task categories are defined by period so all tasks with period 1 millisecond (ms) will share the stack for that task category. The stack for each task class is separate so they do not impose on each other. One thing to note is that since all tasks in a category have the same priority the stack will essentially be empty when the next task in the current period iteration is scheduled for execution. If a task would use too much stack memory exceptions will be thrown and execution of the system will stop.

The heap on the other hand is defined for each task with a certain variable in the .hpp file for the corresponding cyclic .cpp file. This will give the task in question a heap with the set size. This is then used for all allocations by the task in question and if too much heap memory is allocated exceptions will be thrown and execution of the system will stop. The total amount of memory available for the system, as mentioned below, is the full 256 mb SDRAM unit.

5.3.1.3 System  The current software system consists of nine user defined tasks:

- MainCyclic  Holds the state machine for the whole system.
- KnifeCyclic  Activates and Deactivates the knife.
- PCCommunication  Upholds the connection loop to the overlying PC.
- IOController  Handles HMI devices such as buttons.
• **GetStruct** Http service that handles reading of internal variables of the system.

• **SetStruct** Http service that handles writing to internal variables of the system.

• **HttpClientCyclic** Http service that handles trigger based messages.

• **ZebraPrinterController** Handles communication to Zebra’s [22] industrial printers connected to the system.

• **Logger** A logging task that allows for system logging.

The majority of the tasks are written in C++ with an object oriented approach except for the Http services which are written in structured text with an imperative approach. All tasks are scheduled periodically with periods ranging from 1 ms to 200 ms. The tasks of the system are scheduled with a priority based preemptive scheduler. The priority of the tasks is based upon which task class the task is situated in. This is predefined by Automation Runtime™, an example of this can be seen in Figure 19.

During the implementation phase of this thesis the authors started with the smallest task, which was KnifeCyclic, to show the basics of functional programming in an imperative system and then continued with tasks prioritized by code volume where low code volume corresponded to high priority. The focus lay on creating an as pure as possible version of each task. The reason for this implementing priority mechanism that starting with the smallest task and continuing upwards based on code volume was to give us an easy way to manage time scope, and implementation was planned to continue with new tasks until there was not enough time for the next task.

5.3.2 Hardware

The current system is built upon the X20CP1584 [18] PLC module from B&R Automation [5]. The specification for this unit consists of:

• Intel® Atom™ 600 MHz with an additional I/O processor

• Ethernet, POWERLINK [94] with poll response chaining and onboard USB

• 1 slot for modular interface expansion

• Compact Flash as removable application memory

• 256 MB DDR2 SDRAM

• Fan-free
Figure 19: Automation runtime scheduling example
This unit is then paired with one X20DI9371 [19] module for digital input, one X20DO9322 [20] module for digital output and one X20BT9100 [17] bus transmitter for transmission over the X2X [21] bus.

The actuators are pneumatic and controlled by the Digital Out (DO) block. All sensors of the system are digital and connected to the Digital In (DI) block.
6 Implementation

6.1 Introduction

In the following sections the implementation of this thesis is explained. The first section provides the rules that were used during the implementation phase of this thesis to enforce the functional programming paradigm. Section 6.3 with its subsections explains the underlying services and facilities which the tasks are built upon, i.e. the garbage collector. For implemented tasks see Sections 6.4.2, Section 6.4.3 and Section 6.4.4. There are only three out of nine tasks implemented during this thesis. The decision to implement three out of nine was they way used to manage the scope during this thesis.

6.2 Implementation Rules

In this section are the rules of which the implementation of this thesis has been restricted by presented. These are the rules that the system will base its evaluation on and they will enforce the functional paradigm even though an imperative language is used.

1. All I/O operations has to be performed within a monad
2. All iterative processes have to be performed with the help of recursion
3. All data has to be immutable

6.3 Services and Facilities

6.3.1 Unit testing

One of the questions in the case study as posed in Section 5.1 is:

Would a functional implementation allow for thorough unit testing that would increase maintainability and stability of the system?

This question requires that the development of the system in question is done with the TDD model [28]. The development cycle of the TDD model consist of five stages:

1. Add a new test for what is to be developed
2. Run the tests to verify that the new test fails
3. Implement the new function
4. Run the tests to verify that the new test passes
5. Re-factor the code
This cycle is then repeated until the whole system is built. The important part is that the test is written before any of the development is done. This is due to the fact that if the behavior of a function is defined you can write a test for it. If the behavior is not defined one needs to go back to the drawing board and think through the task at hand and what each part should do. This is in line with the test-first principle of extreme programming [29]. This allows for a continuously evolving test suite that always cover the majority of the code base.

To be able to use the concept of test driven development one needs a testing library. The library used in this thesis is CuTest [7]. CuTest is a small lightweight testing library for the C language. It consists of a single .c and .h file and works with all major compilers on Windows, Linux, Unix and PalmOS.

6.3.2 Immutable Data

Keeping the data immutable is one of the fundamental principles of the functional programming paradigm. In this thesis data is kept immutable by deeply copying data resources every time some kind of data mutation is needed. The mutations are then performed on the copy which is returned to the caller instead of the original data. This yields immutable data since one can see the function mutating the data as its constructor.

If we look towards the list library implemented in this thesis as an example, all operations possible with this library will return deep copies of the relevant data. An example of this can be seen in Figure 20. In this example a function that returns the \( n \)th element in a list is shown. This function successively skips the first item of the list until it has reached the needed element. In each recursive call a copy of the list with the first item is omitted is passed into the next call. When the element in question is reached a copy of that element is created which is returned back to the caller.

The downside of this is it puts more strain on memory management and therefore a garbage collector has been developed in this thesis. An explanation of the garbage collector can be seen in Section 6.3.3.

6.3.3 Garbage collector

Due to the heavy use of copies to keep data immutable, and the fact that each copy will occupy space in heap memory, a sound memory management strategy is required. The programming language used in this thesis is C and therefore one does not have automatic memory management which can be found in higher level languages such as lisp. At the same time, in the target system, which is a hard RTOS, unbounded garbage collection can not be tolerated due to the nature of this category of systems. Therefore a small and simple garbage collector has been implemented.

The garbage collector in this thesis follows the theory of a mark-and-sweep garbage collector [76, 4]. The mark-and-sweep garbage collection technique consist of two stages, a mark and a sweep stage. In the mark stage a live bit is set in all referable memory structures, a pseudo-code example of this can
list* lst_nth(list* listHeader, int n, GC_HeapKeeper* gc)
{
    if(listHeader == NULL)
    {
        return NULL;
    }

    if(n == 0)
    {
        return lst_copy(listHeader, gc);
    }

    if(listHeader->next == NULL)
    {
        return NULL;
    }

    return lst_nth(lst_copyList(listHeader->next, gc), n - 1, gc);
}

Figure 20: List nth function

void mark(heap structure p)
{
    if(!p.live)
    {
        p.live = true;
    }

    for each heap structure q referenced by p
    {
        mark(q);
    }
}

Figure 21: Mark stage of mark-and-sweep garbage collection

be seen in Figure 21. After all referable memory structures have been found
the sweep process begins. During the sweep stage all memory blocks that does
not have the live bit set are deallocated, this can be seen in Figure 22. This
cleans the memory out of structures that can not be referred and allows for new
allocations of referable structures. A mark-and-sweep garbage collector waits
with its execution until the memory resources of the system are exhausted.
Which is when the heap usage reaches a predefined threshold. This gives the
illusion of infinite available memory.

The implementation of the mark-and-sweep garbage collector in this thesis
is based upon two lists. One list for all the allocated structures in the heap
memory and one list for all currently used pointers. These two lists require a
wrapper of the malloc function that inserts each new heap node into the list
void sweep ()
    for each heap structure p
        if (p.live)
            p.live = false;
        else
            free(p);

Figure 22: Sweep of mark-and-sweep garbage collection

in question and for the user to register used stack pointers. If stack pointers
aren’t registered within the garbage collector one might get wild pointers [16].
It also relies on each stack pointer being de-registered at the end of that pointers
function scope. If a pointer is not de-registered one would get memory leaks
since neither the pointer nor the structure which it is referring to would get
deallocated. The reason for this simplistic approach is not to put to much effort
into building a managed runtime environment [15].

During the mark stage of the garbage collector implemented in this thesis a
pass through of the list containing all currently used stack pointers is performed
and the live bit is set in the structures that these pointers refer to. In the sweep
stage a pass through the list of the allocated heap memory executes and all
nodes that are not marked during the mark stage are deallocated.

Instead of letting the garbage collector run unpredictably each task will have
its own garbage collector and this garbage collector will run a pass at the end
of each task. This will force each task to take care of its own memory. This will
imply that task A’s garbage collector will not affect the memory allocated by
task B. This will follow the normal convention of heap usage but each task will
track its own allocations and therefore free its own memory. The benefit of this
is that no locking of the heap is needed during collection runs since the collector
of each task will suspend the normal execution of the task in question but not
affect other tasks. One downside though is that fragmentation of the heap may
occur depending on how the tasks are allocating their memory blocks. This will
also not impose any other strict heap usage boundaries besides the one defined
in the .hpp file for the task in question. An example of heap usage can be seen
in Figure 23. In this figure is the heap usage of three tasks, A, B and C, shown.
At time $t_0$ have all tasks allocated a specific amount of memory and between
time $t_0$ and time $t_1$ is a collection run for task A and C performed and task B
allocates some more memory.

To keep memory usage down in this thesis one full mark and sweep iteration
is performed at the end of every task cycle. This will free unreferenced structures
and keep memory usage at a minimum.
Figure 23: Heap memory usage example
6.4 Implemented tasks

6.4.1 General implementation approach

The general approach to the implementation of this thesis is built upon the idea of databases. All the related imperative state of the task in question is contained within a structure which acts like a database for said task. In this thesis this database is called a task world. The task world is then mutated within each periodic cycle of the task and if a desirable state increment is achieved it will be written back to the currently active world. The currently active world is the current state of the machine, i.e. the current state of the actuators and values of sensors.

The functional implementation of the task consists of five stages:

1. Copy the currently active world
2. Mutation of the world based on external factors
3. Mutation of the world based on internal factors
4. Verification of the new world
5. Write back to the active world

The first step is to copy the currently active world to allow mutation without side-effects and therefore keep the data immutable. This copy is then the one that will be mutated through the other steps and finally written back to the currently active world if it is deemed worthy.

The external mutation of the world is an update of the state that is not controlled by logic, i.e. the value from sensors and the position of the axes. This will get the copied world up to date with the current physical state of the machine. This will also allow for other tasks to affect this one through shared global variables, i.e. in the KnifeCyclic task, the list of activations and deactivations of the tool during On The Fly (OTF) drive is given from the MainCyclic task.

When the external update is finished one moves on to internal mutation. This is where the system logic for a task is placed. It is in this step the world is mutated based upon the behavior of the task and the decision to control the related actuators are decided upon.

When the internal mutation is finished the new mutated world will go through a verification stage where it is scrutinized. If the increment is deemed desirable one moves on to altering the physical aspects of the machine. If not, an error is thrown and execution is stopped since the platform is a hard RTOS. Examples of errors may include, lists not being updated correctly, the right actuators are not actuated at the right cycle and so on.

The final step of this task is to write back the new world to the current active world. Here is where the side-effects are placed and therefore it is performed by an actuation monad.
All of these steps except for the write back have no side effects outside of the passed world and that will honor the principle of referential transparency.

6.4.2 KnifeCyclic

The main tool of the iQ Fusion machine has two driving modes, normal operation and OTF.

In normal operation the knife is actuated when an execute bit is set in the system and the outcome is controlled by a command integer. After the knife command is performed the system waits for a short period so the knife can be securely placed in the expected position.

The OTF mode is a little bit different compared with the normal operation mode. In the OTF mode the actuation of the knife is based on the position of the tool not on the alteration of the execute bit. The point of this drive mode is to allow for continuous horizontal movement of the tool and at the same time actuate the tool in question. This is achieved by giving the knife task a list of positions where the state of the knife should be altered. This list is a First In First Out (FIFO) list. The corresponding command of the first item in this list will be performed when the tool has reached the position of that item. These positions are pre-calculated to compensate for the movement of the tool during knife actuation.

The implementation approach of this task is explained in Section 6.4.1. Pseudo code of how this task is implemented can be seen in Figure 24.

6.4.3 IOControllerCyclic

The I/O controller task is a part of the Human Machine Interface (HMI), which in the iQ Fusion machine consists of three buttons and six LEDs. The I/O controller cyclic is the periodic task that controls the LEDs in the HMI of the iQ Fusion machine.

These six LEDs will give the user information about which state the machine is currently in. There are eight different states that these LEDs provide information about. These are:

1. Not connected
2. Waiting to pause
3. Paused
4. Idle
5. Producing
6. Remove finished box
7. Take label
knifeWorld ExternalMutation(knifeWorld world)
    Update axis position;
    Update knife command;
    Update execute knife command bit;

    if(Otf list updated?)
        Update otf list;

    return updated world;

knifeWorld InternalMutation(knifeWorld world)
    if(Axis passed first item in otf list?)
        if(Should activate knife?)
            Prepare knife activation;
        else
            Prepare knife deactivation;

    if(Should execute knife command)
        if(Knife command is not zero)
            Prepare knife activation;
        else
            Prepare knife deactivation;

    return mutated world;

void PushWorld(KnifeWorld world)
    Acutate knife based on world;
    Reset knife execute command bit;
    Overwrite currently active world with new one;

void CYCLIC KnifeCyclic(void)

    mutatedWorld =
        InternalMutation(
            ExternalMutation(
                CopyWorld(CurrentlyActiveWorld)));

    if (IsValid(mutatedWorld))
        PushWorld(mutatedWorld);
    else
        throw errors;

Figure 24: Pseudo code for the operation of the KnifeCyclic task.
8. Error

The LEDs of the iQ Fusion are placed in two rows, the top row LEDs are currently only used when the machine is not connected to a software server and the bottom row signals the different states the machine may be in. Each LED has four light modes, steady light, slow blink, fast blink and off. If we take the waiting to pause and the paused modes as an example, during the waiting to pause mode white LED will blink rapidly and when the machine is paused the white LED will blink slowly.

This task is built upon the principles explained in Section 6.4.1.

The external mutation of this task consists only of letting the IO controller task know which state the machine is currently in. This is done by shared variables between the IO controller and the rest of the system.

The internal mutation consists of interpreting the current state of the machine and give the LEDs a behavior that corresponds to the current state of the machine.

Pseudo code of how this task is implemented can be seen in Figure 25.

6.4.4 HttpClientCyclic

Information about changes in the machine has to be propagated to overlying software. This is the task for the HttpClientCyclic.

This task works with a list of variables supplied by overlying software. This list contains all the variables overlying software is interested in. When changes are made on these variables a HTTP request containing JavaScript Object Notation (JSON) data is created and sent through Ethernet to the overlying software.

Examples of data the overlying software might have an interest in is: error codes, if the machine has finished production of a box, if the machine is idle and wants more jobs to produce and similar data.

The operation of this task consist of six stages:

1. Copy current task world
2. External mutation
3. Filter out changed variables
4. Parse changed variables into JSON data
5. Create HTTP requests
6. Send requests

The operation is similar to the other implemented tasks but this task is more reliant on higher order functions. The passed list of variables from overlying software is filtered so a list containing only the elements that has been altered since the last iteration of the task. These variables are converted into JSON.

The actual sending of the HTTP requests is done within the function which pushes the imaginary world on top of the actual one. A function that handles
IoWorld ExternalMutation(IoWorld currentWorld)
    update machine state of current world;
    return updated world;

IoWorld InternalMutation(IoWorld WorldAfterExternalMutation)
    switch (How should LEDs be lighted)
    case AllOn: Prepare for All On;
        ...
    return mutated world;

doPushState(IoWorld mutatedWorld)
    light the LED depending on how it prepared;

void CYCLIC IOControllerCyclic(void)

    mutatedWorld =
        InternalMutation(
            ExternalMutation(
                CopyWorld(CurrentlyActiveWorld)));

    if (IsValid(mutatedWorld))
        PushState(mutatedWorld);
    else
        throw errors;

Figure 25: Pseudo code for the operation of the IoControllerCyclic task.
```c
void SendHttpRequest(void* arg)
    Send httprequest through ethernet(arg)

void PushWorld(HttpClientWorld world, fn* SendHttpRequest)
    foreach(variable in changed variables)
        SendHttpRequest(variable);

    Overwrite currently active world with new one;

void _CYCLIC HttpClientCyclic(void)
    globalStateCopy = CopyGlobalState();
    externallyMutaded = ExternalMutation(globalStateCopy);

    httpRequestsGenerated =
        GenerateHttpRequests(
            CreateJSONData(
                Filter(globalStateCopy->variablesToNotifyChanges, hasChanges?)))

    if (IsValid(httpRequestsGenerated))
        PushState(httpRequestsGenerated, SendHttpRequest);
    else
        throw errors;
```

Figure 26: Pseudo code for the operation of the HttpClientCyclic task.

the Ethernet communication is passed as an argument to allow for mock ups in
the test suite. This will prove functionality even though no real communication
is performed within the test suite. After all the requests are sent the list that
contains the old values of the variables is updated with new information.

Pseudo code of how this task is implemented can be seen in Figure 26.
7 Testing

7.1 Introduction

Since this thesis is built upon a case study where an already written system is rebuilt using another development paradigm one needs a way of evaluating the actual difference in quality between the two systems. The following sections is a description of the different metrics used to determine the quality in this thesis presented. These metrics include Code metrics explained in Section 7.1.1.1, CPU utilization, explained in Section 7.1.1.2, memory usage, explained in Section 7.1.1.3 and timing properties, explained in Section 7.1.1.4. The tools used to capture these metrics are explained in Section 7.2.0.6 and Section 7.2.0.6.

7.1.1 Test cases

7.1.1.1 Code metrics

One way of comparing two systems that perform the same task equally well is to compare the code metrics of the different source codes. Code metrics [6] is a set of software measurements calculated to provide the designers and developers a better insight on the quality of the code they have produced. These measurements gave the authors of the source indications of what parts of the systems need to be reworked or rewritten.

In this thesis four metrics were used during the evaluation of the code quality. These were:

1. Lines of code
2. Cyclomatic Complexity
3. Percentage of branch statements
4. Percent of lines with comments

Lines of code

As explained in Section 3.1.3. There is a correlation between code volume and system complexity. A big system tends to be harder to understand compared with a small system. A small system that performs the same task as a big system and equally well tends to be easier to maintain, test and less error prone. This is the reason to why this metric was used in this thesis as a indication of quality.

Cyclomatic complexity

Cyclomatic complexity [65, 64] is a well known code metric that can be a quality indication of software. Cyclomatic complexity gives a sense of how easy the written source is to test, maintain, troubleshoot and the likelihood of errors to surface.

Cyclomatic complexity is calculated by measuring the amount of decision logic in the given source by counting the number of independent paths through the source code. A source block with a lot of decision logic has many independent paths through the source and thus a high cyclomatic complexity. A source block
with a low amount of decision logic results in a small number of independent paths through the source and thus a small cyclomatic complexity. This gives that source with a lot of decision logic will be harder to maintain, troubleshoot and test.

If we look towards Figure 27 and Figure 28. In these two figures we can see two source examples, one with no decision logic and one with a large amount of decision logic. In Figure 28 we have no decision logic and therefore this source snippet has a cyclomatic complexity of 1 since there is only one independent path through the code. If we examine Figure 27 we can see that we have nested switch statements. One switch statement dependent on argument \( a \) with 1000 cases and each case contains a switch statement that depends on argument \( b \) with 1000 cases. This gives \( 1000 \times 1000 \) independent paths through this code block and a cyclomatic complexity of \( 10^6 \). That is to say, the same amount of test cases are needed for a fully tested system.

In this thesis an analysis of the cyclomatic complexity of the written source was performed as an indication of the difference in maintainability and reliability.

**Percentage of branch statements**  This metric is related to cyclomatic complexity which is explained in Section 7.1.1.1. A big percentage of branch statements in the code indicates a system with a lot of decision logic and inherently a system that is hard to understand and test. The difference to cyclomatic complexity is that this metric shows the ratio between decision logic and actual system I/O. If there was an unreasonable amount of decision logic compared with the actual I/O operations to achieve the task at hand one should be re-thinking or redesigning the task in question. This metric in combination with cyclomatic complexity highlighted the differences in maintainability and reliability of the two implementations in question.

**Percentage of lines with comments**  As said by Robert C. Martin [63]

The proper use of comments is to compensate for our failure to express ourself in code. Note that I used *failure*. I meant it. Comments are always failures. We must have them because we cannot always figure out how to express ourselves without them, but their use is not a cause for celebration.

This gives, according to the authors, that:

Comments is the proof of ones failure to communicate through code.

This is a statement that the authors of this thesis hold dear. According to the authors comments is a sign of bad code. Code that is hard to understand needs comments. If one is able to write readable and understandable code there is no need for comments. This metric was an indication of the differences when it comes to readability, reasonability and understandability when the two systems were compared with each other.
void foo(int a, int b)
{
    switch(a)
    {
    case 1:
        switch(b)
        {
        case 1:
            printf('1_Bbar');
            break;
        case 2:
            .
        case 1000:
            printf('1_Cbar');
            break;
        }
    case 2:
        switch(b)
        {
        case 1:
            printf('2_Bbar');
            break;
        case 2:
            .
        case 1000:
            printf('2_Cbar');
            break;
        }
    case 3:
    .
    case 1000:
        switch(b)
        {
        case 1:
            printf('1000_Bbar');
            break;
        case 2:
            .
        case 1000:
            printf('1000_Cbar');
            break;
        }
    break;
    }
}

Figure 27: Example of source with a high cyclomatic complexity
void foo ( void )
{
    printf ( 'Hello world ' );
}

Figure 28: Example of source with a low cyclomatic complexity

**Nested block depth**  Nested block depth is the depth of nested blocks of code. Basically it is the number of nested curly-bracket pairs found in each method. More nested blocks lead to a harder to read and more complex systems. A large nested block depth value can indicate overly complicated code and bad algorithmic design. This metric is an indication of which parts of source that might need some redesign or refactoring [12] and inherently it gave indications of the readability, understandability and maintainability of the systems.

In this thesis SourceMonitor Version 3.4 [13] was used since Automation Studio [3] has no formal tools for calculating code metrics. SourceMonitor is introduced in Section 7.2.0.6.

The results of these quality metrics are shown in Section 8.2.

### 7.1.1.2 CPU utilization
A good metric on how much strain is put on the processing unit of a system is to measure the CPU utilization [40]. CPU utilization is the percentage of time that the CPU is actually processing a program or the operating system of a system. In other words the percentage of time that the CPU is not idle. This metric will indicate the speed of the two different systems and show which system requires less computing power to be able to perform its task.

This metric will both evaluate each separate task and the system as a whole. The reason for that is that since only a part of the system has been rebuilt during this thesis one can probably see a bigger difference on each separate task than the system as a whole.

In Automation Studio [3] there is tools to measure the CPU utilization of a system, this is done with a profiler. The profiler of Automation Studio is explained in Section 7.2.0.6.

### 7.1.1.3 Memory usage
Due to the nature of real-time systems memory resources are often scarce and therefore it is important for the source for this type of systems to be efficient in its usage of memory.

The memory model of Automation Runtime consists of defining a maximum memory size for each task class and then share that memory chunk between the cyclic programs that populates the task class in question, as explained in Section 5.3.1.2.
In this thesis the authors are interested in both the overall memory usage of the whole system and the usage of specific tasks classes. This is for the same reason as the CPU utilization, only parts of the system is built and a bigger difference will probably be observed when comparing only task classes instead of the whole system.

The memory usage of each task class will be tracked with the profiler of Automation Studio, the profiler is explained in Section 7.2.0.6. One of the downsides when tracking memory usage with the Automation Studio profiler is that one can not retrieve the exact memory usage for one cyclic program. It is only possible to look at the memory usage of a whole task class. One thing to note is that the tasks implemented in this thesis are all placed in different tasks classes so one can deduce the increase in memory usage in bytes but the increase in percentage is harder to calculate.

7.1.1.4 Response and Execution time

Response time and Execution time are two important metrics when it comes to real-time systems. These are two metrics that will determine if the tasks in a system can honor the timing constraints set upon them by the designer of the system.

**Execution time**  When it comes to execution time in RTOSs one generally speak of the worst case execution time [95]. Worst case execution time is used to determine the timing behavior of certain systems to prove correctness and reliability. This is important when you have safety critical system where the safety of the systems surroundings is dependent on the correctness of the system. If we look towards the case study in this thesis as explained in Section 5.3.1 the system in question is a hard RTOS and therefore will the timing restrictions on different tasks be enforced rigorously. If a task surpasses its timing constraints the whole system will stop. This puts a lot of importance of reliability when it comes to execution time. This is one of the reasons to why this metric is used in this thesis.

Another reason to why the authors puts importance in this metric is that if the result of the two tasks are equally good one can argue that the system that has lower execution time is the better one since one can often purchase cheaper hardware to a faster system which will decrease the production cost of the system in question.

**Response time**  Response time [54] is execution time with the addition everything else that may affect the task in question. Things that can affect a task in this sense is for instance, jitter [33, p. 490], preemption [33, p. 235], blocking [33, p. 212], etc. There are two ways of measuring the response time. The easiest way is to water mark a task. This is done by letting the system run for a long time and to record the longest response time over the total run. This strategy is good in the sense that it is easy to perform but it might not capture the actual worst case response time. The other way of doing it is by
static analysis. With static analysis one calculates the response time with the help of Equation 1.

\[ R_i = \frac{hpi}{\sum_{j=1}^{\infty} R_i/T_j * C_j} + C_i \]  

As seen in Equation 1, the response time of a task \( i \) is given by the sum of the response time for \( i \), divided by the period of all higher priority tasks \( j \) times the worst case execution time of \( j \) with the addition of the worst case execution time of \( i \). This gives that the quality of a response time is heavily reliant on a good estimation of the worst case execution time of the tasks in question.

Watermarking has been used for both the estimation of the worst case execution time and the worst case response time and both of these water markings will be performed with the integrated profiler in Automation studio. The profiler is explained in Section 7.2.0.6.

7.1.1.5 User evaluation A user study was performed where 5 people from Packsize Technologies inspected the source code written and compared it to the old system. Both systems was graded on a scale from 1 to 10, where 10 is a perfect system and 1 is a system that is completely incomprehensible, on these properties:

1. Readability
2. Simplicity
3. Resonability
4. Testability
5. Cleanliness
6. Overall quality

All of these measurements were subjective measurements based on the independent users own preferences.

7.2 Testing strategy

A brief description of the tools that were used for testing in this thesis are presented below.

7.2.0.6 Tools

SourceMonitor 3.4 Source monitor is a free-ware from Campwood Software that allows you to calculate code metrics for source written in C++, C, C#, VB.NET, Java, Delphi, Visual Basic (VB6) and HTML. Metrics that can be calculated include, Number of lines, number of statements, percentage of branches, maximum cyclomatic complexity, average cyclomatic complexity, etc.
Automation Studio Profiler  The Automation Studio Profiler can be used to measure and display important system data such as task runtimes, system and stack loads, memory usage, etc. This can be necessary to determine how much idle time remains in a task class or which task generates a sporadic increased demand for time.
8 Results and Discussion

8.1 Testability

The TDD development strategy was used during this thesis. This is something that had earlier been tried with the development of the iQ Fusion at Packsize Technologies. During the initial stages of the development process tests was written using a mocking library called Hippo Mocks [1]. But as development progressed more and more time had to be dedicated to maintaining the test suite due to the nature of Hippo Mocks. The TDD development process was later abandoned. The reason for using Hippo mocks was that the facilities provided by the hardware supplier did not contain any sustainable way to write a good testing suite for the source code.

The authors could, with the help of a small lightweight testing tool called CuTest [7], produce source code in a rapid fashion without access to the target platform. Functionality was proven with the help of unit tests. This gave us confidence in the source written and the ability to move forward without worrying about defects in the code.

With the principle of referential transparency and functions without side-effects, which are core concepts within the functional programming paradigm, the authors could build unit tests that tested whether or not the world was mutated in a correct fashion given the arguments and only the arguments of the function in question. This proved to be an easy way to implement unit tests which could prove functionality and in the end one could easily integrate the unit tested source into the target system by supplying the domain specific data instead of imaginary data.

When domain specific methods had to be used e.g. during handling of Ethernet traffic, one could pass these functions as arguments with function pointers. This gave the ability to easily create mock-ups of these functions with the same signature in the testing suite and through that prove functionality of the underlying logic even though the platform was not available.

8.2 Code metrics

8.2.1 KnifeCyclic

Below is the code metrics for the KnifeCyclic task presented. One thing to note is that two versions of the functional implementation is shown, one with the underlying libraries and one without the underlying libraries. A more detailed table of the specific metrics for the original version of this task can be seen in Appendix H, the details for the functional implementation with libraries can be seen in Appendix I and the details for the functional implementation without the libraries can be seen in Appendix J.
### 8.2.2 IOControllerCyclic

Below is the code metrics for the IOControllerCyclic task presented. One thing to note is that two versions of the functional implementation is shown, one with the underlying libraries and one without the underlying libraries. A more detailed table of the specific metrics for the original version of this task can be seen in Appendix K, the details for the functional implementation with libraries can be seen in Appendix L and the details for the functional implementation without the libraries can be seen in Appendix M.

<table>
<thead>
<tr>
<th></th>
<th>Functional</th>
<th>Functional (No libs)</th>
<th>Original</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of code</td>
<td>1161</td>
<td>590</td>
<td>523</td>
</tr>
<tr>
<td>Percent Branch Statements</td>
<td>11.9%</td>
<td>4.8%</td>
<td>15.7%</td>
</tr>
<tr>
<td>Percent Lines With Comments</td>
<td>4.5%</td>
<td>0.8%</td>
<td>2.9%</td>
</tr>
<tr>
<td>Maximum Cyclomatic Complexity</td>
<td>8</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>Average Cyclomatic Complexity</td>
<td>2.13</td>
<td>1.52</td>
<td>2.39</td>
</tr>
<tr>
<td>Maximum Nested Block Depth</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Average Nested Block Depth</td>
<td>0.88</td>
<td>0.55</td>
<td>1.16</td>
</tr>
</tbody>
</table>

### 8.2.3 HttpClientCyclic

Below is the code metrics for the HttpClientCyclic task presented. The original HttpClientCyclic is written in structured text so calculating code metrics for that version is not possible with the help of Source Monitor. Therefore is only the two versions of the functional metrics of the task in question is presented in the table below. A more detailed table of the specific metrics for the functional version with libraries of this task can be seen in Appendix N and details for the functional implementation without the libraries can be seen in Appendix O.

<table>
<thead>
<tr>
<th></th>
<th>Functional</th>
<th>Functional (No libs)</th>
<th>Original</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of code</td>
<td>983</td>
<td>550</td>
<td>378</td>
</tr>
<tr>
<td>Percent Branch Statements</td>
<td>10.4%</td>
<td>7.4%</td>
<td>15.8%</td>
</tr>
<tr>
<td>Percent Lines With Comments</td>
<td>3.8%</td>
<td>3.1%</td>
<td>4.0%</td>
</tr>
<tr>
<td>Maximum Cyclomatic Complexity</td>
<td>8</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>Average Cyclomatic Complexity</td>
<td>2.2</td>
<td>1.89</td>
<td>3.57</td>
</tr>
<tr>
<td>Maximum Nested Block Depth</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Average Nested Block Depth</td>
<td>0.91</td>
<td>0.61</td>
<td>1.06</td>
</tr>
</tbody>
</table>
8.3 CPU utilization

Below is the results of for the CPU utilization of the original and the functional versions of the system presented. The unit used in the results is the percentage of full CPU utilization the difference is presented in both Percentage Points (PP) and percentages. A more detailed table on the CPU utilization for the original version of the system can be seen in Appendix E and details of the functional implementation of the system can be seen in Appendix F.

<table>
<thead>
<tr>
<th></th>
<th>Functional</th>
<th>Original</th>
<th>Δ PP</th>
<th>Δ %</th>
</tr>
</thead>
<tbody>
<tr>
<td>KnifeCyclic</td>
<td>5.355%</td>
<td>0.322%</td>
<td>5.033%</td>
<td>16063%</td>
</tr>
<tr>
<td>IOControllerCyclic</td>
<td>0.030%</td>
<td>0.005%</td>
<td>0.025%</td>
<td>600%</td>
</tr>
<tr>
<td>HttpClientCyclic</td>
<td>41.003%</td>
<td>38.096%</td>
<td>2.907%</td>
<td>108%</td>
</tr>
<tr>
<td>Total</td>
<td>46.388%</td>
<td>38.423%</td>
<td>7.965%</td>
<td>121%</td>
</tr>
</tbody>
</table>

8.4 Memory usage

Below is the results for the memory usage of the system presented. The unit used in the results is number of Bytes (B). More detailed tables for both the original implementation and the functional implementation can be seen in Appendix G. One thing worth to note is that to be able to get the HttpClientCyclic task up and running the stack size for that task had to be doubled.

<table>
<thead>
<tr>
<th></th>
<th>Total stack</th>
<th>Functional</th>
<th>Original</th>
<th>Δ B</th>
<th>Δ %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td>32762 / 24570</td>
<td>11340</td>
<td>4060</td>
<td>7280</td>
<td>279%</td>
</tr>
<tr>
<td>KnifeCyclic</td>
<td>8190</td>
<td>2808</td>
<td>1692</td>
<td>1116</td>
<td>165%</td>
</tr>
<tr>
<td>IOControllerCyclic</td>
<td>8190</td>
<td>700</td>
<td>400</td>
<td>300</td>
<td>175%</td>
</tr>
<tr>
<td>HttpClientCyclic</td>
<td>16382 / 8190</td>
<td>7832</td>
<td>1968</td>
<td>5864</td>
<td>397%</td>
</tr>
</tbody>
</table>

8.5 Response and Execution time

Below is the results for response time and execution time presented. The unit used to present the results is microsecond (µs). A more detailed table for the timing properties of the original version of the system can be seen in Appendix E and a more detailed table for the functional version of the system can be seen in Appendix F.

8.5.1 Response time

<table>
<thead>
<tr>
<th>Original System</th>
<th>Max</th>
<th>Min</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>KnifeCyclic</td>
<td>117.370µs</td>
<td>2.160µs</td>
<td>3.211µs</td>
</tr>
<tr>
<td>IOControllerCyclic</td>
<td>10.140µs</td>
<td>8.380µs</td>
<td>8.888µs</td>
</tr>
<tr>
<td>HttpClientCyclic</td>
<td>6779.330µs</td>
<td>2.180µs</td>
<td>4101.868µs</td>
</tr>
</tbody>
</table>
8.5.2 Execution time

8.6 Profiling properties

The data for the CPU utilization, Memory usage, Response time and Execution time was collected with the help of the automation studio profiler. Below is table that shows the number of data points collected during the profiling. More information about the data collected by the profiler can be seen in Appendix E.

8.7 User evaluation

Below are sections of tables presenting the results from the user study. The tables are organized in the following way, User, Readability (R), Simplicity (S), Reasonability (Re), Testability (T), Cleanliness (C), Overall quality (Q) and finally is a summation of all these properties, at the end of each table is the
average of all the participants answers presented. The three sections are split in
a line-up for the original system, one for the functional system and one for the
difference between the original and functional system where positive numbers
favor the functional system and negative numbers favor the original system. In
these tables the character - represents a missing data point, these are not taken
into account when calculating the averages.

Notes from the evaluators are presented at the end of this section.

8.7.1 Original system

Below is the user evaluation presented for the original system.

KnifeCyclic:

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>S</th>
<th>Re</th>
<th>T</th>
<th>C</th>
<th>Q</th>
<th>Σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>7</td>
<td>41</td>
</tr>
<tr>
<td>User 2</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>35</td>
</tr>
<tr>
<td>User 3</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>User 4</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>47</td>
</tr>
<tr>
<td>User 5</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>33</td>
</tr>
<tr>
<td>Average</td>
<td>7.2</td>
<td>7.4</td>
<td>7.2</td>
<td>4.8</td>
<td>6.2</td>
<td>6.5</td>
<td>39</td>
</tr>
</tbody>
</table>

IoControllerCyclic:

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>S</th>
<th>Re</th>
<th>T</th>
<th>C</th>
<th>Q</th>
<th>Σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>5</td>
<td>8</td>
<td>8</td>
<td>45</td>
</tr>
<tr>
<td>User 2</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>37</td>
</tr>
<tr>
<td>User 3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>User 4</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>7</td>
<td>9</td>
<td>50</td>
</tr>
<tr>
<td>User 5</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>1</td>
<td>8</td>
<td>7</td>
<td>39</td>
</tr>
<tr>
<td>Average</td>
<td>6.8</td>
<td>7.2</td>
<td>7</td>
<td>4.8</td>
<td>6.2</td>
<td>7.25</td>
<td>42.75</td>
</tr>
</tbody>
</table>

HttpClientCyclic:

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>S</th>
<th>Re</th>
<th>T</th>
<th>C</th>
<th>Q</th>
<th>Σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>6</td>
<td>28</td>
</tr>
<tr>
<td>User 2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>User 3</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>-</td>
<td>6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>User 4</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>User 5</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>30</td>
</tr>
<tr>
<td>Average</td>
<td>5</td>
<td>4.2</td>
<td>5</td>
<td>1.25</td>
<td>4.2</td>
<td>4</td>
<td>21</td>
</tr>
</tbody>
</table>

8.7.2 Functional system

Below is the user evaluation presented for the functional system.

KnifeCyclic:
### IoControllerCyclic:

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
<th>Re</th>
<th>T</th>
<th>C</th>
<th>Q</th>
<th>Σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>48</td>
</tr>
<tr>
<td>User 2</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>44</td>
</tr>
<tr>
<td>User 3</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>User 4</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>7</td>
<td>42</td>
</tr>
<tr>
<td>User 5</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>41</td>
</tr>
<tr>
<td>Average</td>
<td>7.4</td>
<td>7.2</td>
<td>7.8</td>
<td>6.8</td>
<td>7.5</td>
<td>43.75</td>
</tr>
</tbody>
</table>

### HttpClientCyclic:

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
<th>Re</th>
<th>T</th>
<th>C</th>
<th>Q</th>
<th>Σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>46</td>
</tr>
<tr>
<td>User 2</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>5</td>
<td>41</td>
</tr>
<tr>
<td>User 3</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>-</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>User 4</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>5</td>
<td>35</td>
</tr>
<tr>
<td>User 5</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>40</td>
</tr>
<tr>
<td>Average</td>
<td>6.2</td>
<td>6.2</td>
<td>6.8</td>
<td>7.25</td>
<td>5.8</td>
<td>7</td>
</tr>
</tbody>
</table>

### 8.7.3 Difference

Below is the difference of the points of the different categories presented.

#### KnifeCyclic:

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
<th>Re</th>
<th>T</th>
<th>C</th>
<th>Q</th>
<th>Σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>User 2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>User 3</td>
<td>1</td>
<td>-1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>User 4</td>
<td>-1</td>
<td>-2</td>
<td>-3</td>
<td>1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>User 5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>Average</td>
<td>0.2</td>
<td>-0.4</td>
<td>0</td>
<td>3</td>
<td>0.6</td>
<td>1</td>
</tr>
</tbody>
</table>

#### IoControllerCyclic:

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
<th>Re</th>
<th>T</th>
<th>C</th>
<th>Q</th>
<th>Σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>User 2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>User 3</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>User 4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>User 5</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Average</td>
<td>0.8</td>
<td>1.0</td>
<td>3.2</td>
<td>1.4</td>
<td>1.25</td>
<td>6.25</td>
</tr>
</tbody>
</table>
HttpClientCyclic:

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>C</th>
<th>Q</th>
<th>Σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>User 2</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>User 3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>User 4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>User 5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Average</td>
<td>2.4</td>
<td>3.2</td>
<td>2.2</td>
<td>6</td>
<td>2.4</td>
<td>3</td>
</tr>
</tbody>
</table>

8.7.4 User notes

User 1 Notes

Original System

**KnifeCyclic** The cyclic task is really easy to understand, it is just a bunch of cases. The KnifeController harder to follow than the cyclic task, but it is not that bad.

**HttpClientCyclic** It is a bit of a mess. Everything is nested in different kind of loops, but without too much trouble you understand what it does.

**IoControllerCyclic** It is really straight forward and easy to understand what it does. There is almost no nesting and it contains no loops.

Functional System

**KnifeCyclic** I did not find the old implementation to be that bad but I do find it easier to read and reason about the code in the functional system.

**HttpClientCyclic** In the old implementation you had all the code in a single file and there were not that many lines of code, on the other hand it was a lot harder to understand or get a direct overview of what it actually did. In the functional system on the other hand the code is split between a couple of files but it is actually a lot easier to read and reason about the code. Everything is split into functions with good names so you will at least have an understanding of what the code does before you read the code, which makes it easier.

**IoControllerCyclic** It is just as readable as the old implementation, if not more since the variable/function names are better/easier to reason about.

User 2 Notes
Original System

KnifeCyclic Naming is a little unclear at times with command EXECUTE and state KNIFEXECUTE etc. There is also logging implementation details (strpcy(message, result) etc) directly in the code that is unclear how it works.

HttpClientCyclic There is no abstraction, naming or commenting which makes it really clunky and hard to understand.

IoControllerCyclic The implementation feels a little too complex considering the simplicity of the task.

Functional System

KnifeCyclic The logic in the task is really easy to understand because it is short and uncluttered, good abstraction.

User 3 Notes

Original System

KnifeCyclic Pattern use of statemachine and moving through the states are commonly understood. Code could give some main comment on what system variables will be used and what the expected outcomes. It is not clear what the differnerence is of Execute and OtfExecute. Code is assuming or knows that it will not be completed in the allotted cyclic time.

HttpClientCyclic Class does not look to be consistent with the other two samples. Code seems to be functioning on an ever changing array and is easy to follow.

IoControllerCyclic I dislike how some of the code is encapsulated in a method (DoGreenLightControl) but then just above it there is another bit of code that is manaully controlling it (GreenLed-:lightOff). The logic should be put together and work from some state of all things that needs to be considered.

Functional System
KnifeCyclic  Perhaps I am wrong but by modifying a copy of the current state and not changing the reference values like in the old system it would be harder to find out at what point or what value caused the system to go into an error state?

HttpClientCyclic  The Cyclic looks a little different than the other samples. I am supposing because this may need to run over multiple iterations? Comments should be present as to why it looks different than the others. I am not seeing why a function (sendHttpRequest & CreateJSONData) is sent as method pointers instead of including in the only code package that is using them (functionalhttpclient) it seems like there is some code scatter here where everything else seems to be in close proximity to it is usage.

IoControllerCyclic  Cyclic pattern is consistent and easy to ignore this repeated class and go straight to the class with the meat. I like the fact that all all of the logic is located in a single class/file. I did not notice anything in the logic where one was dependent on another, i.e. if green light is on and status = x then turn on red light.

User 4 Notes

Original System

KnifeCyclic  outIndex is declared in KnifeCyclic without being used. The way that TON (timers in automation studio) is used is a little bit strange. Logging is affecting cleanliness and to some extent readability negative.

HttpClientCyclic  The code is just hard to read and reason about. Testing can only be done with external client using the service.

Functional System

KnifeCyclic  It feels strange that MutateKnifeWorldBasedOnExternalFactors sets members of source and at the next step source is set to return value of GenerateOtfPositionList. No logging makes it cleaner but the current implementation does not fully replicate the original system when it comes to logging.

HttpClientCyclic  Testability is greatly increased due to separation of world mutation and execution. Also the HttpClient function block is nicely abstracted.
User 5 Notes

Original System

KnifeCyclic Straightforward cyclic but nests down into many sub libraries/-functions makes it difficult to overview complete functionality.

IoControllerCyclic Small function that is good implemented as is. Clean interfaces and not very long function call chains.

Functional System

KnifeCyclic Overall against Original system and OOP used there. Recurring way for calls regardless of functionality of code. Clearer points where changes to system can be made. Easier to get a full grip of the complete function than Original system.

8.8 Discussion

8.8.1 Code Metrics

If we look towards the code metrics presented in Section 8.2. One can notice that there are a lot more code in the functional implementations of both the KnifeCyclic task and the IOController task. One has for the KnifeCyclic a 100% increase and in the IOController almost a 240% increase in code volume. This points towards a bigger and more complex system. But if one looks to the rest of the metrics one can see that the functional implementation is better in most cases and at least on par with the original system.

According to the users, the functional system is easier to understand and reason about. This is due to the strict enforcement of referential transparency. It also gives the ability of unit testing which ensures proven system functionality.

According to the authors, the cyclomatic complexity and the percentage of branch statements are the two biggest indicators of the complexity of the system. And if one compares the functional implementations to the original one one can conclude that the functional implementation is the less complex one.

8.8.2 Timing Properties

The data presented in Section 8.5 indicates that the functional implementation is on average a lot slower. One can see that the functional system is a lot more stable when it comes to execution time and response time though. The differences between maximum and minimum execution time lies, for the KnifeCyclic task and the IOController task, only 10 - 40\mu s from each other. So the functional implementation is a lot more consistent when it comes to execution time and response time.
8.8.3 System Utilization

8.8.3.1 CPU Utilization One can easily conclude that the demand of the system is much higher on the functional implementation. And due to the different nature of the implementations one can conclude that it is mainly due to the usage of the garbage collector. The garbage collector has longer collection routes in the KnifeCyclic compared to for instance the IOController and that shows in the percentage increase of the CPU utilization of these two tasks as seen in Section 8.3.

One thing to note is that if one looks towards the total utilization of both the memory and the CPU one can see that there is a lot of available resources for a more demanding system. This gives that that, according to us, it might be a good idea to use this approach to a system of this type since one has a validation stage in each cycle that will practically unit test the result from each module before it will be allowed to affect the system. This in hand would give the ability of more thorough error handling and reporting since one can in the validation append information to system logs for error reporting and also trigger automatic error handling that preferably would pass by end users without them noticing.

8.8.3.2 Memory With the results presented in Section 8.4 one can see that the memory usage for the functional implementation has increased with around 70%. This was expected due to the enforcement of immutable data structures. The upside here is that this does not cause any problems to the system in question since the memory usage is still low compared to the amount that was given to each task class. One could argue that the tasks implemented in this thesis are relatively small so this effect would increase when re-implementing bigger tasks of the system. But since the overall memory usage of the system is stable around 70% one still has a lot of room in the target system for more memory usage.

8.8.4 User Study

The results presented in Section 8.7 clearly shows that the functional system has a higher quality compared with the original system. The task where it is most noticeable is the HttpClientCyclic task the average difference for this task lies between 2.2 and 6 points with a summed average of 19.5 points over the original system. This according to us clearly indications that the functional system holds higher quality. The difference between the other two tasks is smaller but there is only one data set that favors the original system and that is User 4s evaluation of the KnifeCylic task all the other data sets favor the functional implementation. The category where the functional system is mostly favored is the testability. The average difference between the original system and the functional system lies between 3 and 6 points which is according to us strong evidence of the testability of this approach to software development.
9 Conclusions

9.1 Results summary

9.1.1 Questions

Below is the conclusions, according to the authors, to the questions posed in Section 2 and Section 5.

What are the trade offs for the functional language paradigm compared with the imperative or object oriented language paradigms in the real-time domain? The trade offs are CPU utilization and memory usage for stability, reasonability and testability. The functional implementation will give a more stable system since one can unit test the results before each incremental change is imposed on the system, this will allow for better error handling and reporting as mentioned in Section 8.8.1. The functional implementation will also give more testable systems since one can abstract away platform specific method calls with the help of function pointers to mock ups and through that unit test systems without much effort.

Will a functional implementation have a higher or lower complexity? Where would the complexity lie? The functional implementation has a lower complexity compared with the original system. The complexity of the functional implementation lies mostly within the garbage collector and the underlying libraries. The actual production code has low complexity if one bases complexity on cyclomatic complexity and maximum nested block depth. One thing to note is that if one would drop the garbage collector and underlying libraries from the calculation of code metrics one would see that the average cyclomatic complexity would drop with on average 1 point as seen in Appendices J, O and M.

Also, according to Section 3.1, one of the main reason for complexity is the way of handling imperative state. This thesis has focused a lot on building a system that handles imperative state in a good way and this is achieved according to us. Due to this way of state handling the code volume of the system has increased mainly because of the implementation of the garbage collector and underlying libraries but the functional implementation is without mutation of imperative state outside of special constructs. This will effectively decrease the complexity when reasoning about the system. Besides that, with a sequence of copying the current world, mutation based on external factors, mutation based on system logic, validation and finally applying the mutated world, each cyclic task has a uniform execution order and clear boundaries between each stage. This also helps with decreasing the complexity of the system when it comes to general reasonability.

Will the functional paradigm yield more readable, stable and reliable systems, compared with similar systems? The material in this thesis
points towards that the functional paradigm yields more readable, stable and reliable systems. As for readability, the copy-mutate-validate-push logic and the idea of encapsulating all of the related imperative state into a task world is much easier to read and reason about compared with the original system. As for stability and reliability of the system, due to the ability of development with the TDD principles and the validation step in each cycle before applying the new world. This gives the system a good way of detecting errors and handling them appropriately. The user study also implies that the readability of the system has increased and that the system is very testable which will yield stable and reliable systems.

How will the code metrics differ for a system written with the functional paradigm compared with a similar system in either the object oriented or imperative paradigm? If we look towards the results presented in Section 8.2.1 and Section 8.2.2. With the tests performed in this thesis one can conclude that C programs with a functional style has a larger code volume mostly due to the need for a garbage collector but the amount of logic operations needed to achieve the same result is lower or at least on par with a object oriented system. One thing to note here is that the validation stage is taken into account here so one has lower complexity and decision logic with a more reliable system but with some code volume overhead.

This yields, according to the authors, that functional systems are better than imperative and/or object oriented systems when it comes to the complexity of the source in question.

Is it beneficial to combine the functional paradigm with other paradigms? This question depends on if one speaks of combining functional and i.e. object oriented topics in the same code block or in different tasks. In this thesis is only objects used in the functional blocks for data retrieving without side effects which could easily be removed and exchanged for a method that does the same thing. The only reason to why the authors used objects for this is that it was already implemented and a part of the old system. When it comes to combining different paradigms in different tasks we have come across no problems with having parts of the system object oriented and parts functional. The only thing is that it is important to declare clear boundaries between the tasks so the side-effect-based code of the object oriented tasks does not affect the functional tasks.

If we look towards the current state of modern programming languages one can see that more and more functional mindsets are setting in, i.e. lambdas in C# as mentioned in Section 3.2. This gives that even object oriented languages are incorporating more and more functional aspects into their languages, this points towards benefits in combining these paradigms.

Would a functional implementation allow for thorough unit testing that would increase maintainability and stability of the system? Yes,
Is the functional programming paradigm suitable for embedded/real-time systems? This is a double-edged sword, it depends a lot on what kind of system one is developing. The system used in this case study was severely over dimensioned when it comes to CPU power and the size of available memory. So the authors had, for the target system, no issues with memory or timing properties but if one would use lower grade hardware one would definitively have issues with both CPU usage and memory usage.

So is it suitable? Yes and no. If one has a system with a lot of power and memory one could argue that the advantages shown when it comes to complexity is more beneficial than the increased usage of the system.

9.2 Limitations

Since only parts of the whole control system for the iQ fusion is developed in the functional paradigm one can not determine the actual difference between the two systems. One can only look at the tasks in question and compare these to each other. This yields implications of pros and cons of the functional paradigm but it can not give a decisive conclusion of the pros and cons of the functional paradigm in this application.

When it comes to the memory management of Automation Runtime there is no official documentation on the architecture. Therefore one has to take the officials the authors has been in contact with on their word when it comes to that. This is a huge limitation but due to the proprietary nature of the system in question one can not really do anything about it but to trust the B&R officials.

There are also limitations when it comes to the data collection in this thesis. The profiler of Automation Studio can not track memory usage for one cyclic program but it will track usage for a whole task class. This will give indications of the increased memory usage since each of the tasks implemented in this thesis are placed in different task classes, but this will put limitations on the analysis of memory usage since the real increase in memory usage of a specific task can not be calculated.

In the interest of speed, Source Monitor does not fully parse source the same way as a compiler would. Instead Source Monitor searches through the source for tokens to base its metrics calculations upon. This gives, according to the
authors, sufficient data but it’s not as exact as if the source would be parsed the same way as a compiler does it [52, 14].

The user study in this thesis has its pitfalls. The data set is very small with only 5 data points for each property of each task. This will only yield lose implications on the difference between the two systems. Also the grading system had no formal definition except for the extreme points, 1 and 10, this was on purpose since the authors wanted the person evaluating the system to decide for him/herself what the different points meant in the context of each evaluation property.

9.3 Implications

The goal of this thesis was to redevelop a industrial control system using the essence of the functional paradigm. This thesis has shown that its possible to extract the essence of the functional paradigm and use an imperative language for the implementation. This thesis has also shown that TDD is possible when it comes to industrial real-time systems if one enforces referential transparency and mocks the sensors and actuators of the system in question.

The collected data of code metrics implies that a functional implementation will have lower complexity but a higher code volume. When we speak of complexity we mean cyclomatic complexity in combination with both percentage of branch statements and comments.

The collected data when it comes to memory usage and CPU utilization implies that the functional paradigm creates a lot of overhead compared to an imperative or object oriented system. Mainly to the necessity of a garbage collector and immutable data.

The implementation of this thesis has yielded a system where the trade offs has been CPU utilization and memory usage in favor of a more reliable and simplistic system with a smaller part of decision logic.

The overall implications of this thesis are that the functional paradigm is possible to use within the domain of embedded real-time systems but one has to think about what kind of hardware one is using. If there is a lot of extra space both in CPU utilization and memory one should consider the functional paradigm over an object oriented or imperative approach due to testability and maintainability due to lower complexity of the system.

9.4 Future work

In this thesis the authors have finished the implementation and testing of three cyclic tasks of the IQ Fusion control system. With this an analysis of pros and cons of the functional paradigm compared with the original imperative and object oriented system has been performed. It does give a hint of the performance of the functional paradigm, but a system that only uses the functional paradigm will produce a more comprehensive and more accurate results. Therefore, developing the whole control system and evaluating the performance of the whole thing is the majority of the future work.
Due to compiler issues, the authors were forced to use an imperative language with a functional approach in this thesis. Implementing the system in a real functional language would be of great interest with regards to all of the data collected during this thesis.

Besides, the authors who are newly graduated M.Sc students, to some extent, will produce optimizable code as lack of experience. It may effect the fairness of the results. Thus, code optimization should also be taken into account as a part of polishing the system to diminish the influence of personal programming skill of the authors on the results.

The garbage collector of this thesis is the main source of complexity and CPU utilization in the implementation. This is due to the lack of effort that has been put into optimizing the garbage collector. Thus developing the garbage collector further is also one of the main tasks for further development. This is where the authors believe that the most progresses can be made regarding timing properties, CPU utilization and complexity. In addition, the garbage collector of this thesis relies a lot on its being used correctly. Instead complete automatic memory management when it comes to allocation of memory, registration and de-registration of the pointers that are currently in use is reliant on the user. That is to say, a automatic garbage collector as in high-level programming languages like C#, LISP and Java, and light-weight in execution time and CPU utilization is one of the milestones for the future work.

Last but not the least, in order to adopt TDD development strategy, some mock-ups for sensors and actuators has to be created to free the testing from using the actual platform during development. Therefore building a sufficient mocking library should also be part of future work. The mocking library should contain all mock-ups that share the function signatures with some domain specific methods which are unaccessible out of the Automation Studio platform. This in combination with assertable call counters and information gathering of input arguments will provide the test suite with sufficient tools for machine development without the use of a physical machine. This promote the usage of the TDD development cycle and yield the benefits of said development method.
References


A Type declarations

// List types
typedef struct list
{
    void * item;
    int size;
    struct list * next;
} list;

// Memory management types
typedef struct HeapNode
{
    void* gcNS_nodeAllocated_ptr;
    int gcNS_nodeAllocated_size;
    BOOL gcNS_nodeAllocated_used;
    struct HeapNode* gcS_nodeAllocated_next;
}GC_HeapNode;

typedef struct HeapKeeper
{
    GC_HeapNode* gcHS_HeapKeeper;
    GC_PointerNode* UsedPointers;
    int gcHS_HeapKeeperSize;
}GC_HeapKeeper;

// Monad types
typedef struct maybeInt
{
    int something;
    int just;
} maybeInt;

typedef struct monadStep
{
    void ** args;
    void(*fn)(void **);
    struct monadStep * next;
} monadStep;

#endif

92
B API : Memory management

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "../Types/types.hpp"

void GC_Init(GC_HeapKeeper** gc);
void * GC_malloc(int node_size , GC_HeapKeeper* gc);
void GC_RegisterPointer(void** ptr , type pointerType , GC_HeapKeeper* gc);
void GC_DeregisterPointer(void** ptr , GC_HeapKeeper* gc);
void GC_Collect(GC_HeapKeeper* gc);

#ifndef TESTING
#define TESTING
    int GC_usedPointerCounter(GC_PointerNode* head);
#endif

void GC_Init(GC_HeapKeeper** gc) {
    *gc = (GC_HeapKeeper*)malloc(sizeof(GC_HeapKeeper));
    (*gc)->gcHS_HeapKeeper = NULL;
    (*gc)->UsedPointers = NULL;
    (*gc)->gcHS_HeapKeeperSize = 0;
}

void* GC_malloc(int node_size , GC_HeapKeeper* gc) {
    GC_HeapNode* HeapNodeNew;
    if (node_size > 0) {
        HeapNodeNew = (GC_HeapNode*)malloc(sizeof(GC_HeapNode));
        HeapNodeNew->gcNS_nodeAllocated_ptr = malloc(node_size);
        HeapNodeNew->gcNS_nodeAllocated_size = node_size;
        HeapNodeNew->gcNS_nodeAllocated_used = false;
        HeapNodeNew->gs_nodeAllocated_next = NULL;
        if (GC_AppendNewtoListEnd(HeapNodeNew, gc->gcHS_HeapKeeper, gc))
{  
gc->gcHS_HeapKeeperSize += node_size;
  
return HeapNodeNew->gcNS_nodeAllocated_ptr;
}
free(HeapNodeNew->gcNS_nodeAllocated_ptr);
free(HeapNodeNew);
return NULL;
}

BOOL GC_MarkNode(GC_HeapNode * currentNode , void* HeapNode_ptr)
{
  if (currentNode == NULL)  
  {  
    return false;
  }
  if (currentNode->gcNS_nodeAllocated_ptr != HeapNode_ptr)  
  {  
    return GC_MarkNode(currentNode->gcS_nodeAllocated_next , HeapNode_ptr);
  }
  currentNode->gcNS_nodeAllocated_used = false;
  return true;
}

BOOL GC_MarktoRemove(void* ptr , GC_HeapKeeper* gc)
{
  if (ptr == NULL)  
  {  
    return false;
  }
  return GC_MarkNode(gc->gcHS_HeapKeeper , ptr);
}

int GC_MarkUsedNode(void ** stackPtr , GC_HeapNode* current)
{
  if(current == NULL)  
  {  
    return -1;
  }
  if(*stackPtr == (void*)current->gcNS_nodeAllocated_ptr)  
  {  
    current->gcNS_nodeAllocated_used = true;
    return 1;
  }
  return GC_MarkUsedNode(stackPtr , current->gcS_nodeAllocated_next);
}

94
```c
void markNodeBasedOnType(void* pointer, type pointerType, GC_HeapKeeper* heap)
{
    if (pointer == NULL)
    {
        return;
    }
    switch (pointerType)
    {
    case LIST:
        #ifdef MARKTRACE
            printf(" List passed \n");
        #endif
        GC_MarkUsedNode((void**)&((list*)pointer)->item, heap->gcHS_HeapKeeper);
        if (((list*)pointer)->next != NULL)
        {
            if (GC_MarkUsedNode((void**)&((list*)pointer)->next, heap->gcHS_HeapKeeper) == 1)
            {
                markNodeBasedOnType((void*)((list*)pointer)->next, LIST, heap);
            }
        }
        break;
    case KNIFEWORLD:
        #ifdef MARKTRACE
            printf(" knifeworld passed \n");
        #endif
        if (((KnifeWorld*)pointer)->OTFTogglePosition != NULL)
        {
            #ifdef MARKTRACE
                printf(" toggle not null \n");
            #endif
            if (GC_MarkUsedNode((void**)&((KnifeWorld*)pointer)->OTFTogglePosition, heap->gcHS_HeapKeeper) == 1)
            {
                markNodeBasedOnType((void*)((KnifeWorld*)pointer)->OTFTogglePosition, LIST, heap);
            }
        }
    }
```
break;

case HTTPCLIENTWORLD:
    #ifdef MARKIRACE
        printf("HttpClientWorld\n passed\n") ;
    #endif

    if(GC_MarkUsedNode((void**)&((HttpClientWorld *)pointer)->changedVariables,
he->gcHS_HeapKeeper) == 1)
    {
        markNodeBasedOnType((void**)((HttpClientWorld *)pointer)->changedVariables,
he);}
    if(GC_MarkUsedNode((void**)&((HttpClientWorld *)pointer)->changedVariablesInJSON,
he->gcHS_HeapKeeper) == 1)
    {
        markNodeBasedOnType((void**)((HttpClientWorld *)pointer)->changedVariablesInJSON,
he);}
    if(GC_MarkUsedNode((void**)&((HttpClientWorld *)pointer)->notificationVariables,
he->gcHS_HeapKeeper) == 1)
    {
        markNodeBasedOnType((void**)((HttpClientWorld *)pointer)->notificationVariables,
he);}
    if(GC_MarkUsedNode((void**)&((HttpClientWorld *)pointer)->oldNotificationVariables,
he->gcHS_HeapKeeper) == 1)
    {
        markNodeBasedOnType((void**)((HttpClientWorld *)pointer)->oldNotificationVariables,
he);}
    break;
}

void GC_MarkUsedNodes(GC_PointerNode* current, GC_HeapKeeper* heap)
{
    if(current == NULL || heap == NULL)
    {
        return;
    }
#ifdef MARKTRACE
    printf("MARKING_A_NODE_of_type:%d\n", current->pointerType);
#endif

if(GC_MarkUsedNode(current->stackPtr, heap->gcHS_HeapKeeper) == 1)
{
    markNodeBasedOnType(*current->stackPtr, current->pointerType, heap);
}
GC_MarkUsedNodes(current->next, heap);

void GC_Mark(GC_HeapKeeper* gc)
{
    GC_MarkUsedNodes(gc->UsedPointers, gc);
}

int FreeUnusedNodePointers(GC_HeapNode* currentNode)
{
    if(currentNode == NULL)
    {
        return 0;
    }

    if(currentNode->gcNS_nodeAllocated_used == false)
    {
        #ifdef DEALLOCATIONTRACE
            printf("deallocating\n");
        #endif
        free(currentNode->gcNS_nodeAllocated_ptr);
        currentNode->gcNS_nodeAllocated_ptr = NULL;

        return FreeUnusedNodePointers(currentNode->gcS_nodeAllocated_next);
    }

    currentNode->gcNS_nodeAllocated_used = false;
    return FreeUnusedNodePointers(currentNode->gcS_nodeAllocated_next);
}

int Fix_List(GC_HeapNode* previousNode, GC_HeapNode* currentNode, GC_HeapKeeper* gc)
{
    if(currentNode == NULL)
    {
        return 1;
    }
if (currentNode->gcNS_nodeAllocated_ptr == NULL) {
    gc->gcHS_HeapKeeperSize -= currentNode->gcNS_nodeAllocated_size;

    if (previousNode == NULL) {
        gc->gcHS_HeapKeeper = gc->gcHS_HeapKeeper->gcS_nodeAllocated_next;
        free (currentNode);

        return Fix_List(NULL, gc->gcHS_HeapKeeper, gc);
    }

    else {
        previousNode->gcS_nodeAllocated_next = currentNode->gcS_nodeAllocated_next;
        free (currentNode);

        return Fix_List (previousNode, previousNode->gcS_nodeAllocated_next, gc);
    }
}

return Fix_List (currentNode, currentNode->gcS_nodeAllocated_next, gc);

void GC_Sweep (GC_HeapKeeper* gc) {
    Free_Unused_Node_Pointers (gc->gcHS_HeapKeeper);
    Fix_List (NULL, gc->gcHS_HeapKeeper, gc);
}

void GC_Collect (GC_HeapKeeper* gc) {
    GC_Mark (gc);
    GC_Sweep (gc);
}

GC_PointerNode* AppendPointerNode (void** ptr, type pointerType, GC_PointerNode* currentNode, GC_PointerNode* head) {
    if (currentNode == NULL) {
        GC_PointerNode* firstNode = (GC_PointerNode*) malloc (sizeof (GC_PointerNode));
        firstNode->stackPtr = ptr;
}
firstNode->pointerType = pointerType;
firstNode->next = NULL;
    return firstNode;
}

if((currentNode->next == NULL)
{
    GC_PornterNode* newNode = (GC_PornterNode*) malloc(sizeof(GC_PornterNode)) ;
    newNode->stackPtr = ptr;
    newNode->pointerType = pointerType;
    newNode->next = NULL;
    currentNode->next = newNode;
    return head;
}
    return AppendPointerNode(ptr, pointerType, currentNode->next, head);
}

GC_PornterNode* RemovePointerNode(void** ptr,
    GC_PornterNode* previous ,
    GC_PornterNode* currene ,
    GC_PornterNode* start)
{
    if(ptr == NULL)
    {
        #ifdef STACKPOINTERDEALLOCATIONTRACE
        printf(" trying to free null pointer \n");
        #endif
        return start;
    }

    if(current == NULL)
    {
        #ifdef STACKPOINTERDEALLOCATIONTRACE
        printf(" couldn't find pointer in list \n");
        #endif
        return NULL;
    }

    if(current->stackPtr == ptr)
    {
        if(previous == NULL)
        {
            void* temp = current->next;
            free(current);
            return (GC_PornterNode*)temp;
        }
    }
```c
#if define STACKPOINTERDEALLOCATIONTRACE
    printf(" previous is not null\n" );
    printf(" current\nxt: %d\n", current->next );
#endif
void* temp = current->next ;
free( current );
previous->next = ( GC_PointerNode* )temp ;
return start ;
}

return RemovePointerNode( ptr , current , current->next , start );
}

void GC_RegisterPointer( void** ptr , type pointerType , GC_HeapKeeper* gc )
{
    gc->UsedPointers = AppendPointerNode( ptr ,
        pointerType ,
        gc->UsedPointers ,
        gc->UsedPointers );
}

void GC_DeregisterPointer( void** ptr , GC_HeapKeeper* gc )
{
    gc->UsedPointers = RemovePointerNode( ptr ,
        NULL ,
        gc->UsedPointers ,
        gc->UsedPointers );
}
#endif TESTING
int GC_usedPointerCountHelper( GC_PointerNode* head , int count )
{
if ( head == NULL )
{
    return count ;
}

return GC_usedPointerCountHelper( head->next , count + 1 );
}

int GC_usedPointerCounter( GC_PointerNode* head )
{
if ( head == NULL )
{
    return 0 ;
```

100
}  

    return GC_usedPointerCountHelper(head->next, 1); 
}  
#endif
C Maybe monad implemented in C

```c
#include "stdafx.h"

#ifndef MONAD_EXAMPLE
#define MONAD_EXAMPLE

#define TRUE 1
#define FALSE 0

#include <stdio.h>
#include <stdlib.h>

//Type declarations
typedef struct maybeInt {
    int something;
    int just;
} maybeInt;
typedef struct monadStep {
    void const ** args;
    void (*fn)(void** const);
    struct monadStep * next;
} monadStep;

//Constructors
maybeInt Nothing() {
    maybeInt output = { FALSE, 0 };
    return output;
}

maybeInt GetMaybeInt(int i) {
    maybeInt output = { TRUE, i };
    return output;
}

//Monad handling
monadStep * AddToMonad(monadStep * const source,
```
const monadStep * const newStep)
{
    if (((*source).next == NULL)
    {
        (*source).next = newStep;
    }
    else
    {
        AddToMonad((*source).next, newStep);
    }
    return source;
}

monadStep * Bind(void(*f)(const void** const),
const void ** const args,
const monadStep * const monadSource)
{
    monadStep * nextStep = (monadStep *) malloc(sizeof(nextStep));
    (*nextStep).fn = f;
    (*nextStep).args = args;
    (*nextStep).next = NULL;

    if (monadSource == NULL)
    {
        return nextStep;
    }
    else
    {
        return AddToMonad(monadSource, nextStep);
    }
}

// Monad handling

// Monad evaluator
const void** Sequence(const monadStep * const source)
{
    (*source).fn((*source).args);

    if (((*source).next == NULL)
    {
        return source->args;
    }
    else
    {
        return Sequence((*source).next);
    }
}
void Add(const void ** const args)
{
    maybeInt* accumulator = (maybeInt*)args[0];

    if (args == NULL)
    {
        *accumulator = Nothing();
    }

    *accumulator = GetMaybeInt(accumulator->just + (*(int*)args[1]));
}

void Multiply(const void ** const args)
{
    maybeInt* accumulator = (maybeInt*)args[0];

    if (args == NULL)
    {
        *accumulator = Nothing();
    }

    *accumulator = GetMaybeInt(accumulator->just * (*(int*)args[1]));
}

int _tmain(int argc, TCHAR* argv[])
{
    monadStep* start = NULL;
    const void** finalValue;

    //Memory management
    maybeInt* accumulator = (maybeInt*)malloc(sizeof(maybeInt));
    *accumulator = GetMaybeInt(0);

    int arg1 = 1;
    int arg2 = 2;
    int arg3 = 3;
    int arg4 = 4;

    void** call1args = (void**)malloc(2 * sizeof(void*));
    call1args[0] = (void*)accumulator;
    call1args[1] = (void*)&arg1;
}
void** call2args = (void**)malloc(2 * sizeof(void*));
call2args[0] = (void*)accumulator;
call2args[1] = (void*)&arg2;

void** call3args = (void**)malloc(2 * sizeof(void*));
call3args[0] = (void*)accumulator;
call3args[1] = (void*)&arg3;

void** call4args = (void**)malloc(2 * sizeof(void*));
call4args[0] = (void*)accumulator;
call4args[1] = (void*)&arg4;

//Memory management

//Monad creation
start = Bind(Add, call1args, NULL);
start = Bind(Multiply, call2args, start);
start = Bind(Add, call3args, start);
start = Bind(Multiply, call4args, start);

//Monad creation

finalValue = Sequence(start); //Here is where the magic happens

if (((maybeInt*)finalValue[0])->something)
{
    printf("Calculation result: %d", ((maybeInt*)finalValue[0])->just);
}
else
{
    printf("Something went wrong");
}

getchar();
return 0;

#endif
#include "stdafx.h"

#ifndef IOMONADEXAMPLE
#define IOMONADEXAMPLE

#include <stdio.h>
#include <tchar.h>

// Type declarations
typedef struct monadStep
{
  void const ** args;
  void(*fn)(void** const);
  struct monadStep * next;
} monadStep;

// Monad handling
monadStep * AddToMonad(monadStep * const source, const monadStep * const newStep)
{
  if ((*source).next == NULL)
  {
    (*source).next = newStep;
  }
  else
  {
    AddToMonad((*source).next, newStep);
  }
  return source;
}

monadStep * Bind(void(*)(const void** const), const void ** const args, const monadStep * const monadSource)
{
  monadStep * nextStep = (monadStep *)malloc(sizeof(monadStep));

  (*nextStep).fn = f;
  (*nextStep).args = args;
  (*nextStep).next = NULL;

  if (monadSource == NULL)
{ return nextStep;
} else {
    return AddToMonad(mMonadSource, nextStep);
}

// Monad evaluator
const void** Sequence(const monadStep * const source) {
    (*source).fn((*source).args);
    if ((*source).next == NULL) {
        return source->args;
    } else {
        return Sequence((*source).next);
    }
}

void getInput(const void** args) {
    int maxChars = 10;
    const char* message = (char*) malloc((maxChars + 1)* sizeof(char));
    if (gets(message) != NULL) {
        /* Copy over the input from stdin to the output argument that is linked to call 3*/
        memcpy(args[0], message, (maxChars + 1)* sizeof(char));
    }
    free(message);
}

void printToStdOut(const void** const args) {
    printf(args[0]);
}

int _tmain(int argc, _TCHAR* argv[]) {

107
const char * message = "Wadup";
Monad * start = NULL;

//Memory management
void** args1 = (void*)(malloc(sizeof(void*)));
args1[0] = (void*)message;

void** args2 = (void*)(malloc(sizeof(void*)));
args2[0] = (void*)malloc((11) * sizeof(char));

void** args3 = malloc(sizeof(void*));
args3[0] = (void*)args2[0]; //Give information from call 2 to call 3

//Monad creation
start = Bind(printToStdOut, args1, start);
start = Bind(getInput, args2, start);
start = Bind(printToStdOut, args3, start);

Sequence(start); //Here is where the magic happens

getchar();
return 0;
}

#endif
### E. Profiling data for original system

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclic #1</td>
<td>5.791</td>
<td>5.137</td>
<td>230</td>
<td>4693</td>
<td>45.730</td>
<td>51.439</td>
<td>53.700</td>
<td>45.730</td>
<td>51.439</td>
<td>53.700</td>
</tr>
<tr>
<td>Acp10CyclicTC1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KnfeCycl</td>
<td>0.322</td>
<td>230</td>
<td>4694</td>
<td>2.160</td>
<td>3.211</td>
<td>117.370</td>
<td>2.160</td>
<td>3.211</td>
<td>117.370</td>
<td></td>
</tr>
<tr>
<td>Logger</td>
<td>0.089</td>
<td>230</td>
<td>4694</td>
<td>0.900</td>
<td>0.987</td>
<td>2.530</td>
<td>0.900</td>
<td>0.987</td>
<td>2.530</td>
<td></td>
</tr>
<tr>
<td>nodcml</td>
<td>0.213</td>
<td>230</td>
<td>4694</td>
<td>1.860</td>
<td>2.125</td>
<td>3.910</td>
<td>1.860</td>
<td>2.125</td>
<td>3.910</td>
<td></td>
</tr>
<tr>
<td>Cyclic #2</td>
<td>1.795</td>
<td>218</td>
<td>2347</td>
<td>20.460</td>
<td>35.797</td>
<td>265.260</td>
<td>20.460</td>
<td>35.797</td>
<td>265.260</td>
<td></td>
</tr>
<tr>
<td>Main</td>
<td>1.795</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cyclic #3</td>
<td>0.010</td>
<td>200</td>
<td>70</td>
<td>5.460</td>
<td>6.028</td>
<td>6.410</td>
<td>5.460</td>
<td>6.028</td>
<td>6.410</td>
<td></td>
</tr>
<tr>
<td>Acp10NoLinkTask</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cyclic #4</td>
<td>0.039</td>
<td>198</td>
<td>469</td>
<td>2.970</td>
<td>3.859</td>
<td>4.690</td>
<td>2.970</td>
<td>3.859</td>
<td>4.690</td>
<td></td>
</tr>
<tr>
<td>ICCommuic</td>
<td>0.039</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cyclic #5</td>
<td>0.066</td>
<td>196</td>
<td>24</td>
<td>3.880</td>
<td>8.888</td>
<td>10.140</td>
<td>8.888</td>
<td>10.140</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IODController</td>
<td>0.005</td>
<td>196</td>
<td>24</td>
<td>3.880</td>
<td>8.888</td>
<td>10.140</td>
<td>8.888</td>
<td>10.140</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZebraPrint</td>
<td>0.001</td>
<td>196</td>
<td>24</td>
<td>2.796</td>
<td>3.190</td>
<td>3.590</td>
<td>2.796</td>
<td>3.190</td>
<td>3.590</td>
<td></td>
</tr>
<tr>
<td>Cyclic #8</td>
<td>38.754</td>
<td>190</td>
<td>467</td>
<td>4.170</td>
<td>9.090</td>
<td>968.000</td>
<td>4.170</td>
<td>9.090</td>
<td>968.000</td>
<td></td>
</tr>
<tr>
<td>GetStruct</td>
<td>0.091</td>
<td>190</td>
<td>467</td>
<td>2.130</td>
<td>3818.729</td>
<td>6299.360</td>
<td>2.130</td>
<td>3818.729</td>
<td>6299.360</td>
<td></td>
</tr>
<tr>
<td>HttpClient</td>
<td>38.086</td>
<td>190</td>
<td>467</td>
<td>2.560</td>
<td>56.903</td>
<td>21060.410</td>
<td>2.560</td>
<td>56.903</td>
<td>21060.410</td>
<td></td>
</tr>
<tr>
<td>SetStruct</td>
<td>0.568</td>
<td>190</td>
<td>467</td>
<td>2.560</td>
<td>56.903</td>
<td>21060.410</td>
<td>2.560</td>
<td>56.903</td>
<td>21060.410</td>
<td></td>
</tr>
<tr>
<td>Idle Tasks</td>
<td>36.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other System Tasks</td>
<td>17.526</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Calculated Times - Profiler Run Time: 506645.930 [ms]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclic #1</td>
<td>12.944</td>
<td>230</td>
<td>5079</td>
<td>47,010</td>
<td>59,411</td>
<td>128,500</td>
<td>47,010</td>
<td>59,411</td>
<td>128,500</td>
</tr>
<tr>
<td>KiloCyclic</td>
<td>6.467</td>
<td>230</td>
<td>5079</td>
<td>33,470</td>
<td>64,905</td>
<td>291,310</td>
<td>33,470</td>
<td>64,905</td>
<td>291,310</td>
</tr>
<tr>
<td>Longest</td>
<td>0.238</td>
<td>230</td>
<td>5079</td>
<td>2,898</td>
<td>2,898</td>
<td>50,800</td>
<td>0.238</td>
<td>2,898</td>
<td>50,800</td>
</tr>
<tr>
<td>residCtxt</td>
<td>0.033</td>
<td>230</td>
<td>5079</td>
<td>1,820</td>
<td>2,374</td>
<td>42,760</td>
<td>1,820</td>
<td>2,374</td>
<td>42,760</td>
</tr>
<tr>
<td>Cyclic #2</td>
<td>2.002</td>
<td>214</td>
<td>2539</td>
<td>22,050</td>
<td>50,001</td>
<td>330,110</td>
<td>22,050</td>
<td>50,001</td>
<td>330,110</td>
</tr>
<tr>
<td>Main</td>
<td>1.940</td>
<td>214</td>
<td>2539</td>
<td>22,050</td>
<td>50,001</td>
<td>330,110</td>
<td>22,050</td>
<td>50,001</td>
<td>330,110</td>
</tr>
<tr>
<td>Cyclic #3</td>
<td>0.010</td>
<td>200</td>
<td>86</td>
<td>5,650</td>
<td>6,026</td>
<td>6,430</td>
<td>5,650</td>
<td>6,026</td>
<td>6,430</td>
</tr>
<tr>
<td>Acpi10NlLinkTk</td>
<td>0.010</td>
<td>200</td>
<td>86</td>
<td>5,650</td>
<td>6,026</td>
<td>6,430</td>
<td>5,650</td>
<td>6,026</td>
<td>6,430</td>
</tr>
<tr>
<td>Cyclic #4</td>
<td>0.048</td>
<td>196</td>
<td>597</td>
<td>3,180</td>
<td>4,680</td>
<td>60,830</td>
<td>3,350</td>
<td>4,680</td>
<td>60,830</td>
</tr>
<tr>
<td>PCCommunic</td>
<td>0.048</td>
<td>196</td>
<td>597</td>
<td>3,180</td>
<td>4,680</td>
<td>60,830</td>
<td>3,350</td>
<td>4,680</td>
<td>60,830</td>
</tr>
<tr>
<td>Cyclic #5</td>
<td>0.033</td>
<td>196</td>
<td>26</td>
<td>57,540</td>
<td>61,182</td>
<td>100,770</td>
<td>58,540</td>
<td>61,182</td>
<td>100,770</td>
</tr>
<tr>
<td>ICCustRoll</td>
<td>0.033</td>
<td>196</td>
<td>26</td>
<td>2,810</td>
<td>3,003</td>
<td>3,310</td>
<td>2,810</td>
<td>3,003</td>
<td>3,310</td>
</tr>
<tr>
<td>ZadraPrint</td>
<td>0.022</td>
<td>196</td>
<td>26</td>
<td>2,810</td>
<td>3,003</td>
<td>3,310</td>
<td>2,810</td>
<td>3,003</td>
<td>3,310</td>
</tr>
<tr>
<td>Cyclic #8</td>
<td>41,332</td>
<td>190</td>
<td>478</td>
<td>4,120</td>
<td>14,677</td>
<td>2259,340</td>
<td>4,120</td>
<td>15,631</td>
<td>2537,450</td>
</tr>
<tr>
<td>GetStruct</td>
<td>0.190</td>
<td>190</td>
<td>478</td>
<td>4,120</td>
<td>14,677</td>
<td>2259,340</td>
<td>4,120</td>
<td>15,631</td>
<td>2537,450</td>
</tr>
<tr>
<td>HttpClient</td>
<td>41,003</td>
<td>190</td>
<td>478</td>
<td>290,750</td>
<td>4346,014</td>
<td>9009,800</td>
<td>230,750</td>
<td>5014,036</td>
<td>14964,800</td>
</tr>
<tr>
<td>SeeStruct</td>
<td>0.136</td>
<td>190</td>
<td>478</td>
<td>4,120</td>
<td>14,677</td>
<td>2259,340</td>
<td>4,120</td>
<td>15,631</td>
<td>2537,450</td>
</tr>
<tr>
<td>Other System Tasks</td>
<td>43,212</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

F. Profiling data for functional version of the system.
## Stack Usage

**Figure 29: Stack usage for original system**

<table>
<thead>
<tr>
<th>Name</th>
<th>Object Priority</th>
<th>Stack Size [Byte]</th>
<th>Free Stack [Byte]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclic #1</td>
<td>230</td>
<td>8190</td>
<td>6498</td>
</tr>
<tr>
<td>Cyclic #5</td>
<td>196</td>
<td>8190</td>
<td>7790</td>
</tr>
<tr>
<td>Cyclic #8</td>
<td>190</td>
<td>8190</td>
<td>6222</td>
</tr>
</tbody>
</table>

**Figure 30: Stack usage for functional system**

<table>
<thead>
<tr>
<th>Name</th>
<th>Object Priority</th>
<th>Stack Size [Byte]</th>
<th>Free Stack [Byte]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclic #1</td>
<td>230</td>
<td>8190</td>
<td>5382</td>
</tr>
<tr>
<td>Cyclic #5</td>
<td>196</td>
<td>8190</td>
<td>7490</td>
</tr>
<tr>
<td>Cyclic #8</td>
<td>190</td>
<td>16382</td>
<td>8650</td>
</tr>
</tbody>
</table>
H Code metrics for the original version of Knife-Cyclic

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Directory</td>
<td>C:\\\\\\\\\FunctionalPtoProgrammingThesis\Code metrics</td>
</tr>
<tr>
<td>Project Name</td>
<td>originalKnifeCyclic</td>
</tr>
<tr>
<td>Checkpoint Name</td>
<td>Original KnifeCyclic</td>
</tr>
<tr>
<td>Created On</td>
<td>9 Jun 2014, 13:45:08</td>
</tr>
<tr>
<td>Files</td>
<td>10</td>
</tr>
<tr>
<td>Lines</td>
<td>523*</td>
</tr>
<tr>
<td>Statements</td>
<td>319</td>
</tr>
<tr>
<td>Percent Branch Statements</td>
<td>15.7</td>
</tr>
<tr>
<td>Percent Lines with Comments</td>
<td>2.9</td>
</tr>
<tr>
<td>Classes Defined</td>
<td>5</td>
</tr>
<tr>
<td>Methods Implemented per Class</td>
<td>4.60</td>
</tr>
<tr>
<td>Average Statements per Method</td>
<td>5.9</td>
</tr>
<tr>
<td>Line Number of Most Complex Function</td>
<td>(undefined)</td>
</tr>
<tr>
<td>Name of Most Complex Function</td>
<td>KnifeCyclic()</td>
</tr>
<tr>
<td>Maximum Complexity</td>
<td>15</td>
</tr>
<tr>
<td>Line Number of Deepest Block</td>
<td>(undefined)</td>
</tr>
<tr>
<td>Maximum Block Depth</td>
<td>5</td>
</tr>
<tr>
<td>Average Block Depth</td>
<td>1.16</td>
</tr>
<tr>
<td>Average Complexity</td>
<td>2.39</td>
</tr>
<tr>
<td>Functions</td>
<td>8</td>
</tr>
</tbody>
</table>

Functions and Methods in 5 Class(es):

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Complexity, Statements, Max Depth, Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>AxisReader::GetAxisError()</td>
<td>2, 5, 2, 2</td>
</tr>
<tr>
<td>iAxisReader::~iAxisReader()</td>
<td>1, 0, 0, 0</td>
</tr>
<tr>
<td>iKnifeController::~iKnifeController()</td>
<td>1, 0, 0, 0</td>
</tr>
<tr>
<td>KnifeController::KnifeDo()</td>
<td>10, 33, 5, 8</td>
</tr>
<tr>
<td>KnifeCyclic()</td>
<td>15, 25, 4, 14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block Depth</th>
<th>Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>91</td>
</tr>
<tr>
<td>1</td>
<td>139</td>
</tr>
<tr>
<td>2</td>
<td>56</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9+</td>
<td>0</td>
</tr>
</tbody>
</table>
I Code metrics for the functional version of KnifeCyclic

Metrics Summary For Checkpoint 'First draft' (Printed on 30 Jun 2014)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Directory</td>
<td>C:...\FunctionalRcProgrammingThesis\Code metrics</td>
</tr>
<tr>
<td>Project Name</td>
<td>functionalKnifeCyclic</td>
</tr>
<tr>
<td>Checkpoint Name</td>
<td>First draft</td>
</tr>
<tr>
<td>Created On</td>
<td>30 Jun 2014, 15:13:16</td>
</tr>
<tr>
<td>Files</td>
<td>13</td>
</tr>
<tr>
<td>Lines</td>
<td>1,161*</td>
</tr>
<tr>
<td>Statements</td>
<td>631</td>
</tr>
<tr>
<td>Percent Branch Statements</td>
<td>11.9</td>
</tr>
<tr>
<td>Percent Lines with Comments</td>
<td>4.5</td>
</tr>
<tr>
<td>Classes Defined</td>
<td>3</td>
</tr>
<tr>
<td>Methods Implemented per Class</td>
<td>4.33</td>
</tr>
<tr>
<td>Average Statements per Method</td>
<td>2.8</td>
</tr>
<tr>
<td>Line Number of Most Complex Function</td>
<td>(undefined)</td>
</tr>
<tr>
<td>Name of Most Complex Function</td>
<td>markNodeBasedOnType()</td>
</tr>
<tr>
<td>Maximum Complexity</td>
<td>8</td>
</tr>
<tr>
<td>Line Number of Deepest Block</td>
<td>(undefined)</td>
</tr>
<tr>
<td>Maximum Block Depth</td>
<td>4</td>
</tr>
<tr>
<td>Average Block Depth</td>
<td>0.88</td>
</tr>
<tr>
<td>Average Complexity</td>
<td>2.13</td>
</tr>
<tr>
<td>Functions</td>
<td>56</td>
</tr>
</tbody>
</table>

Functions and Methods in 3 Class(es): Complexity, Statements, Max Depth, Calls

<table>
<thead>
<tr>
<th>Function</th>
<th>Complexity, Statements, Max Depth, Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>AxisReader::GetAxisError()</td>
<td>2, 5, 2, 2</td>
</tr>
<tr>
<td>iAxisReader::~iAxisReader()</td>
<td>1, 0, 0, 0</td>
</tr>
<tr>
<td>markNodeBasedOnType()</td>
<td>5, 24, 4, 26</td>
</tr>
<tr>
<td>./VelocityAndAcceleration::GetDistanceTraveledAtUniformedAcceleration()</td>
<td>1, 2, 1</td>
</tr>
</tbody>
</table>

Block Depth

<table>
<thead>
<tr>
<th>Block Depth</th>
<th>Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>243</td>
</tr>
<tr>
<td>1</td>
<td>254</td>
</tr>
<tr>
<td>2</td>
<td>106</td>
</tr>
<tr>
<td>3</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9+</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 32: Code metrics for the functional version of KnifeCyclic
Code metrics of functional KnifeCyclic without libraries

Metrics Summary For Checkpoint 'Checkpoint2' (Printed on 02 sep 2014)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Directory</td>
<td>C:\FunctionalPlcProgrammingThesis\Code metrics</td>
</tr>
<tr>
<td>Project Name</td>
<td>functionalKnifeCyclic</td>
</tr>
<tr>
<td>Checkpoint Name</td>
<td>Checkpoint2</td>
</tr>
<tr>
<td>Created On</td>
<td>2 Sep 2014, 15:25:07</td>
</tr>
<tr>
<td>Files</td>
<td>9</td>
</tr>
<tr>
<td>Lines</td>
<td>590*</td>
</tr>
<tr>
<td>Statements</td>
<td>313</td>
</tr>
<tr>
<td>Percent Branch Statements</td>
<td>4.8</td>
</tr>
<tr>
<td>Percent Lines with Comments</td>
<td>0.8</td>
</tr>
<tr>
<td>Classes Defined</td>
<td>3</td>
</tr>
<tr>
<td>Methods Implemented per Class</td>
<td>4.33</td>
</tr>
<tr>
<td>Average Statements per Method</td>
<td>2.8</td>
</tr>
<tr>
<td>Line Number of Most Complex Function</td>
<td>(undefined)</td>
</tr>
<tr>
<td>Name of Most Complex Function</td>
<td>KnifeCyclic()</td>
</tr>
<tr>
<td>Maximum Complexity</td>
<td>5</td>
</tr>
<tr>
<td>Line Number of Deepest Block</td>
<td>(undefined)</td>
</tr>
<tr>
<td>Maximum Block Depth</td>
<td>3</td>
</tr>
<tr>
<td>Average Block Depth</td>
<td>0.55</td>
</tr>
<tr>
<td>Average Complexity</td>
<td>1.52</td>
</tr>
<tr>
<td>Functions</td>
<td>20</td>
</tr>
</tbody>
</table>

Functions and Methods in 3 Class(es): Complexity, Statements, Max Depth, Calls

- AxisReader::GetAxisError(): 2, 5, 2, 2
- IAxisReader::~IAxisReader(): 1, 0, 0, 0
- KnifeCyclic(): 5, 11, 3, 12
- VelocityAndAcceleration::GetDistanceTraveledAtUniformedAcceleration(): 1, 2, 1

Block Depth Statements

<table>
<thead>
<tr>
<th>Block Depth</th>
<th>Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>166</td>
</tr>
<tr>
<td>1</td>
<td>123</td>
</tr>
<tr>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9+</td>
<td>0</td>
</tr>
</tbody>
</table>

114
K  Code metrics for original IOController

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Directory</td>
<td>C:\ FunctionalPlcProgrammingThesis\Code metrics</td>
</tr>
<tr>
<td>Project Name</td>
<td>originalIOController</td>
</tr>
<tr>
<td>Checkpoint Name</td>
<td>Original IO Controller</td>
</tr>
<tr>
<td>Created On</td>
<td>9 Jun 2014, 13:44:23</td>
</tr>
<tr>
<td>Files</td>
<td>8</td>
</tr>
<tr>
<td>Lines</td>
<td>378*</td>
</tr>
<tr>
<td>Statements</td>
<td>228</td>
</tr>
<tr>
<td>Percent Branch Statements</td>
<td>15.8</td>
</tr>
<tr>
<td>Percent Lines with Comments</td>
<td>4.0</td>
</tr>
<tr>
<td>Classes Defined</td>
<td>4</td>
</tr>
<tr>
<td>Methods Implemented per Class</td>
<td>4.25</td>
</tr>
<tr>
<td>Average Statements per Method</td>
<td>6.1</td>
</tr>
<tr>
<td>Line Number of Most Complex Method</td>
<td>(undefined)</td>
</tr>
<tr>
<td>Name of Most Complex Method</td>
<td>LED::ManageLight()</td>
</tr>
<tr>
<td>Maximum Complexity</td>
<td>9</td>
</tr>
<tr>
<td>Line Number of Deepest Block</td>
<td>(undefined)</td>
</tr>
<tr>
<td>Maximum Block Depth</td>
<td>3</td>
</tr>
<tr>
<td>Average Block Depth</td>
<td>1.06</td>
</tr>
<tr>
<td>Average Complexity</td>
<td>3.57</td>
</tr>
<tr>
<td>Functions</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions and Methods in 4 Class(es):</th>
<th>Complexity, Statements, Max Depth, Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>HMIManager::DoPackStationLightsControl()</td>
<td>21, 3, 12</td>
</tr>
<tr>
<td>IHMIManager::~IHMIManager()</td>
<td>1, 0, 0, 0</td>
</tr>
<tr>
<td>ILED::~ILED()</td>
<td>1, 0, 0, 0</td>
</tr>
<tr>
<td>IOControllerCyclic()</td>
<td>9, 15, 3, 4</td>
</tr>
<tr>
<td>LED::ManageLight()</td>
<td>9, 6, 2, 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block Depth</th>
<th>Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>60</td>
</tr>
<tr>
<td>1</td>
<td>109</td>
</tr>
<tr>
<td>2</td>
<td>44</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9+</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 33: Code metrics for the original version of IOController
### Code metrics for functional IOController

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Directory</td>
<td>C:\..\..\..\..\..FunctionalPlcProgrammingThesis\Code metrics</td>
</tr>
<tr>
<td>Project Name</td>
<td>functionalIOController</td>
</tr>
<tr>
<td>Checkpoint Name</td>
<td>First draft</td>
</tr>
<tr>
<td>Created On</td>
<td>30 Jun 2014, 15:12:44</td>
</tr>
<tr>
<td>Files</td>
<td>7</td>
</tr>
<tr>
<td>Lines</td>
<td>983*</td>
</tr>
<tr>
<td>Percent Branch Statements</td>
<td>10,4</td>
</tr>
<tr>
<td>Percent Lines with Comments</td>
<td>3,8</td>
</tr>
<tr>
<td>Classes Defined</td>
<td>0</td>
</tr>
<tr>
<td>Methods Implemented per Class</td>
<td>0,00</td>
</tr>
<tr>
<td>Average Statements per Method</td>
<td>0,0</td>
</tr>
<tr>
<td>Line Number of Most Complex Function</td>
<td>{undefined}</td>
</tr>
<tr>
<td>Name of Most Complex Function</td>
<td>markNodeBasedOnType()</td>
</tr>
<tr>
<td>Maximum Complexity</td>
<td>8</td>
</tr>
<tr>
<td>Line Number of Deepest Block</td>
<td>{undefined}</td>
</tr>
<tr>
<td>Maximum Block Depth</td>
<td>4</td>
</tr>
<tr>
<td>Average Block Depth</td>
<td>0,91</td>
</tr>
<tr>
<td>Average Complexity</td>
<td>2,20</td>
</tr>
<tr>
<td>Functions</td>
<td>51</td>
</tr>
<tr>
<td>Functions:</td>
<td>Complexity, Statements, Max Depth, Calls</td>
</tr>
<tr>
<td>markNodeBasedOnType()</td>
<td>8, 24, 4, 26</td>
</tr>
</tbody>
</table>

### Block Depth

| 0        | 192 |
| 1        | 240 |
| 2        | 88  |
| 3        | 21  |
| 4        | 5   |
| 5        | 0   |
| 6        | 0   |
| 7        | 0   |
| 8        | 0   |
| 9+       | 0   |

---

Figure 34: Code metrics for the functional version of IOController
M  Code metrics of functional IOController without libraries

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Directory</td>
<td>\path{C:...\\\\\:\FunctionalPlcProgrammingThesis\Code metrics}</td>
</tr>
<tr>
<td>Project Name</td>
<td>functionalIOController</td>
</tr>
<tr>
<td>Checkpoint Name</td>
<td>Checkpoint9</td>
</tr>
<tr>
<td>Created On</td>
<td>2 Sep 2014, 15:26:30</td>
</tr>
<tr>
<td>Files</td>
<td>5</td>
</tr>
<tr>
<td>Lines</td>
<td>550*</td>
</tr>
<tr>
<td>Statements</td>
<td>299</td>
</tr>
<tr>
<td>Percent Branch Statements</td>
<td>7.4</td>
</tr>
<tr>
<td>Percent Lines with Comments</td>
<td>3.1</td>
</tr>
<tr>
<td>Classes Defined</td>
<td>0</td>
</tr>
<tr>
<td>Methods Implemented per Class</td>
<td>0.00</td>
</tr>
<tr>
<td>Average Statements per Method</td>
<td>0.0</td>
</tr>
<tr>
<td>Line Number of Most Complex Function</td>
<td>{undefined}</td>
</tr>
<tr>
<td>Name of Most Complex Function</td>
<td>prepareWhiteLightControl()</td>
</tr>
<tr>
<td>Maximum Complexity</td>
<td>5</td>
</tr>
<tr>
<td>Line Number of Deepest Block</td>
<td>{undefined}</td>
</tr>
<tr>
<td>Maximum Block Depth</td>
<td>3</td>
</tr>
<tr>
<td>Average Block Depth</td>
<td>0.61</td>
</tr>
<tr>
<td>Average Complexity</td>
<td>1.89</td>
</tr>
<tr>
<td>Functions</td>
<td>27</td>
</tr>
</tbody>
</table>

Functions: Complexity, Statements, Max Depth, Calls

| prepareWhiteLightControl()       | 5, 7, 2, 0 |

<table>
<thead>
<tr>
<th>Block Depth</th>
<th>Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>148</td>
</tr>
<tr>
<td>1</td>
<td>124</td>
</tr>
<tr>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9+</td>
<td>0</td>
</tr>
</tbody>
</table>
## N Code metrics for HttpClientCyclic

Metrics Summary For Checkpoint 'First draft' (Printed on 30 jun 2014)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
</table>
| Project Directory                | C:\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\n
Figure 35: Code metrics for the functional version of HttpClientCyclic
Code metrics of functional HttpClient without libraries

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Directory</td>
<td>C:\FunctionalPlcProgrammingThesis\Code metrics</td>
</tr>
<tr>
<td>Project Name</td>
<td>functionalHttpClient</td>
</tr>
<tr>
<td>Checkpoint Name</td>
<td>Checkpoint2</td>
</tr>
<tr>
<td>Created On</td>
<td>2 Sep 2014, 15:27:07</td>
</tr>
<tr>
<td>Files</td>
<td>5</td>
</tr>
<tr>
<td>Lines</td>
<td>461</td>
</tr>
<tr>
<td>Statements</td>
<td>238</td>
</tr>
<tr>
<td>Percent Branch Statements</td>
<td>11,8</td>
</tr>
<tr>
<td>Percent Lines with Comments</td>
<td>0,4</td>
</tr>
<tr>
<td>Classes Defined</td>
<td>0</td>
</tr>
<tr>
<td>Methods Implemented per Class</td>
<td>0,00</td>
</tr>
<tr>
<td>Average Statements per Method</td>
<td>0,0</td>
</tr>
<tr>
<td>Line Number of Most Complex Function</td>
<td>{undefined}</td>
</tr>
<tr>
<td>Name of Most Complex Function</td>
<td>HttpClientCyclic()</td>
</tr>
<tr>
<td>Maximum Complexity</td>
<td>10</td>
</tr>
<tr>
<td>Line Number of Deepest Block</td>
<td>{undefined}</td>
</tr>
<tr>
<td>Maximum Block Depth</td>
<td>4</td>
</tr>
<tr>
<td>Average Block Depth</td>
<td>0,64</td>
</tr>
<tr>
<td>Average Complexity</td>
<td>2,87</td>
</tr>
<tr>
<td>Functions</td>
<td>15</td>
</tr>
</tbody>
</table>

Functions: Complexity, Statements, Max Depth, Calls

HttpClientCyclic() 10, 19, 4, 12

Block Depth

<table>
<thead>
<tr>
<th>Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9+</td>
</tr>
</tbody>
</table>

119