DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2021

A Method for Porting Software Using Formal Specifications

MAGNUS FREDRIKSSON

FREDRIK ÖBERG

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Abstract

Formal specifications are mathematically based techniques with which a system can be analyzed, and its functionalities be described. Case studies have shown that using formal specifications can help reduce bugs and other inconsistencies when implementing a complex system; they are more likely found during the software design phase rather than later.

During the process of porting code, testing has been used to verify that the port has the same functionalities as the original. However, testing alone has been deemed necessary but not sufficient to accomplish this. This thesis questions if formal specifications could be used during the process of porting code to create an accurate model of the system, and thereby provide higher degrees of certainty that the final product conforms to the original.

A step-by-step methodology is presented to answer this question. The methodology ascertains the behavior of a port target through testing and a formal specification model based on these tests is created. This model is then used to create the port. The result indicates that the methodology used has some potential since it provided a high level of certainty that the ported code adhered to the original. Since the methodology puts a high emphasis on the specification and has several layers of verification, it is likely that it is suitable for projects with several modules and interdependencies.

When using it for porting a trivial or non-complex system, the overhead of the methodology may prove high in comparison to the value gained. It was also found that one must take into consideration the implicit functionalities a language provides. Strict reliance on a model could thereby lead to a less flexible process where creativity and consideration of the specifics of the target language may have produced a different result.

Keywords: porting, formal specification, TLA+, testing, Rust

Sammanfattning

Formella specifikationer är matematiska tekniker med vilka ett system kan analyseras och dess funktioner beskrivas. Fallstudier har visat att användning av formella specifikationer vid mjukvaruutveckling kan bidra till att reducera antalet fel och därmed minska omkostnaderna.

Vid portering av kod har tester använts för att verifiera att porteringen har samma funktionaliteter som originalet. Tester anses dock vara otillräckliga - om än nödvändiga - för att fullständigt beskriva ett system. Denna avhandling ställer frågan om formella specifikationer kan användas i en porteringsprocess och därmed ge en högre grad av tillförlitlighet i att porteringen överensstämmer med originalet.

En steg-för-steg-metodik presenteras för att besvara denna fråga. Metodiken använder sig av tester för att beskriva det system som skall porteras och en modell baserad på formella specifikationer skapas genom dessa tester. Denna modell används därefter för att skapa porteringen. Resultatet indikerar att metoden har viss potential eftersom den upplevdes öka förståelsen för referensarbetet och därmed också tryggheten i att den portade koden överensstämde med originalet. Eftersom metodiken lägger stor vikt vid specifikationen och består av flera lager av verifiering är det troligt att den är mer lämplig för system som består av flertalet moduler med inbördes beroende.

Vid användning för att portera ett trivialt system kan metodens omkostnader visa sig vara för höga i förhållande till det värde som erhålls. Det visade sig även att en porteringsprocess måste ta hänsyn till de implicita funktioner ett språk tillhandahåller. En strikt anpassning till en modell kan därmed leda till en mindre flexibel process och därmed leda till ett icke tillfredsställande resultat, där kreativitet och hänsyn till målspråket skulle kunna ha gett ett annat.

Nyckelord: portering, formell specifikation, TLA+, testning, Rust

Acknowledgments

We would like to thank Rasmus Källqvist, Hind Kareem and Johan Montelius for their valuable input and continuous support during this project.

Stockholm, June 2021
Magnus Fredriksson and Fredrik Öberg

Contents

1 Introduction

The computer software world is under constant development. New demands on a system could make it necessary to adapt it into a new environment than what it was originally designed for - a process called porting.

1.1 Background

When porting software, one of the main concerns is that the functionality of the port adheres to the functionalities of the target source. The traditional way of verifying the behavior of computer software has been through the practice of testing. However, testing has limitations, and it is difficult to fully describe a system using testing alone [1]. One way to complement the testing process to improve on this situation could be to set up a model of the system through a formal specification; a mathematically based technique with which the design of a system can be analyzed, and its functionalities be described with the purpose of helping with the implementation of systems and software. This means that independent of what language a software has been written in, if its behavior follows the formal specification, it provides functionality equivalent to any other software conforming to that same specification.

1.2 Commissioned Work

The customer for this project is Young Aces By Sylog (YABS); an IT consultant firm with one of its offices based in Stockholm, Sweden. As consultants they get hired to do work not normally part of their customers daily operations. A lot of this work concerns updating and porting system functionalities consisting of so-called legacy code. This is normally outdated code often severely limiting the performance of the system in question.

1.3 Problem Specification

The expressed problem that makes up the basis for this thesis is that the methods commonly used today are insufficient to verify the outcome of ported functionalities reliably. This thesis aims to answer the question:

is there a way to complement the current porting methods, which could increase the certainty that the ported code adheres to the original using a formal specification?

1.4 Purpose

The use of formal specifications has indicated that it reduces the number of errors during software development. This practice has shown to create more reliable software while at the same time reducing production costs [2]. The purpose of this thesis is to evaluate if formal specifications have the potential to provide a higher level of certainty that ported code adheres to the original.

1.5 Goal

The goal of this thesis is to create a step-by-step methodology for porting software which uses a formal specification. Principally, to find whether the approach taken to incorporate a formal specification into the porting process yielded a valid result. A subjective evaluation of the strengths and weaknesses found when applying the methodology to the sample project will be made.

1.6 Target Audience

The intended target audience is mainly practicing computer engineers as well as students of computer science interested in formal specifications and how it could be used in practice. It could also have the potential to be used as a basis for further work in the academic world.

1.7 Research Methodology

The thesis will be performed as a qualitative case study where a detailed model for the methodology is described and then applied to a sample case. A method for evaluating the results will be described as part of the methodology and will be used to validate whether the process was successful. The quality of work and results of this thesis will be evaluated according to a set of quality criteria for qualitative research [3].

1.8 Delimitations

The investigation's primary focus is on the feasibility of using formal specifications during a porting process; if it has the potential to be used to create a valid port of a system in a practical way by computer engineers during porting of computer software. The thesis will evaluate the validity of the port considering that the functionality provided is correct, but not whether it is achieved following any constraints on its performance.

The project is undertaken in a restricted time frame. Therefore, the scope of the source to which the methodology is applied is limited. This means that the efficiency of the method on other types of projects is left up to further investigation.

2 Background

This chapter provides context to the information discussed throughout the thesis and is accomplished by providing theories, concepts, terms as well as historical data from relevant studies.

Verifying Programs Using Tests describes two testing methods and some of their limitations as well as what a test-driven development (TDD) is. *Verifying Programs Using Tests* also has a description of the concept of property-based testing. *Formal Specification* provides a description of what the purpose of the technique is and how it has been used in the field of software development. It also contains a description of TLA+ which is the formal specification language used during this project. The sections *C*, *Rust* and *Hoare Logic* provides background information about those topics and *FreeBSD Linked List* describes the porting target source code and why it was chosen.

2.1 Verifying Programs Using Tests

Testing is, in general, performed in one of two ways. The first being where the functionality of the system is tested from an outside perspective without peering into its internal structure. This verifies that the expected outputs are produced on a given set of inputs and is a method usually called black-box, or functional testing. The second way of testing is where the internal structure and design of the program - its state - is being tested and is usually called white-box, or structural testing [4]. It is not feasible through testing alone, by either of these disciplines, to guarantee the absence of bugs. It is only possible to verify that a piece of code behaves in some way given a specific, finite, set of inputs. Consider, for example, the pseudo-code in figure 1.

```
global signed z
function x(y):
   z++
   y
```

Figure 1. Pseudo-code Function

The function takes an argument as input and returns it without any mutation. It also applies the increment operator ++ on the global variable *z*. As *z* is signed, there are specific values of *z* where the increment by one operator is not defined

- namely the highest value that z can take. From a black-box perspective, it could be concluded from even somewhat extensive testing, that the function behaves in a deterministic way and the rule described in figure 2 would hold.

```
[for all] numbers y
x.apply(y) is y
```

Figure 2. Pseudo-code Property

Achieving total coverage, even in this trivial example, would be time consuming since the input domain is all possible machine integers. It could also be the case that testing all of them would not necessarily find the bug since a possible undefined behavior of z is independent of the functions input values. There are similar examples where the function would behave as intended when called in a certain application state, but potentially misbehave when the state is different, even if the input is the same each time. In the case of a more complex function which takes more input arguments making a linear search of the input space unfeasibly time-consuming. It could also be the case that even with many samples, the non-deterministic behavior of the function may not necessarily be exposed. Using metrics such as branches taken, or lines executed [4] - common metrics for coverage - would not help in this case. Neither would automatically generate sets of input to be tested, so called property-based testing [5], necessarily find the problem since many consecutive calls to the function are needed (see section 2.1.1). The goal of testing thereby should be to find a reasonable set of tests that can verify the behavior for most input. They should also exercise the program's structural components thoroughly enough, no matter what is the correct, or intended behavior [4].

By considering the structure of the program, it is possible to at least avoid constructing a set of tests which are guaranteed to be insufficient. The reasoning behind this is that it most certainly cannot be made any inference about the effects of a line of code if the line is never executed. Instead of doing an exhaustive search of the input space, structural tests give as good a guarantee that the code is covered, but with fewer test-cases. This is accomplished by basing the tests on the control structure of the program [4] - in other words white-box testing. Tests written using the program's structural properties as a basis for test completeness should therefore be able to ascertain the behavior of the program to a degree sufficient for a starting point from which a mathematical description, or formal specification, can be created.

2.1.1 Property-based Testing

Property-based testing is a way of using test-cases, not to check specific concrete cases, but rather as rules which describe some property of a function. The principle of property testing is described in *QuickCheck* by K. Claessen and J. Hughes [6]. It entails that the test code describes some operation that is permissible unto the system under test along with the constraints on the input data for that operation. Along with the operation, some *properties* such as pre- and post-conditions that must always hold true are defined. Usually this is done by the property function returning either true or false, or by using assertions. To then exercise this property, a set of concrete test-cases are automatically generated from this blue-print. The input data is usually generated according to some heuristic that is likely to find edge cases, e.g., "empty" style inputs, or inputs close to the perimeters of the range [5]. This allows for a huge breadth of input parameters to be tested with very little effort.

There exists a subset of property-based testing, herein called stateful property-based testing. The purpose of this type of testing is to verify that properties of the system hold true, even over the lifetime of the system. To verify this, a model is defined. The model will usually be a simplified - but correct - version of the system under test (SUT). The model is used, before or after each operation, to verify some fact about the state of the SUT. For each operation on the SUT, a corresponding operation on the model is defined along with predicates that determine when the operation is valid. These operations can then be applied consecutively in different orders to find inconsistencies between the model and the SUT. Some testing frameworks will as an addition to finding the sequences of operations which yield inconsistencies, have some heuristic with which these sequences can be reduced to shorter ones [6].

2.1.2 Test Driven Development

TDD is a common software development practice where tests are written first, and code is written only to satisfy these tests which initially should be failing [7]. This leads to an incremental development process where development and testing is interleaved. One benefit to this process mentioned in [8] is that it gives clarity to the intentions of the author - the tests make assumptions about how the code should behave explicitly. It is argued that because of this, tests make the code self-documenting. The fundamental process of TDD is an incremental, iterative one where implementation and refactoring is repeated until the tests pass [8] (see figure 3).

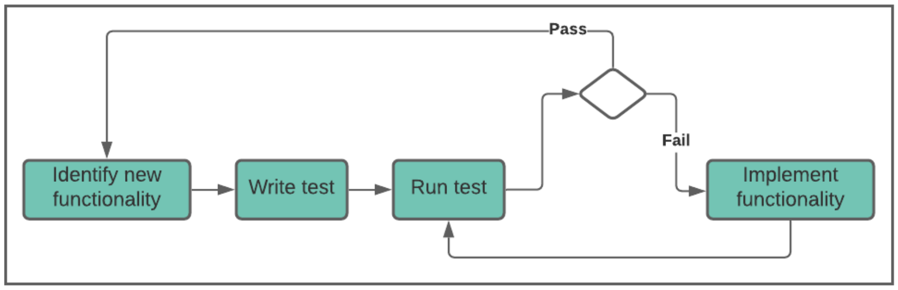


Figure 3. The TDD Process

2.2 Hoare Logic

Hoare logic is a formal system with a set of axioms and logical rules used in proofs of properties of computer programs [9]. Its central feature is the Hoare triple (see figure 4).

$$P\{Q\}R$$

Figure 4. Hoare Triple

This triple describes how the execution of a piece of code changes the state of the computation and should be interpreted as “if the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion”. The use of this deductive reasoning is intended to provide a fundamental understanding of the code.

2.3 Formal Specification

Building a software system is almost entirely a design activity consisting of combining, inventing, and planning the implementation of abstractions. The goal with this is to describe a set of modules interacting with one another in simple, well defined ways [10]. If this goal is achieved it will enable engineers to work on different modules independently, while still accomplishing their common purpose. A good design will also make maintaining the software, as well as modifying its modules, easier without causing unintended effects. In summation, the key to good software design is inventing appropriate abstractions around which to structure the system.

Formal specifications are mathematically based techniques used to help with designing systems and software. Formal in this case means that they have a specific syntax, the domain of the semantics has a distinct set of meanings, and they can be used to infer useful information [11]. This enables them to demonstrate that a system is correct with respect to its specification while simultaneously describing system abstractions. This is accomplished independent of any of its implementations - specifications describe what a system should do, not how it should do it. This enables designers to focus attention on possible inconsistencies, deficiencies, and ambiguities leading to many mistakes and subtle bugs from many sources cropping up in specifications before they do in the implementation. This benefit could be derived from the fact that engineers focus on the safety and liveness properties of a system when using formal specifications; they need to state what needs to go right instead of focusing on what could go wrong, often a mind set when writing tests [2]. Formal specifications encourage this way of thinking by describing an algorithm as a state machine, enabling a designer to create a simple and powerful abstraction of it.

Since computer systems become increasingly more powerful and complex as time passes, the need for better techniques to assist in the design and implementation of reliable software becomes more prominent. Formal specifications have been around since the early days of computers and have been adopted in more traditional engineering disciplines but are not widely used in industrial software development [8]. One of the reasons for this is that the method, by many, is not considered to be cost-effective. There has also been very little interaction between the test and formal specification communities, but some argue that the approaches could be used as complementary to each other for a better result [12].

Recent studies have also found that creating a mathematical model of a system under development led them to find and handle undesired behavior during the system analysis phase [13] [2]. These studies claim that the time spent making mathematical models was well invested since it reduced the amount of time spent finding and fixing bugs during the implementation phase. This is most likely the case, since a model with good system-invariants helps engineers get the design right from the outset. The invariants capture the fundamental reason why the system works by using them as safety properties; properties that need to hold true for each possible state that the system can take. Therefore, they show that the design is not broken, which helps engineers get the code right faster since a broken design will lead to broken software - even if the code passes tests based on the design.

Formal specifications due to its mathematical nature, however, are not an all-purpose tool. The consensus seems to be that they are most likely to be useful in data modeling where data definitions are written in a common, implementation-independent manner which lets the information pass through various development phases without transformation This means that formal specifications are considered useful in situations such as when [13]:

- There is a complex data structure that must be handled correctly.
- A precise function definition is required when a simple function is needed, but it is vital that the function is implemented correctly.
- Complex functionality is involved with many choices to be made or many exceptional conditions arise.

To know when and how to use a formal specification takes a lot of skill and experience. For example, a formal specification does not easily model the performance of a system, and some aspects of performance cannot be modelled at all [2]. If applied without consideration, a formal specification could therefore lead to a sub-optimal design even in the absence of errors. A feasible way to model a system to predict the issue of prolonged severe slowdowns seems to not be known as of this day [2]. This means that other techniques need to be used combined with formal specifications to mitigate that.

2.3.1 TLA+

TLA+ is a language developed by the computer scientist Leslie Lamport [14] and has been used by, for example, Amazon since 2011 as a tool to achieve correctness in sophisticated distributed systems [15]. They evaluated TLA+ among other languages like Alloy [16] and Microsoft VCC [17]. All languages had their pros and cons but TLA+ turned out to be best suited for their needs.

The evaluation found that TLA+ was best for handling very large, complex, as well as subtle problems. TLA+ was claimed to accomplish this by being able to capture rich concepts simply and directly without tedious workarounds, for example when specifying dynamic sequences of many types of nested records. It was also perceived that details from complex designs can be added or removed quickly since TLA+ supports arbitrarily complicated data structures with the ability to define powerful custom operators. This means that it was easy to adjust to a suitable level of abstraction making it a good tool when diagnosing bugs and subtle errors. Another property of TLA+ highlighted was that it tended to minimize the cognitive burden on engineers and accomplishes this by using conventional terminology since it largely consists of standard discrete math and a subset of linear temporal logic. This means that TLA+ allows for the expression of temporal relations between predicates, for example that some predicate will evaluate true *eventually*, or that some predicate may be true only when another evaluates to some specific value.

TLA+ Example

A trivial example of how TLA+ can be used can be seen in this section, and is inspired by the works of Leslie Lamport [18].

```
global signed z
main()
   z = someNumber()
   z = z + 1
```

Figure 5. Program Example

The example program (see figure 5) has an entry point *main* and a global variable z which gets assigned an integer through a call to the function *someNumber*; a function returning said integer and has no side effects. The value of z then gets incremented by one followed by the program terminating. This process can be described with a state machine (see figure 6) which creates a simple abstraction of the code.

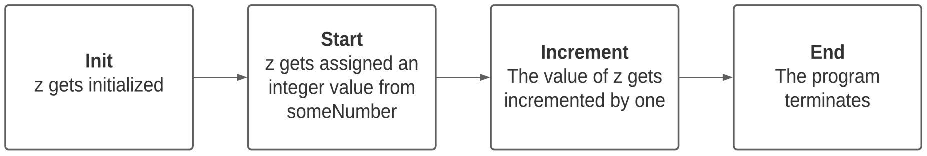

Init
z gets initialized
Start
z gets assigned an integer value from someNumber
Increment
The value of z gets incremented by one
End
The program terminates

Figure 6. State Machine

The state machine can then be implemented in TLA+ (see figure 7). Here there is an additional variable defined called *pc* which is short for program counter. It is used by the TLA+ model checker to know which state is the current one. The *Init* label indicates the first state and initializes the value of the variables of the model, so the model checker knows where to begin. That the value of z is initialized to 0 is a behavior of the language C and is used as an example but could differ depending on the language used when implementing the function.

After the initializations, the model checker moves on to the label named *Next* and checks which of the disjunctions (indicated by the or operator) holds true for every conjunction - in this case where the value of *pc* is "start". *z* then gets assigned its new value, indicated by the prime attached to it. It is accomplished by using the operator \in which is equivalent to the mathematical operator $\in$ - member of. The function *someNumber* is supposed to return any integer but TLA+ only allows for usage of finite sets. Therefore, as a compromise, *z* is assigned an integer between 1 and 1000 indicated by the set operator ".." between those numbers. This means that *z* becomes a member of the set of integers between one and one thousand in the current state. Lastly in this disjunction, *pc* gets assigned its new value "increment". This means that the model checker moves to the state which holds true for the new value, checks it and proceeds with this practice until *pc* is assigned the value "done" followed by its termination. This enables the model to be evaluated to see if every statement holds true given any situation, providing a logical foundation that the implementation does the same. If, however, it is not possible to satisfy one or more of the conjunctions, the model must be invalid, and cannot be implemented as an executable program.

```
VARIABLES z, pc

Init ==
    pc = "start" /\ z = 0

Next ==
    \/ /\ pc = "start"
       /\ z' \in 1..1000
       /\ pc' = "increment"
    \/ /\ pc = "increment"
       /\ z' = z + 1
       /\ pc' = "done"
```

Figure 7. TLA+ Example

TLA+ Tools

The TLA+ ecosystem provides an integrated development environment (IDE) called TLA+ Toolbox [19], which allows for writing and translating the pseudocode like language PlusCal into pure TLA+. PlusCal is a language that can express much of what TLA+ can express, but more conveniently. PlusCal even extends TLA+ with additional syntax and can be especially suitable for implementing algorithms [20].

2.4 C

Since it started to gain popularity during the 1980s has the programming language C been a natural choice when programming at a low-level. Although C provides high level abstractions, nowadays C is considered a low level language due to the lack of automatic memory management and that it provides transparent, efficient access to the underlying hardware [21]. C found lasting use in applications previously coded in assembly. C provides many features not available in assembly such as several levels of scoping. This makes it possible to use the same identifier many times within a program without them necessarily interfering with each other. C also provides static type checking, meaning that type checking is performed at compile time. There are no strict rules about whether a language is strongly or weakly typed, however, since C allows the conversion of a void pointer to any other type and vice versa, it should not be considered strongly typed, Kernighan and Ritchie also consider it as such [22].

One problem with C , especially nowadays when the alternative is not assembly language, but instead other modern programming languages, is the manual management of memory and the risks that come with this – a risk many would claim makes C an unsafe language [23]. This comes from the fact that a misuse of manual memory management will cause undefined behavior. Bugs relating to mismanagement of memory can also be very hard to track down since such errors may lie dormant in the program until later when the program suddenly crashes or misbehaves. The program may also become unpredictable and create security vulnerabilities, for example when accessing some memory after the resource has already been freed (use-after-free) [24]. These problems among others have prompted the development of so-called safe languages [25], many of which achieve safety through moving memory management into a layer not directly accessible to the programmer. This extra layer of virtualization - like most things - has an impact on the performance of the program often caused in large part by the garbage collector which must keep track of all allocated objects in real-time and reclaim them when they are no longer needed [26].

2.5 Rust

Rust is a modern system programming language focusing on safety, speed, and concurrency and claims to accomplish these goals without using a garbage collector. Instead, the safety is enforced by the compiler by checking under what conditions objects in memory are being accessed or mutated according to some rules about data ownership. The rules regarding aliasing in Rust are quite strict, where aliasing means when two or more variables or pointers refer to the same or overlapping regions in memory. One of the most essential rules of Rust, regarding aliasing can be summarized as follows [27]:

There may be one mutable ***or*** *N number of immutable references to an object, but never* ***both*** *at the same time (N here could be any natural number).*

This is enforced statically by Rust's static analyzer called the borrow checker. The concept of mutability is also touched upon here, and it is important to understand that every variable or reference in Rust carries with it metadata about its mutability i.e. whether it can be modified. For example, if a stack-allocated variable like an integer is mutable, the programmer is allowed to reassign it with a new value. For references, mutability determines whether it is allowed to call the referenced object's mutable methods, change its properties directly, or pass it to a function that requires a mutable reference.

By enforcing the rules at compile-time, rather than at run-time, code written in Rust can be as fast as code written in C. Designers of Rust also claim that introducing parallelism in Rust comes at a relatively low risk, since the compiler is designed to catch classical mistakes - such as race conditions - before execution [23]. Code written in Rust is also claimed to always provide type- and memory-safety and situations like dangling pointers, use after free or any other kind of undefined behavior should never have to be endured by the programmer. This allows for more aggressive optimizations since they will not accidentally introduce crashes or vulnerabilities. The purpose of Rust is to eliminate the trade-offs that have been accepted when working with C while at the same time getting comparable performance [27].

Rust can be thought of as a combination of two programming languages - safe and unsafe Rust. Safe Rust is a subset of Rust and often considered the *true* Rust programming language. This is where all the compile-time memory and type-safety checks are applied. Sometimes, however, there will be situations where low-level memory interaction is needed. In unsafe Rust, the programmer is allowed to dereference raw pointers and use other types of operations for which the compiler cannot provide the safety guarantees present in safe Rust [27]. The value of this separation is that the programmer gains the benefits of having the option to use the benefits of an unsafe language — low level control over implementation details — without most of the problems that come with trying to integrate it with a completely different safe language [23]. Another benefit to this separation is that the developer limits the number of lines of code that need to be debugged, whenever such a bug appears.

2.6 FreeBSD Linked List

FreeBSD is a Unix-like operating system used to power modern servers and embedded platforms [28]. FreeBSD is mainly written in C, and its large codebase has been continually developed and updated since the beginning of the 90's by a large community of individuals and organizations. The codebase can therefore be considered well established and its functionality widely tried and tested.

The LinkedList library in FreeBSD is defined in a queue header file, containing several function-like macros for a set of queue data structures. The singly linked list is a list where the ordering is upheld by the individual links themselves; each link - or node - defines the next element of the list, commonly using of pointers. To become useful in a computer program, each node in the list normally stores some value. In C this could be some structure or literal value, or a pointer to data somewhere else in memory.

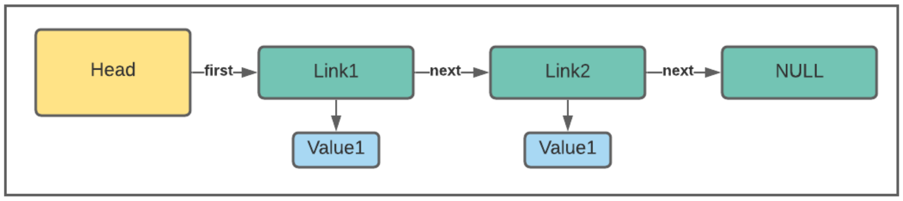


Figure 8. FreeBSD Singly Linked List Structure

In the FreeBSD implementation of a singly linked list, the list consists of two structures: the head, which only contains a pointer to the first entry, and the node structure which contains some data and a pointer to the next node (see figure 8).

2.7 Summary

It has been shown that testing is a necessary part of software verification, but many times not sufficient for a complete understanding of a system. Property based tests with automatic test-case generation used in a white-box test discipline should maximize the utility of tests as a means of describing an existing system.

According to section 2.3, formal specifications can be considered most useful when handling a complex data structure. Although the complexity of a data structure is somewhat subjective, a linked list will likely contain enough complexity that the application of formalization provides some use. At the same it is common and straightforward enough that it should be understood easily by the target audience.

The properties brought up in this chapter made TLA+ a valid choice of language to use during this project. The argument is that TLA+ seems able to handle many types of problems as well as being easy be to learn. The available IDE with PlusCal support is another factor in the choice of modelling language.

If Rust manages to gain popularity and becomes ubiquitous, it is likely that it will be a target language for porting a large amount of C code in the future. Thus, the decision of using C and Rust as source and target language for this thesis seems, from the literature, as a valid and interesting choice. Using these languages will also hopefully help answering the question if formal specifications could help in getting a deeper understanding of the original system, as an intermediate step in the porting process. C and Rust, although meant to solve similar problems, have disparate design philosophies, and offer a differing feature set. This means that the linked list implementations have the potential to look vastly different. A model can describe the system without using language specific constructs, which will be important in bridging the differences of the languages.

The FreeBSD list library includes several different queues and lists. Of these, the singly linked list was chosen as the sample to port. It should to the target audience be a familiar data structure, and it is one with which the authors are well versed. Choosing a common data structure as the subject should reduce the amount of time needed to understand and describe its intricacies in greater detail.

3 Method

This chapter provides a clear and extensive description of the methodology undertaken to address the question the thesis is based on. The intention is to enable the methodology to be reproducible. This is accomplished by dividing the method into several steps and for each one describing the processes they involve, the tools used, and criteria for completion.

3.1 Structure

To answer the question if formal specifications can be used during porting processes, the methodology was divided into discrete steps (see figure 9). The steps and their ordering can be compared to the V-model; a widely adopted software development project model [29]. The V-model describes a procedure, where high-level requirements are assembled first and then as the process goes on, the detail level is increased until implementation. The model also describes verification stages corresponding to each of these previous stages. The verification begins with the smallest granularity of tests and becomes more high-level until finally acceptance testing is performed. The methodology described in this thesis, can be seen as an inversion of the V-model where the process starts already having an original source. By writing tests, the behavior of the original was ascertained. On the highest abstraction level, the model specification lives. This represents the high-level requirements of the traditional V-model. The last verification step exists as a bridge between the original source and the ported source. Figure 9 shows a minimal representation of the V-model, and how this process can be seen as an inversion of it.

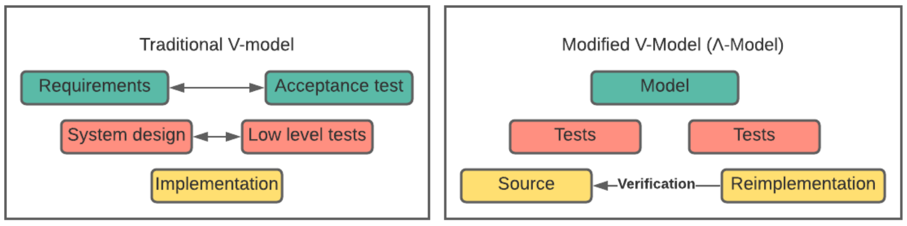


Figure 9. Simplified V-model of Software Development, Alongside a Modified Version of the Model

To get a more detailed description of the method, it was divided into the following steps:

1. Find an established and well used open-source library as a target source code.
2. Select a suitable testing framework and implement tests on the target source code to form an understanding of its behaviors and functionalities.
3. Find a formal specification language and reverse engineer the target source code by creating a model of it. This model is to be derived from the tests implemented in the previous step.
4. Implement tests for the functionalities specified in the created model.
5. Implement the functionalities in the target language based on the tests created.
6. Verify that the implementation is a true port of the target source code using a verification test-suite. The test-suite should run on both the original compiled code and a compiled version of the new implementation without modification.

3.2 Testing

For this thesis, testing was performed to create an understanding of the system under test. This is a reversal of the traditional role of the test code. To select tools and metrics that fitted this purpose best, constraints that applied to this specific scenario were considered. The tests needed to be able to verify to such a degree that all the behaviors of the code were accounted for. For example, there could have been some scenario which yielded unintuitive results; some behavior that did not match the porting parties' expectations. It was important to find these cases, not to expose them to fix, but to build a complete understanding of what the code did.

To achieve this in as an efficient manner as possible, the test-methodology chosen for the thesis fell on property-based testing. This practice entails that the test code describes some operation that is permissible to the system under test along with the constraints on the input data for that operation (see section 2.1.1). This approach was assumed to be particularly useful for this endeavor because it can be utilized without knowing the intentions of the system under test and can be seen as a guided exploration on the test subject rather than verification.

Like most data structures, the subject of the port will likely have some state which may or may not be permuted by the available operations. To make sure that the tests fully exercise this property of the code, and that the assumptions made also hold true over the lifetime of the system, a stateful test-suite was also created (see section 2.1.1). This meant that some operations and predicates which determine when the operation is valid were defined, along with a model.

To perform all these tests, a test framework was used. A test-framework provides a convenience in the test-writing process, foremost since it provides some code to drive the tests - code often referred to as a test runner. This frees the developer from the burden of verifying that all tests are run, as well as collecting statistics and logging failures. To determine when tests were sufficient, the metric of test-coverage by branch and line-of-code was used. That way it was easy to verify that each branch, and/or line had been exercised. There were, therefore, some requirements that a chosen framework had to fulfill to satisfy the needs of the project.

3.2.1 Frameworks

The choice of frameworks fell on GoogleTest [30] as it was a widely adopted unit testing library. It also allowed tests to be written in C++ which enabled the subject to be easily tested with minimal modification. RapidCheck [31] was chosen as a framework for the property-based testing since it was easy to use alongside GoogleTest; RapidCheck allows for specification of properties in an imperative style which works well for C++.

3.2.2 Environment

To execute tests on the C source code, a test environment needed to be set up. The project and dependencies were acquired and built automatically using CMake [32] which is an easy-to-use tool for automating these types of procedures. As an IDE, CLion [33] was chosen as it is a cross-platform IDE for C and C++. CLion also provides integration with CMake, allowing for a simple build procedure. The GNU C++ compiler g++ was used to compile the source code [34].

3.2.3 Implementation Process

The initial intention was to create tests in a methodical and structured way. To accomplish this, these steps of deductive reasoning and test implementations, inspired by Hoare logic (see section 2.2), were created and followed:

1. Select a functionality to write tests for.
2. Provide a logical basis for the program properties. This should be done by making several assertions on the functions precondition as well as on the results obtained on termination i.e., the values which the relevant variables will take before and after execution of the program.
3. Create tests based on the logical basis to see if the program is understood correctly and carries out the functionalities identified.
4. Modify tests to match any discrepancies found between actual and expected behavior.
5. Repeat all steps until all functionality of importance has been identified and tested.

3.3 Reimplementation

A TDD methodology was the approach chosen when implementing the formal model into Rust code, interleaving writing the tests with the actual implementation. The TDD used in this project followed the steps defined in figure 3, with the addition that new functionalities are found only from the model, by one of its invariants. The tests should be written so that when they pass, an invariant - or part of one - should be fulfilled. The fact that the tests are based on the model - and TDD's self-documenting nature - should make explicit the relation between the model and the implementation.

3.4 Verification

With the ported code in place, it should behave the same way as the original. To verify this, the port and source were both built as linkable libraries sharing the same application programming interface (API) allowing them to be tested by one single set of tests (see figure 10). This made it possible to swap out the system-under-test without changing the test-code, thereby serving as an independent verification that the two implementations behaves the same. This can be compared to how hardware components can be tested in a single test-bed and quickly swapped out for the next component.

This verification process requires that the source and target language have interoperability with, and can be called by some common language, preferably the one used to write the original tests (see section 3.2.3). In the case of this project - where C and Rust were the languages used - interoperability can be achieved by creating a foreign function interface (FFI) by declaring matching external functions that call the appropriate library functions.

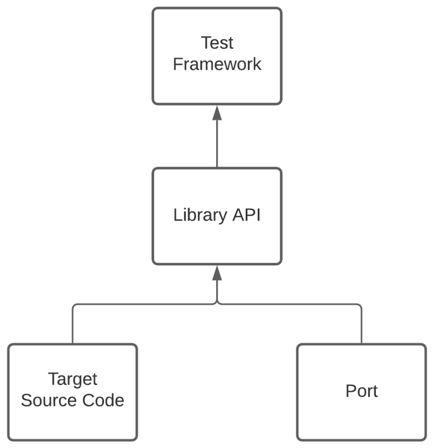


Figure 10. Verification Structure

As well as serving as the final evaluation of the final product, whether this last step is successful also serves as an evaluation of the method proposed. If it is successful then it can be concluded that this method is a feasible way of porting software. There is one caveat to this last step of the method process, knowing some of the differences between C and Rust. While the intention is to leave the original library as unmodified as possible, it may be necessary - or at least convenient - to insert one or a few helper functions into the library version of the source that manages the creation of lists etc. This will likely aid in keeping the interfaces to the libraries identical, while allowing the ported code to be more Rust "idiomatic". The key to the success of this type of verification process will be to not add functionality influencing the test results, and consequently invalidating the conclusion of the tests.

4 Implementation

This chapter serves as a documentation of the resulting code when following the porting methodology specified in the previous chapter. This is accomplished by showing the step-by-step process of how the porting of a function was accomplished. The functionality chosen as an example throughout this chapter is the concatenation of two separate lists into one. For each step in the process a sample implementation of the concatenation operator is presented. This choice was made since it is one of the less trivial functions of the original C code, providing a better foundation for the reader to get a grasp of how the porting process was applied. The complete implementation result can be seen at the project's GitHub repository [35].

4.1 Environment

The first step in the process was to isolate the files needed by the data structure so that they could be executed outside the context of the FreeBSD codebase. As the linked list code chosen was in the lower levels of the operating system, this was an elementary process since there few dependencies needed to be included. However, some FreeBSD specific OS-level definitions which were not present in the Linux standard library needed to be included. This complicated the process somewhat but was solved by including these into the queue source.

The linked list source code was fully made up of macros; predefined text snippets that are expanded into code during the pre-processing stage of compilation. This meant that the macros needed to be expanded into valid C code so that coverage of any tests could be measured. This was done by using the GCC compiler and the -E flag [36]. However, using this method when expanding code directly would yield no output since macros are expanded where they are used, not where they are defined. Therefore, a level of indirection was added by defining proxy functions using the macros which as well as expanding the text into valid C code enabled the coverage of this code to be measured. The added indirection somewhat changed the semantics of how the library was invoked by introducing function calls that would otherwise not be present. Defining these functions as inline could make it so that the resulting compiler output was identical, or very close to the one generated using the macros directly.

4.2 Behavioral Properties

The source code of the concatenation function (see figure 11) shows that the function takes two lists as arguments named *head1* and *head2* (line 2 and 3).

The argument names come from the fact that the first node of a list is referred to and stored in a head structure but will, where appropriate, be referred to as lists. The function concatenates the two lists into one where the head of the first list becomes the head of the concatenated, and the second lists head is set to be the next element to the tail of the first list. This indicates that the tail of the second list becomes the tail of the concatenated list.

```c
 1  void SLIST_CONCAT_impl(
 2      mySinglyLinkedListHead* head1,
 3      mySinglyLinkedListHead* head2){
 4      IntegerSLISTEntry *curelm = head1->slh_first;
 5      if (curelm == NULL) {
 6          if ((head1->slh_first = head2->slh_first) != NULL) {
 7              head2->slh_first = NULL;
 8          }
 9      } else if (head2->slh_first != NULL) {
10          while (curelm->entries.sle_next != NULL){
11              curelm = curelm->entries.sle_next;
12          }
13          curelm->entries.sle_next = head2->slh_first
14          head2->slh_first = NULL
15      }
16  }
```

Figure 11. The Concatenate Function

The concatenation is accomplished by first checking if the first list - the *head1* pointer argument - is populated by any nodes (line 5). If that is the case, it assigns the head of the second list, the *head2* pointer argument, to the address of the first. This is done while in the same statement checking if the second list is populated (line 6). If it is, it sets the second list to be empty to avoid duplicate references (line 7). In the case of the first list being populated, the function checks in the else if clause if the second list is populated as well (line 9). If it is, then the function iterates through the first list to reach its tail element (lines 10 to 12). The head of the second list then becomes the next element of the tail element in the first list (line 13). This makes the second list appended to the first i.e., they are now concatenated. As a last statement is the head of the second list set to NULL (line 14). This shows that the function not only ”copies” *head2* onto *head1*, but ”moves” the second list onto the end of the first.

When describing the valid states and functions of the program, a hypothesis about a certain function was deduced from the code using the previously mentioned Hoare logic (see section 3.2.3). This means that for each function of the target source code were a set of pre- and post-conditions identified. Each precondition needed to identify valid initial states on which the function could be applied, and each post-condition needed to be a statement about the resulting state of the list that should be true after the execution of the function. To form a behavioral understanding on the preconditions of the concatenate function, it can be stated that the function takes two lists as arguments. This allows the conclusion that the precondition can be based on the following four situations:

- Both lists are empty
- The first list is populated and the second is empty
- The first list is empty and the second is populated
- Both lists are populated

These four situations form the foundation for these following behavioral statements where the first one is when both lists are empty. In 12 and forward, the list variables are prepended with P or R - corresponding to the Hoare logic (see section 3.2.3) - indicating whether they are the values before or after the operation (see figure 12). The *isEmpty* helper function indicates whether a list is empty, and the behavioral reasoning states that if both lists are empty, the test needs to check that both lists still are empty after the concatenation procedure.

```
Precondition: isEmpty(P_list1) AND isEmpty(P_list2)

Function: concatenate(P_list1, P_list2)

Postcondition: isEmpty(R_list1) AND isEmpty(R_list2)
```

Figure 12. Properties of Concatenation Operation 1

Following this stipulation, it can be stated that the two cases where only one list is populated results in the same end state. This forms the behavioral basis seen in figure 13. The *validList* helper function checks that a list is valid, *size* returns the number of elements in the list and *first* and *last* return the first and the last element of a list.

```
Precondition:
    • case1 -> validList(P_list1) AND isEmpty(P_list2)
    • case2 -> isEmpty(P_list1) AND validList(P_list2)

Function: concatenate(P_list1, P_list2)

Postcondition:
    • case1 -> isEmpty(R_list2) AND
               first(R_list1) = first(P_list1) AND
               last(R_list1) = last(P_list1) AND
               size(R_list1) = size(P_list1)
    • case2 -> isEmpty(R_list2) AND
               first(R_list1) = first(P_list2) AND
               last(R_list1) = last(P_list2) AND
               size(R_list1) = size(P_list2)
```

Figure 13. Properties of Concatenation Operation 2

This basis indicates that the tests need to assert that the head of the first element of the populated list in the precondition always is returned being assigned to the element of the first list in the post-condition. It must as well assert that the last element of the populated list becomes the last element of the concatenated list. The tests must also assert that the second list in the post-condition always is empty, and that the length of the concatenated list always is of the same length as the initially populated one.

If both lists are populated the behavioral basis becomes more complicated since several more assertions can be made on the post-condition (see figure 14).

```
Precondition: validlist(P_list1) AND validList(P_list2)

Function: concatenate(P_list1, P_list2)

Postcondition: first(R_list1) = first(P_list1) AND
               last(R_list1) = last(P_list2) AND
               size(R_list1) =
               (size(P_list1)+size(P_list2)) AND
               isEmpty(R_list2) AND
               {P_list1} SUBSETOF {R_list1} AND
               {P_list2} SUBSETOF {R_list1}
```

Figure 14. Properties of Concatenation Operation 3

This behavioral basis states that in the case of both lists being populated, the head of the concatenated list must always be the head of the first list in the precondition. The basis also indicates that the tail of the second list always becomes the tail of the concatenated list. Since the lists are being concatenated the size of the concatenated list must also always match the sum of the size of both initial lists. This also means that the size of the second list always must be zero as a post condition i.e., the second list becomes empty. Lastly it states that the nodes of the two initial lists must always be a subset of the concatenated list.

4.3 Tests

To verify that the actual behavior conformed to the behavioral basis, tests were written asserting that this is the case. Both unit and stateful property-based tests were written in *GoogleTest* combined with *RapidCheck*. An example of a testing property for the concatenate function takes vectors of integer values (see lines 3 and 4 figure 15) generated by *RapidCheck* and converts them into two separate lists of the *mySinglyLinkedList* type through the helper function *createList* (lines 6 through 9). These vectors were of an arbitrary number of elements, including empty.

```
 1  RC_GTEST_PROP(SLIST,
 2  concatenatingListsYieldsCorrectSizeOnList1,
 3  (std::vector<IntegerSLISTEntry> a,
 4  std::vector<IntegerSLISTEntry> b)){
 5
 6      mySinglyLinkedListHead headA{nullptr};
 7      createList(headA, a);
 8      mySinglyLinkedListHead headB{nullptr};
 9      createList(headB, b);
10      unsigned int expectedSize = a.size() + b.size();
11
12      SLIST_CONCAT_impl(&headA, &headB);
13
14      IntegerSLISTEntry *entry;
15      int actualSize = 0;
16
17      SLIST_FOREACH(entry, &headA, entries) {
18          actualSize++;
19      }
20
21      RC_ASSERT(expectedSize == actualSize)
22      RC_ASSERT(headB.slh_first == nullptr);
23  }
```

Figure 15. Size Test

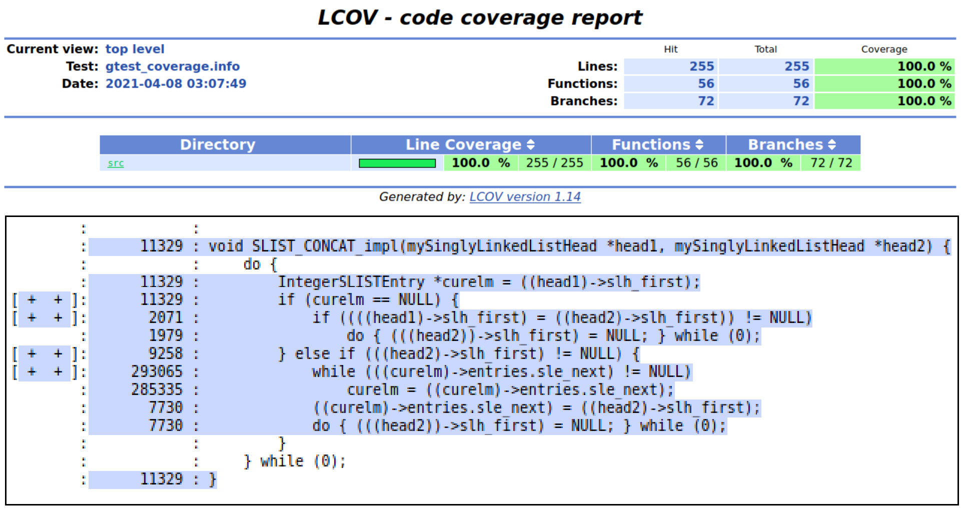
After the lists have been concatenated the test asserts that the size of the concatenated list always is of the same size as the sum of the sizes of the two separate lists (see line 21 figure 15). This is enabled by saving the total size of the two vector arguments in the variable *expectedSize* (line 10). This variable is then compared with the variable *actualSize* which receives the size of the concatenated list by iterating through every element via the target source code macro function SLIST_FOREACH which increments *actualSize* by one for each iteration (lines 17 and 18). The test also asserts that the second list is always empty after the concatenation procedure (line 22). These procedures provide verification coverage that the size state of the lists found during the behavioral understanding stage (see section 3.2.3) holds for every tested situation.

```
struct SLIST_concatenate :
    rc::state::Command<SLIST_model, mySinglyLinkedListHead> {

    std::vector<int> concatenate_with =
        *rc::gen::arbitrary<std::vector<int>>();

    void apply(SLIST_model &model) const override {
        for(auto i: concatenate_with)
        model.list.push_back(i);
    }

    void run(
        const SLIST_model &model,
        mySinglyLinkedListHead &head1) const override {
        auto head2 = new mySinglyLinkedListHead();
        createList(head2, concatenate_with);
        SLIST_CONCAT_impl(&head1, head2);
        RC_ASSERT(SLIST_EMPTY_impl(head2));
    }

    void show(std::ostream &os) const override {
        os << "SLIST_CONCAT: ";
        for(auto i: concatenate_with) { os << i << ", "; }
    }
};
```

Figure 16. Stateful RapidCheck Test Structure

Along with the unit-test style cases, a property-based stateful test-suite was created for the list type. Stateful meaning that each test is executed on the same or equivalent instance of the system-under-test, and any mutation to the system is retained for the next operation. To verify the state at each point, a model was inserted into the test suite. This model is some simplified version of the system under test that implements equivalent operations, and with which the system state is be verified to be correct. In this test-discipline, a set of operations were defined. Each operation implements a function that mutates the model, and one that mutates the system-under-test as well as optionally any assertions that can be made to verify that the system state conforms to the model state. The operations also optionally define some constraint under which it is valid. This allows for operations to be skipped when the operation is not properly defined for the current model state, e.g., getting the first element of an empty list. Operations were defined by implementing a class (see figure 16). In the case of the concatenate operation, an arbitrary vector was generated to create a list (lines 4 and 5), used as the basis for the concatenation (lines 14-18).

To verify that all branches of the tested code were covered by the tests, coverage metrics were collected using the tool gcov and visualized with lcov (see figure 17). This allows for metrics such as which branches were not executed, and how many times each line was executed to be easily ascertained. After the tests specified by the behavioral bases were implemented, the coverage was checked to make sure no lines or branches were untested.


LCOV - code coverage report

Current view: top level
Test: gtest_coverage.info
Date: 2021-04-08 03:07:49

	Hit	Total	Coverage
Lines:	255	255	100.0 %
Functions:	56	56	100.0 %
Branches:	72	72	100.0 %

Directory	Line Coverage			Functions		Branches	
src	🟩	100.0 %	255 / 255	100.0 %	56 / 56	100.0 %	72 / 72

Generated by: LCOV version 1.14

```
         :          :
         :    11329 : void SLIST_CONCAT_impl(mySinglyLinkedListHead *head1, mySinglyLinkedListHead *head2) {
         :          :     do {
         :    11329 :         IntegerSLISTEntry *curelm = ((head1)->slh_first);
[ +  + ]:    11329 :         if (curelm == NULL) {
[ +  + ]:     2071 :             if ((((head1)->slh_first) = ((head2)->slh_first)) != NULL)
         :     1979 :                 do { (((head2))->slh_first) = NULL; } while (0);
[ +  + ]:     9258 :         } else if (((head2)->slh_first) != NULL) {
[ +  + ]:   293065 :             while (((curelm)->entries.sle_next) != NULL)
         :   285335 :                 curelm = ((curelm)->entries.sle_next);
         :     7730 :             ((curelm)->entries.sle_next) = ((head2)->slh_first);
         :     7730 :             do { (((head2))->slh_first) = NULL; } while (0);
         :          :         }
         :          :     } while (0);
         :    11329 : }
```

Figure 17. Visualization of Coverage Metrics Using lcov

4.4 Model

With the tests in place, it was time for the creation of the TLA+ model. The model was split into two modules named LinkedList and Main. LinkedList describes the data structure itself, the structure of the data and the operations that are permitted to be performed on it. Main contains the model checking algorithm verifying that the model works for each possible state, as well as the invariants verifying that the structure of the linked list holds for each state of the algorithm.

TLA+ almost entirely consists of mathematical sets and functions. Therefore, inspired by the linked list implementation in the book Practical TLA+ [20], was the linked list in this implementation defined as a function. It has a domain of a set of labels, which could be a pointer or a reference in a source code implementation, where each domain entry represents a node in the linked list. The range of the function consists of a function with a finite domain with the labels next and value representing the variables stored in each node (see figure 18).

```
[NULL |-> [next |-> NULL, value |-> NULL]]

[label1 |-> [next |-> NULL, value |-> INT],
 label2 |-> [next |-> label1, value |-> INT],
 label3 |-> [next |-> label2, value |-> INT]]
```

Figure 18. Linked List Examples

The label "next" maps to some other label in the linked list domain, indicating that it is the element following the current of the list, or NULL, indicating that the current node is the last. The "value" label maps to some integer held by each node, which in the target source code could be any arbitrary primitive data type or structure.

To check that a function is a valid linked list, a TLA+ operator called IsLinkedList was created (see figure 19). It takes a function as argument, named PointerMap, and checks that for each element in the domain of the PointerMap - NULL included and the label of the first element of the list excluded - there exists a node with the next value representing each node in the function. If that is the case it returns true and false if not. Based on this definition, lists containing cycles are also excluded.

```
IsLinkedList(PointerMap) ==
    \A el \in ((DOMAIN PointerMap \union {NULL}) \
    {First(PointerMap)}): \E x \in DOMAIN PointerMap:
    PointerMap[x]["next"] = el  /\ el /= x
```

Figure 19. IsLinkedList Operator

As a part of creating arbitrary lists for usage during the model checking process, an operator called PointerMaps was created. It creates a set of functions by mapping from a domain of unique labels it receives as argument to a range (see figure 20). PointerMaps creates with this every possible permutation of the domain mapping to every possible permutation of the next range. This means that PointerMaps creates a set of functions that include all valid lists, but also mappings which are not valid.

```
PointerMaps(domain) ==
    [domain -> [value: VALUE, next: domain \union {NULL}]]
```

Figure 20. PointerMaps Operator

PointerMaps, in combination with IslinkedList, is used in an operator called LinkedList which filters the output from PointerMaps by choosing one of the mappings that IsLinkedList validates as a LinkedList (see figure 21). This list, represented as the variable pm, is then returned to be used in the model checking algorithm. If LinkedList on the other hand receives an empty set of domain labels, it returns the representation of an empty list. The empty list is a special case, where NULL is allowed as the only value in the domain of the list (see figure 18).

```
LinkedList(domain) ==
    IF NULL \in domain
        THEN Assert(FALSE, "Null cannot be in domain")
    ELSE IF domain \subseteq {}
        THEN EmptyList
    ELSE
        CHOOSE pm \in PointerMaps(domain):
        IsLinkedList(pm)
```

Figure 21. LinkedList Operator

With the definition and the ability to create linked lists in place, the concatenation function was added to the model (see figure 22). It was accomplished via a TLA+ operator taking two lists as arguments and checks if either of the lists are empty. If that is the case, it returns the lists in a TLA+ sequence specified with << >>, where the empty list is placed at the second index of the sequence. A TLA+ sequence is another way of representing a function where the index of the sequence is equivalent to the domain of a function. It is used in this case to enable the operator to return two lists in a specified order to enable usage using the model checking process (see figure 23). If both lists are populated there is the special situation where the next value of the last element of the first list needs to be mutated. This is needed since the last element of a list always points to a NULL value and instead needs to show the label of the first element in the second list for them to be considered concatenated.

```
Concat(list, list2) ==
    IF Empty(list) THEN
        <<list2, list>>
    ELSE IF Empty(list2) THEN
        <<list, list2>>
    ELSE
        LET newLast ==
            CHOOSE x \in [{Last(list)} ->
                [value: VALUE, next: {First(list2)}]]:TRUE
        IN
        <<(newLast @@ list) @@ list2, EmptyList>>
```

Figure 22. Concatenation Operator

The concatenation operator is called in the model's PlusCal algorithm (see TLA+ Tools section 2.3.1). The concatenation steps consist of a set of labels - PRECONCAT and CONCAT - with its associated operations (see figure 23). In PlusCal, the code in each label is disjunct, meaning that each one is a discrete step between which time can progress, and after which the algorithm can stop. Conversely, the code in one label is seen as an atomic unit of progress that cannot be divided further. For this reason, labels must be inserted when, for example, mutating a variable twice.

```
PRECONCAT:
with size \in 0..1 do
list2 := LinkedList(NewDomain(size, list));
end with;

CONCAT:
temp := Concat(list, list2);
list := temp[1];
list2:= temp[2];
```

Figure 23. Concatenation PlusCal Code

All operations under the same label have an associative property meaning that they can be performed in any order. Dividing the operations between labels thereby ensures that the steps of the algorithm are performed in the order intended. PRECONCAT uses the “with” behavior which instructs the model to run the algorithm within the with clause, with every value in the set it is provided – in this case the values 0 and 1. This enables the model checker to perform concatenation operations with all the list sizes to the model’s main list variable. The call to the concatenation operator is made under the CONCAT label and stores the returned sequence in the temp variable. From there is then the returned concatenated list - index 1 of the returned sequence - stored in the *list* variable and the returned empty list – index 2 - is stored in the *list2* variable. For every step of the algorithm, the assumptions made about the outcome of the concatenation operator were checked to hold true. This is accomplished through an invariant; a property of a mathematical object which remains unchanged after operations or transformations have been applied to an object of interest (see figure 24). This means that they verify for each state that the linked list structure stays valid through the entire model checking phase.

```
ConcatInvariant ==
    IF Empty(list) THEN Concat(list, list2)[1] = list2
         /\ Empty(Concat(list, list2)[2])
    ELSE IF Empty(list2) THEN Concat(list2, list)[1] = list
         /\ Empty(Concat(list2, list)[2])
    ELSE /\ DOMAIN list \subseteq DOMAIN Concat(list, list2)[1]
         /\ Empty(Concat(list, list2)[2])
         /\ DOMAIN list \subseteq DOMAIN Concat(list2, list)[1]
         /\ Empty(Concat(list2, list)[2])
         /\ DOMAIN list2 \subseteq DOMAIN Concat(list, list2)[1]
         /\ DOMAIN list2 \subseteq DOMAIN Concat(list2, list)[1]
```

Figure 24. Concatenation Invariant

Several other invariant operators were also implemented (see figure 25). In the case of this model were they based on the properties established on the list structure such as the list should always be either empty or have a first and last element etc.

```
HasFirst ==
    Empty(list) \/
    \E el \in DOMAIN list: First(list) = el /\
    First(list) \notin Range(list)

HasLast ==
    Empty(list) \/
    \E el \in DOMAIN list: list[el]["next"] = NULL

NullNotInDomain ==
    Empty(list) \/ NULL \notin DOMAIN list
```

Figure 25. Some Model Invariants

Lastly, since the state was written in PlusCal, it was translated by the TLA+ toolbox into pure TLA+ (see figure 26). This is necessary since the model checker can only work with pure TLA+.

```
PRECONCAT ==
    /\ pc = "PRECONCAT"
    /\ \E size \in 0..1:
       list2' = LinkedList(NewDomain(size, list))
    /\ pc' = "CONCAT"
    /\ UNCHANGED << depth, index, i, from, list, temp,
       arg, lab >>

CONCAT ==
    /\ pc = "CONCAT"
    /\ temp' = Concat(list, list2)
    /\ list' = temp'[1]
    /\ list2' = temp'[2]
    /\ pc' = "INCREMENT"
    /\ UNCHANGED << depth, index, i, from, arg, lab >>
```

Figure 26. Concatenation State In Pure TLA+

4.5 Reimplementation

Initially when discussing the details of the reimplementation, it was decided that two versions of the port should be made - one unsafe and one safe (see section 2.5). This was due to some apprehension regarding what was possible to do in safe Rust, and if a linked list could be implemented with the limitations safe Rust brings. If possible, would both versions utilize safe Rust, but not being restricted in one of them could lead to different and interesting model interpretations.

4.5.1 Unsafe Version

The unsafe version was made to match the C-style usage more closely, and references to nodes could be passed to the functions and mutated in place. To achieve this, the nodes were wrapped in a Box type - a type of smart pointer (see line 4 figure 27) [37]. By wrapping nodes in these smart pointers, the problem of infinitely recursive types as well as other issues were solved. Working with references in general also removes some of the constraints which otherwise would be necessary to put on the concrete type used for the list values, e.g., the clone and/or copy traits. The node struct is used in the linked list structure which stores the head variable representing the first node of the list (see line 8 in figure 27).

```
1 pub trait LinkedListValue: Debug {}
2 pub struct Node<T: LinkedListValue> {
3     pub value: T,
4     pub next: Option<Box<Node<T>>>
5 }
6
7 pub struct LinkedList<T: LinkedListValue> {
8     pub head: Option<Node<T>>
9 }
```

Figure 27. Unsafe Data Structures

Working with boxed references in Rust is like working with references or pointers in C or C++, but with explicit function calls for getting access to the reference - either immutably or mutably. However, unlike a reference in C++, it is not allowed to have both a mutable and immutable reference to an object at the same time, and no two mutable references may refer to the same object, with some exceptions (see section 2.5). As well as using the Box type, it should also be noted that the concept of NULL is not often used in Rust and the consensus is to avoid NULL-pointers at all costs [27]. Instead, values which could be NULL-like are wrapped in the Option type, where a value of None represents the NULL case.

Since a TDD methodology was used for the reimplementation, the tests for the unsafe version were then implemented. Every test written was directly based on some invariant or property defined by the model (see figure 28 for concatenate example). Several of the hypotheses formed earlier and verified using tests could be combined into a single invariant checking many things, and in the end a single test case in the reimplementation.

Target Source Code Test	Model Invariant	Rust Test
concatenatingListsEmptiesSecondList	ConcatInvariant	prop_concat
concatenatingListsMakesHeadALast-NextHeadBFirst	ConcatInvariant	prop_concat
concatenatingListsYieldsCorrectSizeOn-List1	ConcatInvariant	prop_concat
concatenatingWithHead1EmptyPuts-HeadAFirstOnHeadB	ConcatInvariant	prop_concat
concatenatingWithHeadBEmptyPuts-HeadAFirstOnHeadA	ConcatInvariant	prop_concat
concatenatingListsPreservesOrder	ConcatInvariant	prop_concat

Figure 28. Tests Corresponding to Concatenation Invariant

To continue the concatenation example, a test was implemented verifying that the sizes of the lists after the concatenation operation were correct (see figure 29). The test function receives two vectors as argument (line 1). These vectors are used to create lists through the function *linked_list_from*, which creates lists with equivalent values as the vector arguments (lines 2 and 3). These lists are then concatenated followed by asserting that the *linked_list_2* variable is empty and that size of the *linked_list_1* variable corresponds to the sum of the size of the two vector arguments (lines 6 to 9). It is also asserted that the concatenated list, the *linked_list_1* variable, is of the same size as the sum of the length of the two received vector arguments (line 10). To verify that the values stored in the concatenated list are correct, it is lastly iterated through, and each value compared to those stored in the vectors (lines 13-24). Just as in the original C tests, property-based tests were used through RapidCheck.

```
fn prop_concat(list_1: Vec<i32>, list_2: Vec<i32>) -> bool {

    let mut linked_list_1 = linked_list_from(&list_1);
    let mut linked_list_2 = linked_list_from(&list_2);
    LinkedList::concat(&mut linked_list_1, &mut linked_list_2);

    if linked_list_2.size() != 0 ||
         linked_list_1.size() != list_1.len() + list_2.len() {
         return false;
    }

    let mut i = 0;
    for x in linked_list_1.iter() {
        if i < list_1.len() {
            if *x != list_1[i] {
                return false;
            }
        } else {
            if *x != list_2[i-list_1.len()] {
                return false;
            }
        }
        i += 1;
    }
    true
}
```

Figure 29. Unsafe Size Test

The concatenation functionality was then implemented in accordance with the tests (see figure 30). The function receives two linked list arguments (line 2). If the first one has some nodes stored, it iterates through the list to reach the last one (lines 4-9). It then checks if the second list is populated by performing a pattern matching (line 11). If the list is populated then the head of that list is attached to the previously reached last element of the first list (line 12). If on the other hand the first list is empty, then the head of the second list is moved to that empty first list (line 17). This makes the content of the second list being moved to the first while simultaneously asserting that the second list becomes empty.

```
 1 pub fn concat(
 2 list_1: &mut LinkedList<T>, list_2: &mut LinkedList<T>) {
 3 
 4     if let Some(head_1) = &mut list_1.head {
 5         let mut current = &mut head_1.next;
 6 
 7         while let Some(c) = current {
 8             current = &mut c.next;
 9         }
10 
11         match &list_2.head {
12             Some(_) => *current = Some(Box::new(list_2.head.
13             take().unwrap())),
14             None => {}
15         }
16     } else {
17         *&mut list_1.head = list_2.head.take();
18     }
19 }
```

Figure 30. Unsafe Concatenation Function

In line 17 figure 30, the function *take* is used. When *take* is called on an Option type, the object it is called upon becomes None, and a new Option containing the value is returned. This essentially moves the value from one Option to another in place [38], in this case appending the head of one list onto the tail of another.

In the cases where an immutable reference to an object was needed, but only needed for equality checks - i.e., never read from - this was solved by using the address of the object in question instead of a reference to it. This is both safe and mitigates the issues working directly with references would cause. The implementation of the insert before function (see figure 31), which takes the address of a node and a value to insert, serves as an example of this.

```rust
pub fn insert_before(&mut self, node: *const Node<T>, value: T) {
    if self.head.is_some() {
        let head = self.head.as_mut().unwrap();
        if node == head as *const _ {
            self.insert_head(value);
        } else
        {
            head.insert_before(node, value);
        }
    } else {
        panic!("Cannot insert into empty list");
    }
}
```

Figure 31. Insert Before

If a reference was used to identify the node to insert before, the signature would have been valid but very difficult to use. That is, since the list would have a reference to the node, which would also be passed a reference to the function, this usage is disallowed by the borrow checker. As a work-around, something like what can be seen in figure 32 would have been needed. Working directly with addresses only works when there is some guarantee that the node itself will never unexpectedly move around in memory. Since the node does not implement either copy or clone, this holds for all cases except where the user explicitly moved it.

```rust
pub fn create_unsafe_ref<T>(node: &Node<T>) -> Option<&Node<T>> {
    let mut return_node: Option<&Node<T>> = None;
    unsafe {
        return node = Some(&*node as *const _);
    }
    return_node
}
```

Figure 32. Unsafe Helper Function

The signature for *insert_before* could then instead be as in figure 33. In the end it should be noted that even though there were initially thought that unsafe code would be necessary to implement the functionality using reference, unsafe was not necessary.

```rust
    pub fn insert_before(&mut self, node: &Node<T>, value: T)
    ...
```

Figure 33. Insert Before With References

4.5.2 Safe Version

The safe version was implemented through a structure consisting of several fields. The ”nodes” field is a vector holding each node located on the list (see line 10 in figure 34). The ”head” field stores the vector index position of the first node of the list (line 11) and the ”size” field indicates how many nodes the list currently consists of (line 12). To know which indexes of the nodes vector - if any - are free due to nodes having been removed from the list a stack called ”freeindex” is also stored (line 13). The node structure stores the value of the vector index it is located on in contrast to what position it has in the list (line 4). It also stores the vector index of the node which follows it in the list in its ”next” field (line 5) and has a ”value” field for node value storage as well (line 6). As opposed to the unsafe version, the idea here is to work with copies of the nodes rather than references. These copies are then used by the list similarly as if they are references, but with the difference that the user cannot change the structure of the list without calling predefined methods.

```rust
1  pub trait LinkedListValue: Copy + Clone {}
2  #[derive(Copy, Clone)]
3  pub struct Node<T: LinkedListValue> {
4      index: usize,
5      next: Option<usize>,
6      pub value: T
7  }
8
9  pub struct LinkedList<T: LinkedListValue> {
10     nodes: Vec<Node<T>>,
11     head: Option<usize>,
12     size: usize,
13     freeindex: Vec<usize>
14 }
```

Figure 34. Safe Data Structures

The safe test has many similarities to the unsafe version where it receives two vector arguments of arbitrary sizes and values (see line 1 figure 35), used to create two lists which get concatenated. The sizes of the lists are then asserted followed by the concatenated list being iterated through to check that the values are correct.

```
 1  fn prop_concat(list_1: Vec<i32>, list_2: Vec<i32>) -> bool {
 2      let mut linked_list_1 = linked_list_from(&list_1);
 3      let mut linked_list_2 = linked_list_from(&list_2);
 4  
 5      match LinkedList::concat(&mut linked_list_1 ,
 6      &mut linked_list_2)
 7      {
 8          Ok(_) => {
 9              if linked_list_2.size() != 0 ||
10              linked_list_1.size() != list_1.len() + list_2.len() {
11                  return false;
12              }
13  
14              let mut i = 0;
15              for x in linked_list_1.iter() {
16                  if i < list_1.len() {
17                      if x != list_1[i] {
18                          return false;
19                      }
20                  } else {
21                      if x != list_2[i-list_1.len()] {
22                          return false;
23                      }
24                  }
25                  i += 1;
26              }
27              true
28          },
29          _ => false
30      }
31  }
```

Figure 35. Safe Test

The safe reimplementation of the concatenation operation takes two lists as arguments (see line 1 figure 36). If the second list is not populated, the function simply returns since in this case it is irrelevant if the first list is populated or not (lines 4-6). If the second list is populated and the first list is empty however, *mem::swap* is used (lines 8-11). This swaps the memory contents of the lists which provides the intended functionality of moving the second list to the first and vice versa. If both lists on the other hand are populated, then the last element of the first list is found by using the helper function *node_at_index* (line 13). This function takes the list position of the node of interest as argument and iterates through the list until it has reached the node in question and returns it. The head of the second list is then inserted after this node (lines 23-26). A procedure accomplished by using the *insert_after* function which takes a node and a value as argument and creates a new node with said value and inserts it after the received node. The head then lastly is removed to remove the node from the second list (lines 27-31). When this procedure of insertion and removal has been performed for each element of the second list, the lists are concatenated and an Ok result is returned (line 34).

```
1  pub fn concat(list_1: &mut Self, list_2: &mut Self) ->
2  LinkedListResult<()>  {
3  
4      if list_2.is_empty() {
5          return Ok(());
6      }
7  
8      if list_1.is_empty() {
9          mem::swap(list_1, list_2);
10         return Ok(());
11     }
12 
13     let mut list_1_end = match list_1.node_at_index(
14     list_1.size()-1) {
15         Ok(Some(node)) => node,
16         Ok(None) => {
17             mem::swap(list_1, list_2);
18             return Ok(())
19         },
20         Err(e) => return Err(e)
21     };
22 
23     while let Some(next) = list_2.head() {
24         if let Ok(new_end) = list_1.insert_after(
25         list_1_end, next.value) {
26             list_1_end = new_end;
27             match list_2.remove_head() {
28                 Ok(_) => {},
29                     _ => return Err(LinkedListError{message:
30                     String::from("Error concatenating lists")})
31                 }
32             }
33         }
34     Ok(())
35 }
```

Figure 36. Safe Concatenation Implementation

4.6 Verification

To reliably verify the result, the two ports were - as well as the original code - wrapped in a common API (see figure 37). Each version was built as a library and linked into the test program. The common interface always returns an unsigned integer where negative values means that an error occurred. The library calls were wrapped in implementations of a *LinkedListLib* class - *Clib* and *RustLib* - which were responsible for calling the correct library functions. This helped in making the libraries easily interchangeable (lines 4 and 10).

As Rust has no stable API for C++, the interface was called from C, although other parts of the test code were written in C++. The intention was to reuse the original property-based tests with the two linkable libraries. To call Rust code from C, headers specifying the library interface must be created. No tool that does this is directly available in the common rust build tools, but the community provides a tool called cbindgen to do this automatically [39].

```
 1  int32_t Clib::concatenate(
 2  uintptr_t identifier_list_1,
 3  uintptr_t identifier_list_2) {
 4      return clib_concatenate(identifier_list_1, identifier_list_2);
 5  }
 6
 7  int32_t RustLib::concatenate(
 8  uintptr_t identifier_list_1,
 9  uintptr_t identifier_list_2) {
10      return rlib_concatenate(identifier_list_1, identifier_list_2);
11  }
```

Figure 37. Agnostic Test APIs

An example of how this interface works is the concatenate call (see figure 38). It returns zero if no error occurs (lines 7 and 17), otherwise another integer value representing an error-code is returned (line 21). The library is in control of the memory allocations for, and the managing of the lists. A call to the library can create a new or modify an existing list. When creating a list, the library returned a unique identifier corresponding to that list. All other functions represented either a modification to an existing list, or a getter for a value from the list or some property of the list. These functions take one or more list identifiers and optionally some value and/or index. The libraries were then subjected to property-testing. The original test suite could be reused, although with slight modification to work with the list libraries.

```
int32_t clib_concatenate(
uintptr_t identifier_list_1,
uintptr_t identifier_list_2) {
    auto head_1 = lists.at(identifier_list_1);
    auto head_2 = lists.at(identifier_list_2);
    SLIST_CONCAT_impl(head_1, head_2);
    return 0;
}

pub extern "C" fn rlib_concatenate(
identifier_list_1: usize,
identifier_list_2: usize) -> i32 {
    let mut lists = LISTS.lock().unwrap();
    if let (Some(list_1), Some(list_2)) =
    lists.borrow_two(identifier_list_1, identifier_list_2) {
        match LinkedList::concat(list_1, list_2) {
        Ok => 0,
        _ => -1
        }
    } else {
        -1
    }
}
```

Figure 38. FFI Interface Concatenate Functions

5 Result

The functionalities of the target source code are deemed to be ported successfully, although some of the macro definitions were not modelled directly. The equivalent functionality provided for the omitted macros was provided by more than one macro, only one of each such case was modelled.

Functionality	Ported
HEAD	X
CLASS_HEAD	-
HEAD_INITIALIZER	-
ENTRY	X
CLASS_ENTRY	-
INIT	X
EMPTY	X
END	X
FIRST	X
NEXT	X
FOREACH	X
FOREACH_FROM	X
FOREACH_SAFE	-
FOREACH_FROM_SAFE	-
INSERT_HEAD	X
INSERT_AFTER	X
CONCAT	X
REMOVE_AFTER	X
REMOVE_HEAD	X
REMOVE	X
REMOVE_PREVPTR	-
SWAP	X

Figure 39. Ported Functions

All the macros specified in the target source code are present in listing 39. The functions marked with - are not present directly in the model or reimplementation. A detailed reason why is given for each of them:

- CLASS_HEAD: This macro is meant specifically for usage with class types as values. In the concrete test cases, a class type was not used and therefore this macro was omitted.

- HEAD_INITIALIZER: This macro simply expands to {NULL}, and as such does not provide enough functionality to warrant modelling. Its usage is very specific to the C language.
- FOREACH_SAFE and FOREACH_FROM_SAFE: The functionalities provided by these macros are very similar to what FOREACH and FOREACH_FROM provides, respectively. The difference is that the safe functions permits both removal and freeing the current node without interfering with the list traversal. A functionality not deemed necessary to model.
- REMOVE_PREVPTR: This provides the same functionality as REMOVE but differs in implementation to provide a faster running time.

The original C source and the ported code was built as a linkable library to enable the verification procedure. These libraries were then subjected to a verification test-suite, which was the original stateful property-based tests modified slightly to work with the list libraries. The result of these tests indicate that the libraries built from the Rust source provide the same functionality as the one built from the C source. Therefore, the functionality can reasonably be determined to be successfully ported.

6 Discussion

The usage of formal specifications during the project seemed to increase the certainty that the functionality ported was true to the original; it gave a higher degree of understanding of the design of the target source code. It was beforehand uncertain if basing the model on tests was a valid work process, but the usage of property-based tests provided a usable basis for the model, indicating at least the feasibility of the method. Since the authors experience of working with porting was limited it is difficult to make judgements on the efficiency of this method compared to others. However, the result does not rule out working with the methodology created in this project further.

The experience of working with TLA+ showed that the language needed some initial investment of time to understand it well enough to apply it in the project. It had a steep learning curve, at least when being used working with programming languages as was the case for the authors. The reason for this is that TLA+ in many ways looks like a regular programming language but in fact works quite differently. For example, the statements in a TLA+ state are commutative; they can be evaluated in any order. This differs a lot from how it works in a language like C where each statement is executed in the order they are written and getting acquainted with this behavior of TLA+ took some time. PlusCal therefore, was a well-received inclusion to the language, making this transition easier.

For the sample source code chosen for this thesis (see section 2.6), the verification step of the process can seem excessive. When porting a more complex system with many modules, each with its own states and transitions that may be valid or not depending on other modules states, it may however be a necessary part of the process and should provide verification to a very high degree that the ported behavior is correct. In the end, the ported code of this project will likely not be used as a standalone linkable library, but instead compiled as part of a bigger project. However, since the source module should be isolated enough to be tested independently, and have formal specifications written for it, building it as a standalone library should pose only a minor issue. If it is not possible to build into a standalone library, it would likely indicate a failure in previous steps of the process.

One of the issues encountered during the project was with the choice of target source code - the linked list. The intended outcome using this as a source, was to reduce the amount of time needed to understand and explain it. This was achieved, but came at a cost. The choice of target source code may have impacted the level of usefulness of the methodology; a complex system may likely benefit greatly if not make it necessary to use a rigorous specification process in preparation for a successful porting. For example, if there are different modules who interact in complex ways, and where some state in module A prohibits some state in module B or vice versa, this is critical information that is expressed best as an invariant. However, if the source is not very complex, the process may prove to be more cumbersome than the value gained from following it. A simple system as the linked list library used in this project has a limited state and does not have any interaction between the different modules. This makes the information gained from a model limited. This is further emphasized if the final module is run through the same tests as the original code, which was the case in this project.

Other possible issues discovered during the project is how to consider implicit functionality and features such as access levels - properties which often differ between programming languages. For example, access levels in C are different from those available in Rust. Properties such as these may need to be considered but are not possible to model and perhaps should be incorporated into the process some other way. How to address these possible situations puts responsibility on the engineer doing the porting. The engineer also needs to consider how the code has originally been used; some use-cases may be strange or abuse unintended functionality. This could be an issue with the methodology used in this project since it puts a large emphasis on matching the original interface, when constraints of the target language may make a similar but different design more suitable. This could, of course, be an issue using any porting methodology. Particularly to this methodology the trade-off is that it provides security that the result conforms to a high degree with the original work. A strict adherence to a model, however, removes some of the freedom from engineers to apply creative solutions and customizations to find a good result.

6.1 Validity of Work

Since the research made in the thesis was mainly of a qualitative design, this section serves the purpose of providing arguments to the trustworthiness of the result. This is accomplished by applying a set of quality criteria [3]: *credibility*, *transferability*, *dependability*, *confirmability* and *reflexivity*.

6.1.1 Credibility

The premise of this thesis comes from a quite simple question: is it feasible to use formal specifications in a practical way by computer engineers during porting of computer software? Each step of the process is thoroughly presented in the thesis, making it easy for a reader to follow the process without much left to question. The feasibility was evaluated using a metric decided on beforehand. This evaluation was in the form of the verification test-suite run on both the original code and the port. Because of this, it should be clear that the limited conclusions drawn about the feasibility of the methodology are credible.

6.1.2 Transferability

Since the methodology was thoroughly described using a step-by-step presentation, it is reasonable to consider it possible for other practitioners in the field to use and improve upon the proposed methodology. It should be general enough for other programming languages and types of functionalities being ported since there is nothing about the languages chosen that should impact the usefulness.

6.1.3 Dependability

The dependability of the result should not differ depending on when or by whom an evaluation of the result is being performed. Whether the result will hold stable over time could depend on factors such as an increase in computing power. This could make the process a subject for automation and thereby invalidate any conclusions about the usefulness of the method described.

6.1.4 Confirmability

As a measure of ensuring that this process is repeatable and its results reproducible, the methodology evaluated has been made very explicit. However, as the study was limited in its scope there is a risk that aspects of the methodology were not fully realized which could have prevented the discovery of weaknesses in its execution or structure. This issue could reasonably be considered to have been mitigated to some degree since continuous meetings with the company that proposed the subject were used as performance reviews along during the process. There possible issues were discussed, and solutions and refinement of the methodology was a constant throughout the project.

6.1.5 Reflexivity

Reflexivity is the process of critical self-reflection. Since the authors have no formal background in research it is not unlikely that there might have been some bias and preconceptions during the project. For example, knowing the target and source language early on and our preconceptions about them, could have affected the early phases of the research. With another selection process, other languages might have been chosen and a different answer to our question might have been the result.

7 Conclusion

The result of the project indicates that the method described could be used as a basis when porting legacy code. The practice of creating a model based on tests, gave a solid foundation for building a port. Some cases were found where the method could be extended to consider properties of the source and target language. The target project may also affect the effectiveness of the method. The application of TLA+ as a modelling language was found to be a good choice and comes with the authors recommendations for software engineers trying to acquaint themselves with formal specifications.

All in all, the experience working with the project has been rewarding and has given the authors many tools which will hopefully be useful in future endeavors. It will be interesting to see if formal specifications will enjoy a wider adoption in the future.

7.1 Future Work

It would be interesting to see the method applied to a more complex system. As has been mentioned in the discussion, a more complex system may make better use of a formal specification. If used by more experienced engineers it could also provide a greater insight in the efficiency of the method.

The process described in this thesis explicitly did not consider the run-time characteristics of the system. This decision was taken to limit the scope of the process. However, it may be reasonable to extend the process taking these into account. As an example, in the case described here - the linked list - it is trivial to determine some run-time characteristics of the list. For example, insertions after a certain element are O(1), while inserting before a specific element is O(N). Some benchmarks in complexity as well as absolute run-time performance may be necessary to make a useful port. In fact, these characteristics are likely the motivating factor for choosing a linked list rather than a memory-contiguous collection. This can be especially critical to specify from the beginning, as such characteristics may get lost in the modelling stage. The more mathematical description of the system likely invokes unrealistic run-time costs due to the usage of sets and the permutations of these to create the model states. This was also noted by the Amazon case study. The assessment of the run-time speed characteristics should likely be done in parallel with the writing of the test code, estimating the complexity characteristics by identifying critical control flow operations such as loops and then verifying these assumptions using tests. Estimating and testing the run-time memory characteristics could be more challenging and is left up to future research.

8 Bibliography

References

[1] M. Nami and W. Suryn, "Software testing is necessary but not sufficient for software trustworthiness", vol. 320, Jan. 2013, pp. 34–44, ISBN: 978-3-642-35794-7. DOI: 10.1007/978-3-642-35795-4_5.

[2] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How amazon web services uses formal methods", *Communications of the ACM*, vol. 58, pp. 66–73, Mar. 2015. DOI: 10.1145/2699417.

[3] I. Korstjens and A. Moser, "Series: Practical guidance to qualitative research. part 4: Trustworthiness and publishing", *European Journal of General Practice*, vol. 24, no. 1, pp. 120–124, Dec. 2017. DOI: 10.1080/13814788.2017.1375092. [Online]. Available: https://doi.org/10.1080/13814788.2017.1375092.

[4] M. Weiser, J. Gannon, and P. McMullin, "Comparison of structural test coverage metrics", *IEEE Software*, vol. 2, no. 2, pp. 80–85, 1985. DOI: 10.1109/MS.1985.230356.

[5] J. A. Duregård, "Automating black-box property based testing", ISBN: 9789175974316, Ph.D. dissertation, Chalmers University of Technology, 2016. [Online]. Available: https://research.chalmers.se/en/publication/240807 (visited on 05/22/2021).

[6] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs", *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, vol. 46, Jan. 2000. DOI: 10.1145/1988042.1988046.

[7] K. Beck, *Test Driven Development. By Example (Addison-Wesley Signature).* Addison-Wesley Longman, Amsterdam, 2002, ISBN: 0321146530.

[8] I. Sommerville, *Software engineering*, 9th ed. Boston: Pearson, 2011, OCLC: ocn462909026, ISBN: 9780135140758.

[9] C. A. R. Hoare, "An axiomatic basis for computer programming", *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969. DOI: 10.1145/363235.363259. [Online]. Available: https://doi.org/10.1145/363235.363259.

[10] J. V. Guttag and J. J. Horning, *Larch: Languages and Tools for Formal Specification.* Berlin, Heidelberg: Springer-Verlag, 1993, ISBN: 0387940065.

[11] A. v. Lamsweerde, "Formal specification: A roadmap", in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00, Limerick, Ireland: Association for Computing Machinery, 2000, pp. 147–159, ISBN: 1581132530. DOI: 10.1145/336512.336546. [Online]. Available: https://doi.org/10.1145/336512.336546.

[12] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, and et al., "Using formal specifications to support testing", *ACM Computing Surveys*, vol. 41, no. 2, pp. 1–76, Feb. 2009. DOI: 10.1145/1459352.1459354. [Online]. Available: http://dx.doi.org/10.1145/1459352.1459354.

[13] P. Larsen, J. Fitzgerald, and T. Brookes, "Applying formal specification in industry", *Software, IEEE*, vol. 13, pp. 48–56, Jun. 1996. DOI: 10.1109/52.493020.

[14] L. Lamport, *The TLA+ Home Page*. [Online]. Available: https://lamport.azurewebsites.net/tla/tla.html (visited on 05/28/2021).

[15] C. Newcombe, "Why amazon chose TLA", in *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2014, pp. 25–39. DOI: 10.1007/978-3-662-43652-3_3.

[16] Massachusets Institute of Technology, *Alloy analyzer*, version 5.1.0, Aug. 14, 2019. [Online]. Available: https://alloytools.org.

[17] Microsoft, *Vcc: A verifier for concurrent c*, Jan. 20, 2021. [Online]. Available: https://www.microsoft.com/en-us/research/project/vcc-a-verifier-for-concurrent-c.

[18] *TLA+ Video Course*. [Online]. Available: https://lamport.azurewebsites.net/video/videos.html (visited on 05/28/2021).

[19] L. Lamport, *Tla+ toolbox*, version 1.7.1, Jan. 4, 2021. [Online]. Available: https://lamport.azurewebsites.net/tla/toolbox.html.

[20] H. Wayne, *Practical TLA+: Planning Driven Development*, 1st. USA: Apress, 2018, ISBN: 1484238281.

[21] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev, "Demystifying magic", in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments - VEE '09*, ACM Press, 2009. DOI: 10.1145/1508293.1508305. [Online]. Available: https://doi.org/10.1145/1508293.1508305.

[22] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd. Prentice Hall Professional Technical Reference, 1988, ISBN: 0131103709.

[23] *Introduction - The Rustonomicon*. [Online]. Available: https://doc.rust-lang.org/nomicon/ (visited on 05/28/2021).

[24] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards optimization-safe systems: Analyzing the impact of undefined behavior", in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 260–275, ISBN: 9781450323888. DOI: 10.1145/2517349.2522728. [Online]. Available: https://doi.org/10.1145/2517349.2522728.

[25] J. Gosling, ""java: An overview" the original java whitepaper", Feb. 1995.

[26] R. Hundt, "Loop recognition in c++/java/go/scala", Jan. 2011.

[27] S. Klabnik and C. Nichols, *The Rust Programming Language*. USA: No Starch Press, 2018, ISBN: 1593278284.

[28] *The FreeBSD Project*. [Online]. Available: `https://www.freebsd.org/` (visited on 05/28/2021).

[29] A. Tierno, M. Santos, B. Arruda, and J. Rosa, "Open issues for the automotive software testing", Nov. 2016, pp. 1–8. DOI: `10.1109/INDUSCON.2016.7874609`.

[30] *Google/googletest*, original-date: 2015-07-28T15:07:53Z, May 2021. [Online]. Available: `https://github.com/google/googletest` (visited on 05/28/2021).

[31] emil-e, *Emil-e/rapidcheck*, original-date: 2014-05-08T20:19:43Z, May 2021. [Online]. Available: `https://github.com/emil-e/rapidcheck` (visited on 05/28/2021).

[32] *Cmake(1) — CMake 3.20.3 Documentation*. [Online]. Available: `https://cmake.org/cmake/help/latest/manual/cmake.1.html` (visited on 05/28/2021).

[33] *CLion: A cross-platform IDE for C and C++*. [Online]. Available: `https://www.jetbrains.com/clion` (visited on 06/10/2021).

[34] *GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF)*. [Online]. Available: `https://gcc.gnu.org/` (visited on 05/28/2021).

[35] M. Fredriksson and F. Öberg, *Exjobb repo*, version 0.1.0, May 27, 2021. [Online]. Available: `[https://github.com/magfred-fobe/a-method-for-porting-software-using-formal-specifications]`.

[36] *Gcc(1) linux user's manual*, Jul. 2013. [Online]. Available: `[https://linux.die.net/man/1/gcc]`.

[37] B. Stroustrup, "A history of c++: 1979–1991", *Sigplan Notices - SIGPLAN*, vol. 28, pp. 271–297, Mar. 1993. DOI: `10.1145/155360.155375`.

[38] *Rust Documentation std::option::Option*, 2021. [Online]. Available: `https://doc.rust-lang.org/std/option/enum.Option.html`.

[39] eqrion, *Cbindgen*, version 0.19.0, Apr. 8, 2021. [Online]. Available: `[https://github.com/eqrion/cbindgen]`.