Working Title

Magnus Fredriksson and Fredrik Öberg

Examiner: Leif Lindbäck

KTH Royal Institute of Technology, Sweden

May 9, 2021

Abstract

$E = mc^2$

Contents

1 Introduction

The computer software world is under constant development. Modern languages and tools could make a piece of software considered to be cutting edge technology just a few years ago to be outdated, leading to it needing to be upgraded or replaced. There could also be the situation where a piece of software needs to be adapted to a computing environment different from the one it was originally designed. For example, if the hardware or the operating system(OS) has changed. This state of constant development is not likely to change soon and how a company operating in the software market manages this axiom, could make, or break its entire business.

1.1 Problem Specification

Young Aces By Sylog(YABS) is an IT consultant firm with one of its offices based in Stockholm - Sweden. As consultants they get hired to do a lot of work not normally a 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 This is normally outdated code often severely limiting the performance of the system in question. A problem they have encountered during the code paraphrasing process though, is that the tools they use today not completely verifies that a ported code base have the same functionalities as the original. Since porting is a significant part of their business model, they thereby have formed the question if there could possibly be some working method or tool, they are not using today which could increase their assurance that the ported software indeed does what it is intended to do?

1.2 Premise

The traditional way of verifying the behavior of computer software has been through the practice of testing. This practice has in the past decades generally been about 40-50 per cent of the total development costs of software systems [Software Testing is Necessary But Not Sufficient for Software Trustworthiness] and even with this significant cost does program failures with possible data loss happen a lot. One way to complement the testing process to improve on this situation could be to set up a model of the system through of a formal specification; a mathematically based technique with which a system can be analyzed, and its functionalities be described with the purpose to help 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 the same functionality as any other software conforming to that same specification.

1.3 Goal

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[Use of Formal Methods at Amazon Web Services]. With these interesting observations does this thesis intend to examine if formal specifications also could be used while refactoring legacy code. This is accomplished by taking an existing open-source code base written in C and create a model of the code through a formal specification. This model will then form a base on which a port of the original code to the Rust programming language will be created.

1.4 Purpose

The purpose of the goal of this thesis is to evaluate if formal specifications has the potential to be used in a structured and practical way by computer engineers working with limited resources in a highly competitive market during porting of computer software. This investigation is performed focusing on the method without taking to account its efficiency. This is due to the thesis being conducted during a restricted time frame. This makes it important to limit the scope of the issue, making the area of efficiency open to future investigation.

2 Background

This chapter provides context to the information discussed throughout the thesis. The purpose of this is to enable the intended audience of practicing computer engineers and students of the topic to understand the motivation behind the thesis and its importance. This is accomplished by providing theories, concepts, terms as well as historical data from relevant studies. The chapter also provides a more thorough discussion about the problem statement and the rationale of the thesis as it was described during the introduction.

2.1 Testing

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 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 software, i.e., its state, is being tested. It accomplishes this by testing paths within and between system modules and is usually called white-box, or structural testing [https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=1695302].The normal purpose subjecting any piece of code to one of the testing methods is to verify that the behavior of the system conforms to predefined specifications.

To select a testing method that fits the verification of a developed software best, one should consider the constraints that apply to any given scenario. The

tests must, that is, be able to verify the behavior of the code to such a degree that eventual bugs in the code are accounted for. A bug in this case would be defined as any behavior that does not conform to the original design, implicitly specifying the purpose of the code. This means that the purpose of testing should not be to expose bugs and fix them, but to document the behavior of the implemented system to build an understanding of it. With this understanding could an engineer then hopefully verify that the code does what it is intended to do and that the system design is valid. An implementation conforming to an existing piece of code to this level, could be defined as "bug-complete". This means that it conforms well to original model, rather than the intentions of the programmer.

Since the purpose of testing is to document the behavior of the program, one must define in a clear way how to measure the coverage of the testing. There must also be a definition of what degree of coverage is sufficient. This leads to a significant weakness with testing since the behavior of the tested system can only be verified for those input situations provided in the test data[Software Testing is Necessary But Not Sufficient for Software Trustworthiness]. This means that the tests cannot show that there does not exist any bugs for every possible situation, but only for the given situations. With that must developers avoid assuming that tests for a complex system provides all possible input values, input value sequences and combinations since that is practically an impossibility due to the its scope. Deriving comprehensive test values on such a level of complexity is also a labor-intensive process, increasing the cost of development. The fact that tests become less reliable with an increase in system complexity means that developers most likely will miss something the more complex a system gets. A released version, with that stated, most likely will have an unknown number of bugs. As a consequence of that will a goal of a "bug-complete" system through testing in practice be an impossibility, with increased cost due to troubleshooting and repairing of these bugs a likely scenario.

As an example, would achieving total coverage be a near impossibility and even difficult to define considering even a trivial piece of code. Take for instance a function which produces some side-effects. It is easy to imagine a case where such a function would behave as intended when called a certain application state. It is also plausible that it could misbehave when the state is different, even if the input is the same each time since it performs an operation on a variable not included in the functions return value. This is visualized through the following pseudo-code:

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

The function takes an argument as input – the variable y – and returns it without any mutation. It also performs the operator ++ on the global variable z which

in most languages increments z with one. This situation though, symbolize that any type of operation could be performed on z. From a black-box perspective, one could conclude from even somewhat extensive testing, that the function has a deterministic behavior since the function always returns it input. This leads to the situation where the invariant described by the following property would hold:

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

Even with many test samples may the non-deterministic behavior of the function not be exposed, since the operation on the variable z is not considered in the invariant. Using metrics such as branches taken, or lines executed[1] - common metrics for coverage - would not help in this case. Neither would property-based testing[3] – a way of testing where facts about the code that given certain precondition should always be true - necessarily find the problem. That is, since an exhaustive (consecutive) search of the input domain is necessary, and in many cases not possible depending on what is being tested and how large the input domain is.

If is dangerous draw to great conclusions based on the information provided in this section but it does not seem unreasonable to conclude that testing is necessary but not sufficient for providing software trustworthiness. As a consequence of that should the goal of software verification be to find a reasonable set of tests which verifies a systems behavior for reasonable input domain. The tests should also exercise the programs structural components thoroughly enough, no matter what is correct, or intended behavior[1]. This could be achieved by a combination of black- and white-box testing where the problem is though, how to define how large the scope of the tests should be since the number of inputs to a system could be unlimited. This then is dependent on the skill and experience of the engineers doing the implementation to make the call on what is reasonable. This leads to a conclusion that software verification most likely would benefit from the addition of techniques complementing testing. Techniques dealing with the subject of system verification in a different way.

2.2 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[Larch]. 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 affecting unintended functionalities. To sum it up is the key to good software design inventing appropriate abstractions around which to structure the system.

Formal specifications are mathematically based techniques where formal means that they have a syntax, their semantics fall within one domain, and they can be used to infer useful information[Formal Specification - A roadmap]. 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 it in the implemented software. This benefit could be derived from the fact that engineers focuses on the safety and liveness properties of a system when using formal specifications. They need state what needs to go right instead of focusing on what could go wrong, often a mind set when writing tests[Use of Formal Methods at Amazon Web Services].

Formal specifications encourages this way of thinking by describing an algorithm as a state machine, enabling a designer to create a simple and powerful abstraction of it. A trivial example of this is the following pseudo-code inspired by the works of Leslie Lamport[Leslie Lamport's The TLA+ Video Course]:

```
Global signed z
Function x()
 z = someNumber()
 z = z + 1
```

This function has 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 the program terminating. This process can be described with the following state machine:

This state machine could be described with a somewhat mathematical model in the following way:

$$currentState = Init \land currentState' = Start \land z = 0$$

$$currentState = Start \land currentState' = Increment \land z \in \mathbb{Z}$$

$$currentState = Increment \land currentState' = End \land z = z + 1$$

$$currentState = End$$

This model has the variable currentState, indicating which state is the current one by having an equality between it and a string of text. This model starts with currentState having the value of Init, indicating that the first statement is the first state of the function. Here there are conjunctions between this equality and one where currentState has a prime attached to it. This indicates which value of currentState is in the state following the current one. There is also a conjunction with one statement showing that the value of z is initialized to the

value 0. This is a behaviour of the language C and is used as an example but could differ depending on the language used when implementing the function. If the first statement holds true for all conjunctions then it is followed by the statement where currentState has the value of currentState', in this case Start. With this practice can this model now be evaluated to see if every statement holds true given any situation providing a logical foundation that the implemented program does the same.

Since computer systems becomes 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 has been adopted in more traditional engineering disciplines but is not widely used in industrial software development (Sommerville). One of the reasons is that it by many is a method considered not to be cost-effective to apply the technique. There has also been very little interaction between the test and formal specification communities but a new consensus have appeared where both approaches has been seen as complementary[Using Formal Specifications to Support Testing]. Recent studies have also found that creating a mathematical model of a system under development led them to find and handle exceptional behavior during the system analysis phase[Applying formal spec. in Industry][Use of Formal Methods at Amazon Web Services]. These studies have that the time spent on making a mathematical model was well invested since it led to less time being spent during the implementation part of the development through less bug hunting. This is most likely accomplished since a model with good system-invariants help engineers getting the design right. That is since these invariants captures the fundamental reason why the system works by using them as a safety property which holds for each state of the system. This shows that the design is not broken and helps engineers getting the code right since the code most likely becomes broken if the design is broken. This then could lead to engineers being deceived into the code being "correct" since it passes tests based on an implementation of this broken design.

Formal specifications are not without problems though, and the consensus seem to be that they are most likely to be useful when:

- 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 be implemented correctly.
- complex functionality is involved with many choices to be made or many exceptional conditions arise[Applying formal specification in industry].

To know when and how to use formal specification takes a lot of skill and experience. It for example, does not evaluate the performance of a system and incorrectly used could lead to a sub-optimal design even though there is not logic bug present in the system[Use of Formal Methods at Amazon Web Services]. A feasible way to model a system to predict the unwanted behavior of prolonged

severe slowdowns seem to not be known as of this day. This means that other techniques need to be used combined with formal specifications to mitigate that. The upsides of formal specifications though, seem to be encouraging enough so it would not be surprising if the technique should be more widely adopted in the future.

2.3 Rust

Since it started to gain popularity during the 1980s has the programming language C been a natural choice when working at low level programming. It found lasting use in applications previously coded in assembly since it is a programming language supporting structured programming, lexical variable scope, and recursion. C also provides static type checking enabling verification of the type-safety of a program as well as constructs that map efficiently to typical machine instructions. This provides low-level access to memory. One problem with C though, has always been its manual memory management and the risks this design property brings – a risk many would claim makes C an unsafe language. This comes from the fact that a misuse of manual memory management will cause undefined behavior – the program becomes unpredictable since it gives the compiler full rights to do arbitrary things[The rustonomicon]. Many programming languages' solution to this problem has been for example through layering of virtualization of the developed program and has in many contexts come with the cost of decreased performance.

Rust is a modern system programming language focusing on safety, speed, and concurrency and claims to accomplish these goals while being memory safe without using garbage collection. It is designed to guide the programmer to reliable code that is efficient in terms of speed and memory usage. Designers of Rust also claims that introducing parallelism in Rust is a relatively low risk operation since the compiler is designed to catch classical mistakes before execution. Code written in Rust 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 - at least if Rust is used as intended. This allows for more aggressive optimization in the code to be implemented since they will not accidentally introduce crashes or vulnerabilities. With that, is the purpose of Rust to eliminate the trade-offs that have been accepted when working with C while at the same time getting comparable performance.[The rust programming language]

2.4 Motivation

There is a lot of legacy code out there inherited from older versions of a system, most likely in the need of updates. Why these updates have not been carried out could be, for example, because it is code written in an old language making it hard to understand and/or difficult to change. Tests have also shown that it is a necessary part of software verification but many time not sufficient for understanding a system since bugs are a common sight in the industry. Tests

also take up lot of the development cost so there is potentially a lot to be gained if the verification process could be improved - hopefully also when porting legacy code. If Rust succeeds with its goals and becomes an ubiquitous language then it is likely that it will be a candidate for porting of the large amount of legacy C code out in the world in the future. Thus, the reason of use both languages in this thesis seemed as a valid and interesting choice. Using these languages will also hopefully help answering the question if formal specifications could help verifying that the a port has the same functionality as the original. That is, since they are languages which such disparate design philosophies and the implementations thereby have the potential to look vastly different.

3 Method

The goal of this chapter is to provide a clear and extensive description of the methodology undertaken to address the question the thesis is based on. The intention of this is to enable the methodology to be reproducible. This is accomplished by first providing a description of how the questions were framed by dividing the method into several steps and what each step needed to accomplish for it to be considered finished. The chapter also provides a description of what tools had been used during the project and motivations of why they were chosen.

3.1 Structure

To enable answering if formal specifications could be used during porting processes has the methodology being divided into these steps:

1. Find an established and well used open-source library of suitable size as a work of reference.
2. Find a suitable testing framework and implement tests on the reference work to form an understanding of its behaviors and functionalities.
3. Find a formal specification language and reverse engineer the work of reference by creating a model of it. This model is to be derived from the tests implemented in the previous step.
4. Implement Rust tests for the functionalities specified through the model created.
5. Implement the functionalities in Rust code based on the tests created in the previous step.
6. Verify to a high degree that the created Rust code is a true port of the work of reference.

The definition of a suitable size for the work of reference was set on functionalities the authors of this thesis were well versed in. The purpose of this was so that the project could be focused on the implementation methodology since the resources in the form of time was limited.

3.2 FreeBSD linked list

The reference-work this project was decided to be based on is part of the FreeBSD project[https://www.freebsd.org/]; a Unix-like operating system used to power modern servers and embedded platforms. The choice was made since it is a project that has a large community of individuals and organizations which has continually developed it since the beginning of the 90's. This leads to the code base being well established and its functionalities tried and tested.

The module chosen to port was from its queue header file, containing several macro functions for a set of queue data structures. An intended choice since formal specification is considered most useful when there is a complex data structure that must be handled correctly[korsreferens]. Of these data structure where a singly linked list chosen to be implemented. This is a data structure most would consider to be more trivial than complex. The logic behind this is that it is a well-known data structure and one which the authors are well versed with. This choice would then most likely end up saving the time it would take to learn a more complex one. It would also spare the need for explaining how it works since it should be reasonable to assume that most readers of this paper know how a linked list works. Enabling the project to be of as large of a scope as needed.

3.3 Test tools

When testing computer software, normally a test framework is used. A test framework provides a convenient syntax for writing tests, foremost since they provide 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. There are many frameworks to choose from and it is not obvious that one is necessarily better than another. There are, however, some requirements that a chosen framework should fulfill to satisfy the needs of the project.

The framework must support property-based testing as well as having the ability to measure the coverage of the tests in terms of branches taken to see that the coverage will be sufficient. The choice fell on GoogleTest[https://github.com/google/googletest] as it is a widely adopted unit testing library. It also allows tests to be written in C++ which will enable the C code of the work of reference to be easily tested with minimal modification. Rapidcheck[https://github.co m/emil-e/rapidcheck] was chosen as a framework for the property-based testing since it is easy to use alongside GoogleTest. That is since writing tests in RapidCheck allows for specification of properties in an imperative style which makes sense for C++. This

should also be helpful when translating the properties into the project's formal specification language of choice.

3.3.1 Environment

To execute tests on the work of reference a test environment needed to be set up. The first step in this process was to isolate the files needed by the data structure so that they could be executed outside the context of FreeBSD. As the linked list code chosen is in the lower levels of the operating system, was this an elementary process since there were few dependencies needed to be included. The reference work is fully made up of macros; predefined text snippets that turn into code at the pre-processing stage of compiling. This means that the macros needed to be expanded into compilable code so they could be tested. This was done by using the GCC compiler[https://gcc.gnu.org/] and the -E flag as argument. 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, was a level of indirection added by defining proxy functions using the macros which enabled the coverage to be measured. The added indirection somewhat changes the semantics of how the library was invoked by introducing function calls that would otherwise not be present. Defining these functions as inline made it so that the resulting compiler output was identical, or very close to when generated from using the macros directly. Lastly was the project and dependencies built using CMake[https://cmake.org/] which is an easy-to-use tool for automatic enabling of these type of procedures.

3.3.2 Implementation Process

The initial intention was to create tests in a methodical and structured way. To accomplish this were these steps of deductive reasoning and test implementations created and followed:

1. Select a functionality to write tests for.
2. Provide a logical basis for the proofs of 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. Rewrite tests to match any discrepancies between actual and expected behavior.
5. Repeat all steps until all functionality of importance has been identified and tested.

These steps were based on a set of axioms and rules used in proofs of properties of computer programs[An axiomatic basis for computer programming] thereby, to some extent, prove the correctness and behavior of the program. These logical bases had the following structure:

$$P\{Q\}R$$

This should be interpreted as "if the assertion P is true before initiation or a program Q, then the assertion R will be true on its completion". This deductive reasoning provided a fundamental understanding of the code, enabling an easier reverse engineering process of it.

3.4 TLA+

The choice of formal specification language to use during this project fell on TLA+[https://lamport.azurewebsites.net/tla/tla.html]. This is a language developed by the computer scientist Leslie Lamport and has since 2011 been used by Amazon as a tool to achieve correctness in the sophisticated distributed systems they continually develop[Why Amazon Chose TLA+]. Amazon evaluated TLA+ among other languages like Alloy[https://alloytools.org/] and Microsoft VCC [https://www.microsoft.com/en-us/research/project/vcc-a-verifier-for-concurrent-c/]. All languages had their pros and cons but TLA+ turned out to be best fit for their needs.

They found that TLA+ was best for handling very large complex as well as subtle problems. TLA+ accomplishes this by being able to capture rich concepts simply and directly without tedious work arounds, for example when specifying dynamic sequences of many types of nested records. Details from complex designs can also be added or removed quickly since TLA+ supports arbitrarily complicated data structures with the ability to define powerful custom operators. This means that it is easy to adjust to a suitable level of abstraction making it a good tool when diagnosing bugs and subtle errors.

Another property of TLA+ is that it is considered to minimize the cognitive burden on engineers since it is an untyped language and avoids esoteric concepts which could be difficult to learn. It does this by using conventional terminology since it largely consists of standard discrete math and a subset of linear temporal logic. It also uses a small set of constructs from set theory and predicate logic plus a new but straightforward notation for defining functions. TLA+ also has the advantage of having a simple syntax based on ordinary mathematics meaning that engineers having some experience in that field should have a low learning threshold. TLA+ also intends to avoid ambiguity by having a small amount of syntactic overloading where an operator could be used for different functions depending on the context.

As an example has the state machine created in the background chapter[korsreferens] been translated into pure TLA+(see figure 1). Here has the currentState variable been replaced by the variable pc, which is short for program counter. It is used by the model checker to know which state is the current one

and terminates when the variable is assigned the value "done". The Init label initializes the value of the variables of the model so the model checker knows where to begin. It then moves on to check which state it should begin with by moving to the Next-label and checks which of the disjunctions(indicated by the or operator) holds true. It carries on with this procedure until the final state has been reached. There is also a way to write TLA+ in its integrated development environment(IDE)[https://lamport.azurewebsites.net/tla/toolbox.html] called PlusCal[a pluscal user's manual] which is similar to pseudo-code. This pseudo-code gets translated by the IDE into correct TLA+ and should limit the learning curve of any experienced engineer having at least some basic knowledge of set theory and predicates.

These properties brought up in this section makes Amazon consider TLA+ to provide the best return on investments of the languages they have investigated. The argument is that TLA+ can handle all types of problems and with the help of PlusCal makes them consider it easy to learn. This makes it a logical choice for this project. It will also be interesting to see if the experience of using TLA+ during refactoring differs from how Amazon experience using it for development.

3.5 Re-Implementation

A test-driven development(TDD) methodology was the approach chosen when implementing the formal specification-based model into Rust code, interleaving the development of both tests and code. The TDD used in this project followed these steps:

1. Identify a new functionality in the formal specification.
2. Write tests for this functionality and implement them as automated tests meaning that the executed tests report whether they have passed or failed.
3. Run the tests. Since the tests have not had their functionality implemented they should fail indicating that the tests add something new to the test set.
4. Implement the functionality and re-run the tests. This may involve refactoring existing code to improve them and add new code to what is already there.
5. Once all tests for a given functionality runs successfully, repeat all steps until all functionalities have been implemented. [Software Engineering, 10th Global edition]

TDD as a re-implementation method was chosen since it enables an incremental development and gives a lower-level understanding of what the implemented code segment does. It thus brings some level of validation that the functionality implemented follows the formal specification. This is a level of validation most likely not there if the implementation method would use another approach where tests had been omitted during development.

3.6 Verification

With the ported code in place, it should behave the same way as the original. To verify this will the port and source code be built as two libraries sharing the same application programming interface(API) enabling them to be tested by the same set of tests. This will make it possible to swap out the system-under-test without changing the test-code, thereby serving as an independent verification that the two implementations behave the same.

For the example source code chosen for the study[korsreferens] is this last step of the implementation process likely to be excessive. When porting a more complex system with many states and transitions, 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 would the ported code of this project 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 no issue. If it is not possible to build into a standalone library, it would likely indicate a failure in previous steps of the process.

This verification process requires that the language has interoperability with and can be called by C or C++ since that is the language used by the test frameworks used in this project[korsreferens]. In the case of this project, where C and Rust are the languages used, can interoperability be achieved by creating a foreign function interface(FFI) by declaring matching external functions that call the appropriate library functions. As Rust has no stable API for C++ must the interface be called from C, although other parts of the test code may be written in C++. The intention then becomes to reuse the original property-based tests with the two linkable libraries. The cbindgen tool[https://github.com/eqrion/cbindgen] will be used to automatically create the necessary C headers from the external function declarations.

As well as serving as the final evaluation of the final product, whether this last step is successful will also serve as an evaluation of the method proposed in this thesis. If it is successful then it can be concluded that this method is a feasible way of verifying ported software. There is one caveat to this last step of the 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 as a consequence invalidating the conclusion of the project.

4 Result

5 Discussion

6 Conclusion