# Dissertation Title

Enda O’Shea

Dissertation 2017

Erasmus Mundus MSc in Dependable Software Systems



Department of Computer Science,

Maynooth University,

Co. Kildare, Ireland.

A dissertation submitted in partial fulfilment

of the requirements for the

Erasmus Mundus MSc Dependable Software Systems

Head of Department : Dr Adam Winstanley

Supervisor : Dr Rosemary Monahan

15-June-2018

Word Count: 0000

Contents

[Declaration i](#_Toc447110597)

[Acknowledgements ii](#_Toc447110598)

[Abstract iii](#_Toc447110599)

[List of Figures iv](#_Toc447110600)

[List of Tables iv](#_Toc447110601)

[**Chapter one: Introduction** 1](#_Toc447110602)

[Summary 1](#_Toc447110603)

[1.1 Topic addressed in this project 1](#_Toc447110604)

[1.2 Motivation 1](#_Toc447110605)

[1.3 Problem statement 1](#_Toc447110606)

[1.4 Approach 2](#_Toc447110607)

[1.5 Metrics 2](#_Toc447110608)

[1.6 Project 2](#_Toc447110609)

[**Chapter two: Related Work** 3](#_Toc447110610)

[Summary 3](#_Toc447110611)

[2.1 Topic material 3](#_Toc447110612)

[2.2 Technical material 3](#_Toc447110613)

[**Chapter three: The Problem** 4](#_Toc447110614)

[Summary 4](#_Toc447110615)

[3.1 Project UML documentation 4](#_Toc447110616)

[3.2 Problem analysis 4](#_Toc447110617)

[**Chapter four: The Solution** 6](#_Toc447110618)

[Summary 6](#_Toc447110619)

[Depending on your type of project, you may not need to include all of these: 6](#_Toc447110620)

[4.1 Analytical Work 6](#_Toc447110621)

[4.2 Architectural Level 6](#_Toc447110622)

[4.2 High Level 6](#_Toc447110623)

[E.g. Packages, Class Diagrams, etc. 6](#_Toc447110624)

[4.2 Low Level 6](#_Toc447110625)

[E.g. Method specifications, Algorithms, etc. 6](#_Toc447110626)

[4.2 Implementation 6](#_Toc447110627)

[**Chapter five: Evaluation** 7](#_Toc447110628)

[Summary 7](#_Toc447110629)

[5.1 Solution Verification 7](#_Toc447110630)

[E.g. use your equations to verify the correctness of your solution 7](#_Toc447110631)

[5.2 Software Design Verification 7](#_Toc447110632)

[5.3 Software Verification 7](#_Toc447110633)

[5.3.1 Your test approach (i.e. unit testing, sub-system testing, system testing) 7](#_Toc447110634)

[5.3.2 Your tests (e.g. scenarios, test cases, test data, etc.) 7](#_Toc447110635)

[5.3.3 Your test results 8](#_Toc447110636)

[5.3.4 An interpretation of the results 8](#_Toc447110637)

[5.4 Validation/Measurements 8](#_Toc447110638)

[5.4.1 Results 8](#_Toc447110639)

[5.4.2 Explanation of Results 8](#_Toc447110640)

[5.4.3 Analysis of Results 8](#_Toc447110641)

[5.4.4 Comparison with previous solutions (if relevant) 8](#_Toc447110642)

[**Chapter five: Conclusion** 9](#_Toc447110643)

[**Summary** 9](#_Toc447110644)

[5.1 Contribution to the state-of-the-art 9](#_Toc447110645)

[5.2 Results discussion 9](#_Toc447110646)

[5.3 Project Approach 9](#_Toc447110647)

[5.3 Future Work 9](#_Toc447110648)

[**References** 10](#_Toc447110649)

[**Appendices** 11](#_Toc447110650)

[Appendix 1 Schematic of the hardware associated with this project. 12](#_Toc447110651)

[Appendix 2 Code developed for this project. 13](#_Toc447110652)

[Appendix 3 UML Class, Use Case and sequence diagrams for this project. 14](#_Toc447110653)

[Appendix 4 Screen shots of the project implementation 15](#_Toc447110654)

[Appendix 5 Taught M.Sc. Dissertation Guidelines (valid from Oct 2015) 16](#_Toc447110655)

## Declaration

I hereby certify that this material, which I now submit for assessment on the program of study as part of Master of Science in Dependable Software Systems qualification, is *entirely* my own work and has not been taken from the work of others - save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: Date:

## Acknowledgements

Rosemary Monahan

David R. Cok

## Abstract

Formal specification and software verification have become increasingly pertinent in the past decade as a way of supplementing the already popular software testing techniques, to both improve software quality and provide a more concrete proof of reliability. However, the use of these proof techniques has not been wholly adopted by industry due to business factors such as the time required for specifying the source code and costs related to such a process, along with more technical factors such as the difficulty in specifying and verifying code with the current tools and languages available.

In this project we will be focusing on a verification tool called OpenJML, developed by David R. Cok with Java as its target language, that set out to simplify the development of specifications, using the JML language, and simplify the verification process, using SMT provers, with the overall goal of wide adoption by industry professionals. This project sets out to examine the updated version of this tool to see if a novice user can adopt the techniques required to specify and verify pieces of software. We plan to determine OpenJML’s validity as an industry alternative in comparison to similar existing verification tools and to examine its performance as a standalone specification and verification tool.

**Category, Terms, Keywords: OpenJML, Formal Specification, JML, KeY, Why3, Deductive Verification**

**NB: See Appendix 5 for the official guidelines on how to write this thesis.**

## List of Figures

[Figure 3‑1 UML class diagram overview for this project. 4](#_Toc445718606)

## List of Tables

[Table 2‑1 Table of interest: Aspect of your implementation 2](#_Toc445714278)

[Table 2‑2 Data sources used in your implementation 2](#_Toc445714279)

# **Chapter one: Introduction**

## Summary

Chapter 1 describes….

**Introduction – a high level description of the research question and the problem domain that can be understood by somebody new to the subject area.**

* + **Objectives –** A single sentence that describes the purpose of this section.
  + **Research Question** – State the technical problem that you have focused on in your project in the form of a question which you address.
  + **Motivation** – Discuss the reasons for solving this problem. Detail the problem domain and who would be interested in the solution. Describe the likely impact of your work. Address both why it is an interesting technical problem, and also the value of solving it in more general terms.
  + **Aims and Objectives** – State the aims and objectives of your project. The **aims** of your project are the overall goal, and the **objectives** are the stepping stones in reaching that goal. Identifying the objectives helps the reader to understand your overall project approach.
  + **Report Structure** - Outline the structure of the report summarizing each chapter in one sentence.

## 1.1 Overview

Formal specification and software verification of software have become increasingly pertinent in the past decade, as a way of supplementing the already popular software testing techniques, to both improve software quality and provide a more concrete proof of reliability. This lead to the Programming by Contract approach that was popularised by Bertrand Meyer (*Meyer, B. 1992*) , however was presented in earlier works (Insert Works?), with the overall goal being to reduce defensive programming and increase reliability by introducing mathematical proofs into a methods specification, therefore enforcing the clients and suppliers compliance (InsertQuote?).

However, the use of these proof techniques has not been wholly adopted by industry due to business factors such as the time required for specifying the source code and costs related to such a process, to the more technical factors such as the difficulty in specifying and verifying code with the current tools and languages available, with an expert in the domain often required to get valid implementations.

## 1.2 Motivation

‘VerifyThis’ (Pm.inf.ethz.ch. (2018) is a program verification competition that requires contestants to specify and verify a certain number of tasks within a certain time limit, usually 45 minutes per question. The winners of these competitions in the past five years, 2018 included, were teams that used the verification tools Isabelle, Why3, KIV, Verifast with KeY and Dafny also proving popular. These tools, with the exception perhaps of Dafny, are non-intuitive by nature and require vast amounts of expertise and skill to master with no regular cross-over functionality between them or interface to connect them (Source).

The developers of these tools do not communicate regularly (Source) with each other and focus primarily of developing their own tool’s functionality. This lack of co-ordination has led to many different tools that, even though proven to work, are not adopted by many users outside of their field. Novice users, just coming into the formal verification domain, especially have a steep learning curve with separate libraries and syntax variables to conquer while trying to embrace the core concepts of Programming by Contract. This lack of co-operation and co-ordination has increased the delay of verification being adopted outside of academia with industry primarily focused on developing software products in a timely, cost effective matter. Ensuring reliability is paramount to all software development projects, however the time and expertise required for integrating one of the verification tools above seems to be too much for industry to handle and relies primarily on the proven but not fully sound software testing techniques (Source).

OpenJML aims to bridge this gap (Source) by allowing its freely available tool to be integrated into the Eclipse IDE directly and using only the popular JML specification language with sequential Java programs. A command-line tool is also available and the overall goal of the tool is simplicity for novice and expert users alike. This project aims to evaluate how easy in fact it is to use this tool in comparison to its competitors, KeY also has an Eclipse plugin, and if the stripping down to just the basics of JML with Java would be viable for real-life industrial systems.

## 1.3 Objectives

The main goal of this project is to determine how effective OpenJML can be at specifying software programs with its use of JML and determine if the verification tool can provide adequate and accurately valid results for said specifications. We set out to solve programs from the VerifyThis 2012 competition, specifically the PrefixSum and Longest Repeating Substring questions, as they have been specified and verified by other tools with a clear benchmark in place for comparison.

From the OpenJML Deductive Verification process we hoped to determine its difficulty, adaptability and usability in working with these programs and therefore determine its validity in comparison to other similar tools such as KeY and Why3.

We also would provide feedback and data to the developer, David R. Cok, of the OpenJML tool as we progress through our implementations; reporting issues, bugs and specification difficulties for both assistance and possible recommended updates that may be required. Our overall goal was to determine if the OpenJML tool is complete enough to replace all other verification tools and streamline the formal verification academic area to focus solely on this tool moving forward reducing complexity for users and with the hope of widespread adoption in both academia and industry alike.

## 1.4 Approach

Summarise how you addressed solving the problem.

Provide an overview of how you analysed the problem, how you designed a solution, and how you evaluated your solution. (e.g. use of models, simulation, prototypes, real-world experiments, cases studies, etc.). What important variables did you control, ignore, or measure in your evaluation.

PrefixSum

Longest Repeating Substring

We started the process by selecting two programs from the VerifyThis 2012 competition, PrefixSum and Longest Repeating Substring.

## 1.5 Metrics

Describe how you are going to evaluate your work.

Comparison to KeY and Why3 tools on similar programs.

* Lines of code
* Specification differences
  + Difficulty
  + Adaptability
  + Usability
  + Validity
* Libraries
* Proof Obligations
* Proof Discharges
* Symbolic Execution vs VCG
* Standalone tools vs Plugins
* Valid proofs

## 1.6 Project

List, and briefly describe your significant achievements in the project (probably 3-5 of these in a typical project). If you have come up with any contributions

# **Chapter two: Related Work**

## Summary

The purpose of this chapter is to show your depth and breadth of reading and understanding of the problem domain

**Related Work – Details what others have done that is relevant to your work.**

* + **Objectives –** A single sentence that describes the purpose of this section.
  + Describe the context of the research question in detail, defining terminology, and with references.
  + Explain how the problem, or related problems, has been solved previously. Critically analyze existing solutions. Discuss how your approach compares to these solutions.
  + Explain other techniques that you have used to: help understand and analyze the research question; motivate your own work; evaluate your solution.

## 2.1 Topic material

Literary Review

Deductive Software Verification is the process of developing specifications that can be mathimatically proven to show a program functions as intended. The program, along with its specifications, are turned into formulas which can then be proven (*Filliâtre, J. (2011))* using inference rules applied to sets of axioms determined by the programming language used as well as the logic applied *(Hoare, C. (1983))*. Deductive verification is primarily employed for transitional systems using Hoare Triples to model the input-function-output structure ( {P}S{Q} *(Hoare, C. (1983))* ) and can be performed on programs with an infinite state space. There are many types of logic available today however the main logics we focus on regarding deductive verification are first-order preidcate logic, propositional logic and Hoare logic.

First-Order Predicate Logic is a basis for constructing formulas from using “symbolic structures comprised of predicates, functions, variables, constants, quantifiers and logical connectives” (*Yang, K.H., Olson, D. & Kim, J. (2004))*  and is the basis for many verification systems in use today, albeit expanded to include such theories as equality, linear arithmetic, purely applicative arrays and bit vectors (*Filliâtre, J. (2011))* .

Propositional Logic is a simplified version of predicate logic by assigning a true or false value to each variable in a formula and defining the connectives through truth tables (*Yang, K.H., Olson, D. & Kim, J. (2004))*. This logic is popular when constructing normal forms to be used in SAT solvers, discussed later in this chapter, to determine the validity of formulas, an example of which being the construction of Conjunctive Normal Forms to be used in the DPLL framework (*Nieuwenhuis, R., Oliveras, A. & Tinelli, C. (2006))*.

Hoare Logic was proposed by Tony Hoare in his paper “An Axiomatic Basis for Computer Porgramming” *(Hoare, C. (1983))*, and introduced applying deductive reasoning as a way to formally reason about and develop software programs, thats could be mathimatically proven to function as required. Inference rules and axioms were developed to reason with computer programs, such as the widely used {P}S{Q} notation stating “*If proposition P is true when control is at the beginning of statement S, then proposition Q will be true when control is at the end of statement S*” *(Hoare, C. (1983))*. These inference rules were applied using assertions to ensure all programs satisfied these inference rules.

This contributed to the paper by Bertrand Meyer, “Design by Contract” *(Meyer, B. (1992))*, which introduced creating contract specifications for each method through the use of pre and postconditions. This introcued the concept of a client and supplier for each method contract with the former being the user that calls the method and the later supplying the implementation usually through an interface. It stated that each precondition must be satisifed by the client with the supplier ensuring the postconditions are satisfied upon the methods execution, therefore satisfying the contract. This paper also talked about loop invariants, an assertion that must hold before, during and after a loops execution, class invaraints which “*must be preserved by every exported routine of the class. Any such routine must guarantee that the invariant is satisfied on exit if it was satisfied on entry*” *(Meyer, B. (1992))*, as well as the implications of inheritance, based on enforcing behavioural subtyping from the Principle of Substitutivity theory, with rules stating that preconditions cannot be strengthened and postconditions unable to be weakened. All public class invariants can be inherited by the subclasses during inheritance and they may use them as is or strengthen them if required as well as creating their own invariants when needed. All of these assertions introduced by Meyer require checking prior to the contract being assessed with a mechanism called Runtime Assertion Checking used to ensure no violations occur within the assertions themselves.

Runtime Assertion Checking main application is debugging programs by translating assertions into runtime checks to see if any violations occur during program execution *(Cs.ru.nl. (2018a))*. If any assertion violation occurs an error is produced, “*providing information about the cause of the problem, rather than the consequence*” *(Cs.ru.nl. (2018b))*. RAC applied alongside testing provides an easy and cheap process for checking that the program works as intended, “*tests agaisnt what a developer thinks their software does versus what it actually does*” *(Cs.ru.nl. (2018b))*. Once this process has been completed, static verification tests can commence on the specifications to ensure that the contracts for each method can be satisfied. One of the earliest tools used for this process was called ESC/Java2 which applied extended static checking to Java programs annotated with the Java Modelling Language.

ESC/Java2 was an extension of the ESC/Java tool that supported more JML functionality with the goal of proving correctness of the specifications at compile time *(Cok, D.R. & Kiniry, J.R. (2005))*. The tool operates on a modular basis, taking each method individually, using fully automated verification when proving correctness of specifications at compile time and is very useful for finding potential bugs early and proving the absense of runtime exceptions. However it cannot prove soundness, may miss an error that is present, or completeness , may warn of errors that are not possible, (*Kiniry, J., Morkan, A. & Denby, B. (2006))* , due to the engineering limitations imposed on ESC/Java2, as it is not a fully fledged verification tool but rather an additive to RAC and testing procedures used by programmers.

The structure of the ESC/Java2 tool is split into three steps *(Cs.ru.nl. (2018c))*:

1. Parsing Phase
   * Used to check the syntax of the code and specifications
   * Produces cautions and errors
2. Type-Checking Phase
   * Type and usage checking of the code and specifications
   * Produces cautions and errors
3. Static Checking Phase
   * Reasoning to find bugs by converting assertions to verification conditions (VCs) and then using an SMT prover called Simplify to check for correctness of these VCs
   * Produces warning of what caused the error
   * Produces a counter-example with a data model showing how error occurred

The Java Modelling Language is a “*behavioural interface specification language*” used for annotating Java program interfaces and classes, as used by ESC/Java2 as well as various other deductive verification tools such as KeY and Krakatoa, and has evolved continually since its introduction *(Leavens, G. T. , Baker, A. L. & Ruby, C. (1999))*. It is used for specifying the behaviour of a software module as opposed to a whole program and is used by the client to ensure they operate the modules correctly, while the supplier ensures they function correctly as discussed earlier by Meyer. JML was designed to provide programmers with a simplified specification language that avoided “*heavy use of mathimatical operators” and use of assertions that are specific to the underlying programming language*” and instead used a “*side-effect free subset of Java’s expressions to which are added a few mathimatical operators such as the quantifiers \exists and \forall*” whose inclusion incorporated the first-order predicate logic into the language and “*hides mathimatical abstractions, such as sets and sequences, within a library of Java classes*” *(Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C. & Cok, D.R. (2005))*. The overall goal of the JML language was to provide a “*provide a common notation for both formal verification and runtime assertion checking that gives the users the benefit of several tools without the cost of changing notations*” *(Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C. & Cok, D.R. (2005))*.

The JML syntax and capabilities changed based on the verification tool it was being used for, leading to a large subset of JML versions being designed by developers to be optimised for their specific verification tool. This has reduced the capability of users to easily change verification tools due to these JML changes and has resulted in OpenJML being developed with the goal of declaring one universal JML version that can then be developed in unison will all other verification tool developers to ensure parity in all future JML versions. This requires incorporating as many JML variable into the OpenJML tool as possible, as well as providing an automatic verifier that can handle complex specifications and programs, in order to convince the other verification tool developers that it is worthwhile swithcing to this system.

An example of JML, the OpenJML version, is shown in Figure x and is used to specify a module involving a loop. This covers the basic JML structure with each keyword explained on a line by line basis however does not cover all of JML’s functionality or the adaptions of JML.

**JML Example**



Figure x: KeY Array-Search Loop Example

Line 3: normal behavior keyword indicates that if the method functions correctly, the following specifications have to hold

Line 4: requires keyword states the precondition of the contract that must be satified by the client for the method to execute correctly

Line 4: a != null is the constraint placed on the precondition that states the array ‘*a*’ must not be null.

Line 5: ensures keyword states the postcondition that must be satisfied by the execution of the method implemented by the supplier.

*(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))*.

Line 5: \result keyword states the result of the method after execution, which in this method’s case will hold a boolean value of either true or false

Line 5/6:\result == (\exists int i; 0 <= i && i < a.length; a[i] == val); is the constraint put on the postconidition that states if it is true that the value exists in the array ‘*a*’, then the method should return true into the \result parameter and vice versa if no match was found.

Line 10: maintaining keyword represents a loop invaraint (called loop\_invariant in KeY JML version) that must hold before, during and after the execution of a loop

Line 10: !(\exists int j; 0 <= j && j < i; a[j] == val); is the contraint applied to the loop\_invariant that relates to the while loop on Line 14, indicating that the previous index searched (index ‘*j*’) of array ‘*a*’ did not match the value passed into the ‘*search*’ method

Line 11: maintaining 0 <= i && i <= a.length; is another loop\_invariant that breaks the guard of the postcondition set in the ensures clause (i < a.length 🡪 i <= a.length) to indicate that the method has executed fully and the loop is finished.

Line 14: decreasing keyword represents the loop variant (called decreases in KeY JML version) that ensures that the loop terminates by reducing with each loop iteration

Line 14: a.length – i; states that with the counter ‘i’ increasing with every iteration (Line 17: *i++)* it will eventually break the ‘*while(i < a.length)’* statement ensuring loop termination if ‘*val*’ is not found.

One such adaption of JML was the addition of specification only variables called “model” and “ghost” (*Leavens, G. T. , Baker, A. L. & Ruby, C. (1999))* ,that do not form part of the Java code. The model type can be either an abstraction only variable/method used to help the specification or respresent the value of a concrete variable in the Java code which updates in sequence when the Java variable value changes. A represents clause is commonly used when assigning a Java variable to a model type and this is commonly used to preserve encapsulation as well as provide developers with the opportunity to change implementation details without altering the public interface available to clients *(Leavens, G.T., & Cheon, Y. (2003))*. Ghost variables are similar to model variables in that it is used only within specifications but does not use a represents clause for setting its value, instead its value is set either during initialisation or later using the “set” keyword (*Leavens, G.T., Poll, E., Clifton, C., et al., (2013))*.

Another major adaption was the introduction of additional quantifiers to represent a subset of commonly used mathimatical operations. The additions were the \product, \sum, \max and \min quantifiers which could be used in conjunction with the first-order predicate quantifers \exists and \forall to ease the specification process. However, not all verification tool currently implement these additions to the JML library, such as Krakatoa and OpenJML, resulting in specifications being much more complicated to construct within these tools. Also the conjunction of quantifers can result in longer proof statements being created which Verification Condition Generation can struggle with, resulting in the KeY tool gaining an advantage due to its use of Symbolic Execution with regards to simplifying specification. The use of JML within these tools as a result of these and other parameters, such as helper methods, pure methods and spec\_public to name a few introduced in the JML tutorial (*Leavens, G.T., Poll, E., Clifton, C., et al., (2013))*, can prove both freeing and constraining based on the version being employed. One of the original JML goals was to create proofs that could be reused (Burdy, L., et at. (2005)) ,however this will be become increasingly difficult with developers working only on their own versions of JML and no unifying language being proposed and agreed upon.

Once the specification has been completed in their tool, the JML annotations along with the Java code are then translated into an intermediate verification language that is passed to the automatic program verifier, of choice used by the tool, in order to generate Verification Conditions (VC). “*Intermediate Verification Languages (IVL’s) exist as a way to encode computer programs into a common language while maintaining (only) the important logical and stateful properites of the original program*” (*Segal, L. & Chalin, P. (2012))*. IVL’s are used to create an abstaction of the program, regardless of the programming language used, that can then generate Verification Conditions to be discharged by the theorom provers. Translating the programming and specfication languages to IVL’s allows for a further consistent and repeatable translation to VC’s. A common IVL used is called Boogie, which takes converts multiple different languages such as Dafny, Spec#, Java with JML and Eiffel into an abstract language to later be translated into VC’s (*Segal, L. & Chalin, P. (2012))*. Another IVL is called Jessie which takes Java and FramaC languages to translate them to WhyML in the Why Tool with the WhyML then translated to VCs using VCGs *(Krakatoa.lri.fr. (2018b)).*

The “*dominant approaches for the construction of automatic program verifiers are Verification Condition Generation (VCG) and Symbolic Execution (SE)*. *VCGs use a programming calculus such as Weakest Condition Calculus to compute one VC per module, this VC must hold all available knowledge required to prove the correctness of said module*” *(Kassios, I.T., Müller, P. & Schwerhoff, M. (2012))*. This results in VC’s becoming large and uninterpretable to humans and, although theorom provers can apply optimization techniques to the VC, the theorom provers may be unable to process them adequately without some interactive direction from the user *(Kassios, I.T., Müller, P. & Schwerhoff, M. (2012))*.

Symbolic Execution (SE) uses symbols to replace the concrete values to provide a higher level of abstraction to derive the proof against. Branches are determined based on ‘*path conditions*’, such as if statements or loops, and each paths’ validity is determined, with invalid paths removed from the search space in future runs of the same method *(Kassios, I.T., Müller, P. & Schwerhoff, M. (2012))*. The symbolic execution also prunes the search space based on learnt clauses which are created when a conflict is found in an execution path in order to stop a search of this path again. This increases efficiency and search speeds when determined satisfiability *(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))*. This process is similar to the DPLL algorithm applied for most SAT solvers, for more information on this please see reference (*Nieuwenhuis, R., Oliveras, A. & Tinelli, C. (2006)).*

“*Verification Conditions are logical formulas whose satisfiability implies program correctness, and the satisfiability check can be performed, if at all possible (because, in general, the problem of verifying program correctness is undecidable), by using special purpose provers or Satisfiability Modulo Theories (SMT) solvers*” *(De Angelis, E., Fioravanti, F., Pettorossi, A. & Proietti, M. (2017))*. One of the original standalone theorom provers was Isabelle/HOL which was implemented in the functional programming language ML (Meta Language) which focused on interactive theorom proving in higher-order logics *(Nipkow, T., Paulson, L.C., Wenzel, M. (2006))*. Another standalone interactive theorom prover was the Prototype Verification System (PVS) whose formal system was based on sequent calculus with a typed higher-order language (*Bernardeschi, C. & Domenici, A. (2016))*. These standlaone theorom provers then gave way to the more popular Satisfiability (SAT) solvers and SMT solvers which were used by verification tools such as KeY, Why3 and OpenJML as external provers, allowing the user to create programs with specifications in non-functional languages such as Java and C++.

All Verification Conditions are logical formulas and as such can be modelled in propositional logic, which assigns a true or false value to each variable in the formula. A ‘Decision Procedure’ determines whether a formula is valid, returning a true or false answer and can be either sound or complete. A sound decision procedure is a valid formula that is in fact valid and not a false response, while a complete decision procedure will be valid for all available inputs. These terms will be used later in this paper to determine the validity of the proofs generateed by certain tools, especially those using first-order arithmetic due to this arithmetics incompleteness *(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))* resulting in an inability to get sound or complete proofs.

SAT solvers will try to find a propositional model where a formula is satisfiable (true for some model) or valid (true for all models), else returning unsatisfiable (true for no model). There are two approaches for determining validity with SAT solvers, the first of which is the eager approach which will translate the formulas into propositional Conjunctive Normal Form which is then checked by the SAT solver for correctness, with the second version being the lazy approach which uses a DPLL framework, (*Nieuwenhuis, R., Oliveras, A. & Tinelli, C. (2006))*, for determing if a propositional model of the formula satisfies the theory, pruning the search space as it goes to remove invalid models (*Ganzinger H., Hagen G., Nieuwenhuis R., Oliveras A., Tinelli C., (2004))*. An example of an symblic execution process working on multiple paths is attached in Appendices Figure SE.

Satisfiability Modulo Theories solvers extends SAT solver by using the DPLL framework to solve a propositional abstraction of the problem and using algorithms in theory specific solvers for concrete areas, that are not covered by this abstraction process, such as uninterpreted fucntions, array logic, quantified formulas and linear arithmetic. The combination of these algorithms for the concrete areas “*allow SMT solvers to prove formulae using a more expressive range of logical thoeries than propositional logic*” (Healy, A.(2016)) as well as “*providing extended static checking, predicate abstraction, test case generation and bounded model checking over infinite domains, to mention a few*” (*de Moura, L. & Bjørner, N. (2008))*. Z3, Alt-Ergo and Coq are examples of SMT Solvers, each having their own strengths and weaknesses. For example, “*z3 has a unique and effective approach to reasoning about quantifiers, while Alt-Ergo produces excellent results for VC’s containing polymorphic typ*es” (Healy, A.(2016)). Verification tools use these provers differently, with KeY and OpenJML only allowing the selection of one SMT prover per program and Why3 allowing the use of multiple different SMT solvers within the one program providing this tool with a significant advantage over its competitors.

Model checking, as opposed to deductive verification, is performed on reactive systems that often have no termination point and use temporal logic for specifying their requirements. The two types of temporal logic employed are Linear Time Logic (LTL), “*a property of a computation sequence*” , and Computation Tree Logic (CTL), a brachning logic where “*every temporal operator has to be preceded by a path quantifier, and hense such a formula expresses a property of a computation tree*” (*Maidi, M. (2000))*. Model checking applies algorithms to Kripke structures, a collection of first-order structures describing finite state space and representing the variables and their values as states, to determine if any model of the system can satisfy the formulae by checking the correctness of the temproal logic patterns. KeY uses JavaDL as its basis of logic and it evaluates formulas in a Kripke structure. A valid path through a Kripke structure could be an infinite sequence of transitions through states for which a formula can hold *(Kuhtz, L. & Finkbeiner, B. (2011))*. Safety, nothing bad happens, and liveness, something good happens, are two crucial aspects of a Kripke structure ensuring the model functions correctly, with deadlock-freedom also an ideal characteristic.

### Why3 Verification Tool

Why3 is a standalone deductive verification tool that provides a framework for the use of different specification languages in creating program contracts, and the interleaving of different and use of multiple external SAT solvers and SMT provers for the process of proving a program mathematically valid.

The Why3 tool comes with built-in libraries and logical theories for basic operations, such as integer arithmetic, as well as the ability to create axioms, lemmas and predicates for further precise specification requirements. WhyML is the primary intermediate language used in the Why3 framework for verifying C, Java and Ada programs in a similar fashion to the Boogie language for Spec#, Dafny and other specification languages (*Felleisen, M., Gardner, P. & SpringerLink (Online service) (2013)*).

The WhyML language is built upon the mathematical language ML, a first-order predicate language used primarily for sequential programs, with no memory model so static names are given to all variables during proof obligation generation. This results in no mutable components being allowed in recursive methods with the inductive properties required being exported to lemmas and/or predicates (*Felleisen, M., Gardner, P. & SpringerLink (Online service) (2013)*). For more information regarding the WhyML syntax and semantics, please refer to the paper *"Let's verify this with Why3"* (*Bobot, F., Filliâtre, J., Marché, C. & Paskevich, A. (2015)).*

*(Key-project.org. (2018a))*

Figure 1: Why3 Platform

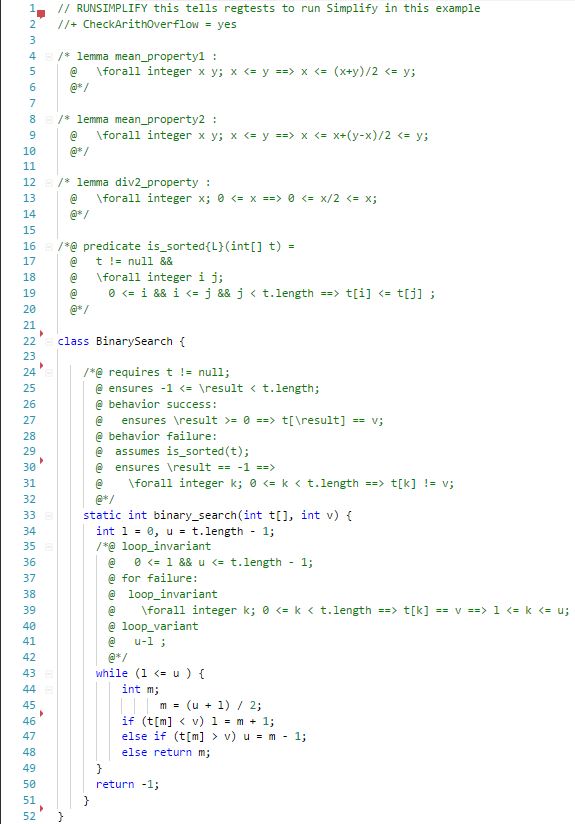
The Why3 tool uses both automatic and interactive theory proving with the ability to use a variety of theorem provers to prove logical goals. Verification Condition Generators (VCG) is the process used to create proof obligations which uses weakest precondition calculus to collectively transform programs and their properties into one large proof obligation which then must be discharged using the theorom provers either automatically or interactively from the user (*Burns, D., Mostowski, W. & Ulbrich, M. (2015)*.

The Why3 framework provides the capability to use a multitude of different front-ends for specifying programs written in different languages. While there are multiple front-ends such as Frama-C, Spark2014 and EasyCrypt *(Key-project.org. (2018a))*, we will be focusing our efforts on the Krakatoa front-end due to it being the platform for Java programs with JML specifications. Krakatoa was developed to verify sequential Java programs however, a particular focus was put on verifying Javacard programs which used short programs that required high levels of confidence (*Marché, C., Paulin-Mohring, C. & Urbain, X. (2004))*. Javacard programs have a smaller language scope than main Java programs and due to the need for all specifications to work for both Java and Javacard programs, the JML used throughout the specifications had to be limited to what was common for both languages. This resulted in a very basic version of JML being used in Krakatoa, with only the core types and quantifiers supported.

Quantifiers such as \sum and \product were not supported, however the ability to create lemmas and predicates to substitute in such functionality is provided and increases the user’s ability to create specifications for more complicated proofs. The development of Krakatoa went on until the Why tool was at version 2.3, however once the Why3 framework was released, future development was focused on the WhyML language specifically due to its larger syntax and ability to make more precise specifications to cover more complicated proof problems. ‘*Krakatoa now has the option of generating intermediate code for the Why3 VC generator*’ ensuring that the system can still be used, however the development of the tool itself has been stopped *(Krakatoa.lri.fr. (2018b))*. Adoption of the Why3 tool now requires learning the WhyML language which can be quite complicated for beginners and those used to standard programming syntaxes such as those used in the C and Java languages.

*(Krakatoa.lri.fr. (2018b)).*

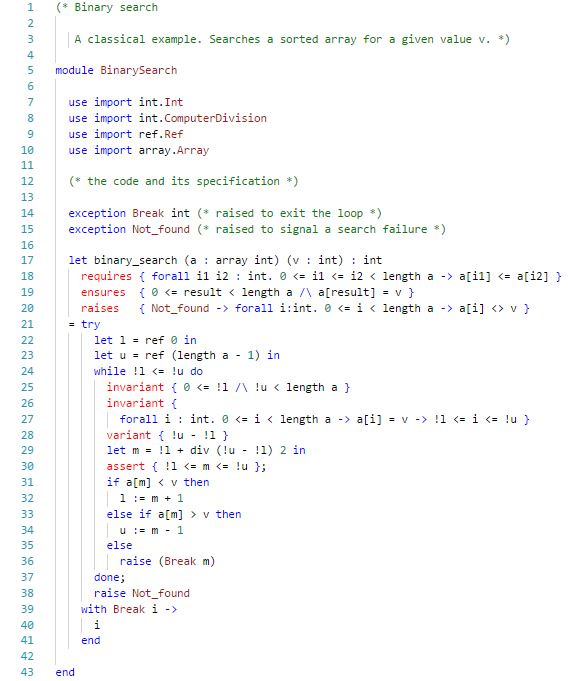
Figure 2: Krakatoa with Why tool

All front-end tools allow for the creation of programs and specifications in their preferred languages, however in the end they all get translated to the WhyML language before being turned in proof obligations for the SAT solver or SMT provers. The Why3 tool provides a PO-discharging back-end that can either automatically determine the correct SAT solver or SMT prover to use for the specification or provide an interactive option allowing the user to choose a solver/prover for each certain specific section of code (*Healy, A. (2016))*. This ability to choose different theorem provers and satisfiability solvers provides an advantage over most other verification systems due to its ability to select provers that can handle different program characteristics (e.g mathematical constructs, recursion, linear arithmetic) and discharge all the proof obligations while other verification systems would only be able to choose one solver/prover per program resulting in only partial proof correctness.

The wide adoption of Why3 may however be restricted due to the limited JML library that the Krakatoa can use as well as the complexity of learning WhyML. In the examples referenced we show the differences in specifying the Binary Search algorithm (source), a simple search algorithm that takes an ordered list and prunes the search space using a middle marker and then moving the cursor left or right of the marker based on the value, repeating this process until the value is found. The Krakatoa version *(Toccata.lri.fr. (2018a))* can be completed in 60 lines of code with all comments removed with approximately 80% being standard Java with JML specifications along with three lemmas and one predicate used to help with the proof along. Learning to develop the lemmas, that provide a functionality to perform complex operations that can then then be used in the specifications, requires a reasonable amount of time to master along with the predicates that return true or false values based on the parameters passed in and its implementation. While developing the correct JML specifications for each method and loop can be time consuming, the process itself rarely changes, as a result; with practice the speed at which this development takes place reduces.

*(Toccata.lri.fr. (2018a))*

Figure 3:Binary Search - Krakatoa

The WhyML version *(Toccata.lri.fr. (2018b))* required 43 lines of code and can use the inbuilt libraries, such as the div method on Line 29, instead of having to specify lemmas or predicates. While the specification in this example actually is smaller and more concise than the Krakatoa version, the difficulty arises with the ML syntax that must be learned by the user along the semantics of certain characters which can have different meanings and consequences to what was expected. A simple difference is the use of the ‘!u’ character on Line 24 which in normal programming terminology would mean not ‘u’, however in WhyML is dereferencing the variable u to return its value, similar to dereferencing pointers in C. While this seams like a trivial example that can be learned quite easily, as the programs become more complicated the WhyML language also becomes more complex and would require a module in itself to learn, see example *(Toccata.lri.fr. (2018c))*. Due to the use of JML in both the KeY and OpenJML tools, allied with the complexity of WhyML for those unfamiliar with functional languages, that we focused on the Krakatoa for this project. It should be noted however that the Why3 tool with WhyML has been annually at the top end of the Verify This competitions and is proving to be a leader in its field with the use of multiple back-end automated solvers proving its greatest asset.

*(Toccata.lri.fr. (2018b))*

Figure 4: Binary Search - WhyML

### KeY Verification Tool

The KeY tool was created by [Reiner Hähnle](https://www.se.informatik.tu-darmstadt.de/de/se/group-members/reiner-haehnle/), Wolfram Menzel, and [Peter Schmitt](https://lfm.iti.kit.edu/pschmitt.php) at University of Karlsruhe in 1998 *(Schmitt, P., Tonin, I., Wonnemann, C., Jenn, E., Leriche, S. & Hunt, J. (2006))*. It was developed as a source-code based verification system to be used for sequential Java programs along with their specifications written in the Java Modelling Language (JML) with the objective being to *‘integrate design, implementation, formal specification and formal verification of object-oriented software as seamlessly as* possible*’ (Ahrendt, W., Beckert, B., Hähnle, R., Rümmer, P. & Schmitt (2007)).*.

 Chapter 7: JML KeY version with Libraries

The JML specifications used in KeY java programs are translated into proof obligations in JavaDL before this is further refined to a taclet language for application of proof rules.

:Quantifiers with libraries

One of the KeY tools main advantages over other deductive verifiers is its abiliity to deal with theories, and specifically finite sequences denoted by the keyword ‘*\seq*’. This is used to deal with abstract datatypes such as Lists and provides certain libraries, for example seqLen(x) returns the length of x, to work with sequences. The addition of these libraries and there use in combinitation with the JML quantifiers, and the extended version of JML that KeY employs, provides a far greater range of proof obligations that can be generated by the KeY tool when translating the program. The technique of creating specification contracts using a combiniation of quantifiers and theories interlinked and their translation as a whole to proof obligations in JavaDL, gives the tool a significant advantage over other similar JML verifiers, albeit with the drawback of learning to master these speciifcation combination techniques as they often prove challenging and require expert knowledge. For more information on finite sequences, please refer to Chapter 5 of reference *(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))*.

*(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))*.

Figure 5: The KeY Verification Workflow

The KeY tool has a dedicated interactive theorem prover that lets the user find a proof, provide values for quantifier instantiations and step through each proof in stages. It provides its own standalone IDE for applying direct proof obligations as well as a plugin for the Eclipse IDE, however the Eclipse plugin cannot apply direct proof obligations to code. The KeY IDE also has an automated feature which will automatically select the optimal SMT solver and proof strategy for each section of code, this technique was used in KeY to avoid a common human interpretation issue with counter examples that are generated, usually, in Normal-Form (*Burns, D., Mostowski, W. & Ulbrich, M. (2015))*. If a SMT solver fails to provide a complete proof for a certain section of the code; the user can use the KeY IDE to select a different SMT Solver for that specific section, e.g Alt-Ergo is better for arithmetical proofs than z3. The proof strategies employed by the KeY automated verification tool ‘*provides compound interaction steps combine the application of several basic deductionsteps to achieve a specific purpose*’ and are defined as:

* + *Propositional expansion* (without splits) apply only non-splitting propositional rules
  + *Propositional expansion* (with splits) apply only non-splitting propositional rules
  + *Finish symbolic execution* apply only rules for modal operators
  + *Close provable goals* automatically close all open goals for which possible

(*Burns, D., Mostowski, W. & Ulbrich, M. (2015))*

Using the Design by Contract paradigm (*Meyer, B. (1992))*, KeY was built to support modular specification and verification. This proposed removing the specifications from the concrete implementations and moving them to the abstractions, such as interfaces, ensuring reusability and giving both the client and supplier a greater understanding of what was required for each contract to be satisfied. In 2013 , KeY 2.0 was released which allowed recursive method implementations to be modularly verified (*Burns, D., Mostowski, W. & Ulbrich, M. (2015))* by introducing a termiantion witness variable that uses the keyword ‘measured\_by’ that ensures total correctness for the recursive method by decreasing at each meothd call to itself *(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))*.

Java Dynamic Logic is the basis of the KeY logic system. The syntax of JavaDL extended first-order logic with program variables and program modalities *(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))*, and was designed to match the Java type system to reduce the learning curve required when using the tool. As discussed earlier, JavaDL uses a Kripke structure to evaluate formulas to determine valid paths and models. JavaDL uses parameterised modal operators (p) and [p], where p can be any sequence of legal Java statements which refer to the final state of program p, with (p)ɸ expressing that the program p terminates in a state which ɸ holds and [p]ɸ expressing that p does not demand termination but it if did then ɸ holds. Another type of modal operator, called ‘updates’, describes program state transitions that are stated as ‘*simple function updates corresponding to assignments in an imperative programming language, which in turn can be composed sequentially and used to form parallel or quantified updates’ (Ahrendt, W., Beckert, B., Hähnle, R., Rümmer, P. & Schmitt (2007)).* These updates always terminate and never have any side effects, only showing what state transition has occurred for the current path. Verification calculus transforms Java programs into these ‘updates’ with the KeY tool simplifying them to apply to formulas. However, as JavaDL uses first-order arithmetic when determining validity of a path, it results in the JavaDL logic never being both sound and complete due to this arithmetic being incomplete *(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))*. Relative completeness, however, is possible meaning all proofs are capable of being proven with the exception of some proofs that require specific first-order arithmetic operations that are not covered.

The construction of proofs in KeY is done differently to most other deductive verifiers. Instead of using the popular Verification Condition Generation (VCG) technique, which uses weakest precondition calculus to transform a program into one single proof obligation formula to then be discharged using a general purpose theorem , it uses the symbolic execution technique. This technique axiomatizes the program logic into a sequent calculus, written in a taclet language, to determine the final state constraints for each possible branch in the program, which are then evaluated by the provers (*Burns, D., Mostowski, W. & Ulbrich, M. (2015))*. *‘This process was used as it provided more feedback to the user since the formulae are more human-readable and allows for the debugging of said program’* (*Burns, D., Mostowski, W. & Ulbrich, M. (2015))*. ‘*Taclets are a concise description of rules that specify the logical content, context and pragmatics of its application*’ *(Ahrendt, W., Beckert, B., Hähnle, R., Rümmer, P. & Schmitt (2007)).* To perform this technique the statements of the program are expanded into simpler equivalent expressions, a process called unfolding that provides syntactic updates, and continues this process until all statements can no longer be simplified. Local variables are added to the expressions to hold intermediate computation results and then case distinctions are developed based on possible scenarios that could occur with the statement.

**

Figure 6: Symbolic Execution with Case Distinction

*(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))*.

The two processes combined (syntactic updates and case distinctions) are the essence of symbolic execution and work for normal Java statements but require further details, loop\_invariants, when dealing with loops as the unwinding process would be unbounded resulting in continuous interations. ‘Method invocations should be symbolically executed using a methods contract to ensure it is only symbolically executed once’ *(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))*.

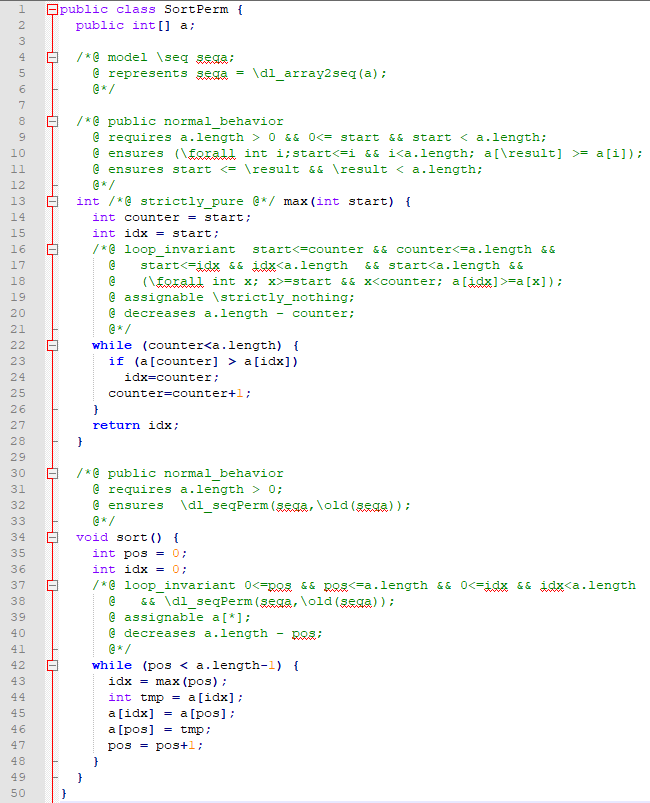
Chapter 3.6: Rules for Symbolic Execution

Symbolic Debugger Tool

Taclets is a theory formalization language represesting the first-order predicate logic and dynamic logic used in programs, as one logical sequent calculus that is used by KeY to build the interactive prover. The rules available for this new formula cover nearly all the rules used in both first-order predicate logc and dynamic logic, which enables KeY to create proof strategies that can be applied during proof automation. The taclet language captures the axioms of theories and algebraic specifications as rules and allows the use of lemmas in programs to help specific proofs where needed *(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))*.

**Key Case Study:**

1. Additional JML Libraries



*(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))*.

Figure 7: Sort Permutation

### OpenJML Verification Tool

OpenJML is a deductive verification tool for sequential Java programs annotated with JML specifications that performs type checking, runtime assertion checking as well as static verification *(Sanchez, J. & Leavens, G. (2014))*. It was created in 2009 by David R. Cok as an ‘*experiment to determine if the OpenJDK could replace the custom parser used in ESC/Java2 and the MultiJava compiler that underlies the JML2 tools*’ *(Cok, D.R. (2014))* , however it has grown significantly since 2011 with the goal of replacing ESC/Java2 with a universal JML implementation. This universal JML implementation would then, in theory, be adopted by industry and acedemia as part of their development structure and would set a standard implementation of JML for all Java specifications, stopping the ever growing subsets of JML that are in production such as those seen in the KeY and Why tools. The developers aim to achieve this goal by ‘*providing an IDE for managing program specifications that naturally fits into proactice of daily software development and so becomes a part of expected software engineering practice*’ *(Cok, D.R. (2014))*. Documentation is scarse on this tool due to it being relevantly new, with only a couple of case studies available along with one complete user manual written in 2014 *(Cok, D.R. (2014))*. As of this date another updated user manual is being created by David R.Cok *(Cok, D.R. (2016))*, however due to the development of the tool to increase its functionality taking precedence, the document is still in its early stages with the majority of material yet to be added.

OpenJML extends OpenJDK with modifications made to the parts OpenJDK to ensure correct functionality, such as using only non-public API’s along with other visibility changes *(Cok, D.R. (2014))*. The current version of OpenJML can be run on the command line as well as having a built-in plugin for the Eclipse IDE, providing a GUI version of OpenJML, with the target Java version being JDK8. OpenJML intends to be a sound tool *(Cok, D.R. (2014))* in that if a specification of a Java program in JML returned a valid result, the result was indeed valid and not a false positive. Errors in specifications can lead to program paths becoming infeasible and hide further specification errors that would occur later in the code which may affect soundness. The incompleteness of logical theories, such as first-order arithmetic *(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))*, may result in invalid counterexamples being produced by the SMT solvers, particularly when quantifiers are used in the specifications *(Cok, D.R. (2014))* . Static verification is done on a modular basis with each method’s specification and feasibility being checked independently.

The OpenJML tool itself checks that the JML specification satisfies the Java implementation using the Design by Contract paradigm and is therefore valid. OpenJML is designed in a similar fashion to other deductive verification systems with it being adapted from the ESC/Java2 system *(Cok, D.R. (2014))* with the process of determining the validity of program specifications started by translating the specifications into assertions and assumptions interleaved with the Java code. These constructs then generate Verification Conditions (VCG), expressed in SMTLIBv2 with a single VC generated for each method in a modular fashion, through an intermediate language using a block form that uses single-assignment labeling of variables. These VCs can then be discharged by an external SMT solver, chosen by the user, with valid being returned for each correct method specifications that the SMT solver can handle. If there is an error in the specification such as, invalid assertions, the chosen SMT solver cannot handle the type of VC, or infeasible paths that result in assertions or the method exit being unobtainable; a counterexample is created to guide the user to where the issue arose with a corresponding data model to show how the error path can be reproduced *(Cok, D.R. (2014))*.

JML Specifications

Examples – Binary Search case study

# **Chapter three: Case Studies**

The main objective of this thesis was to use OpenJML to test if it provides a simplified verification alternative to the more complex verification tools currently being developed. We have chosen the two competitor tools based on their prevalence in the VerifyThis annual verification challenge and their similarities to OpenJML. These tools are the Why3 verification tool with Krakatoa as its front-end and the KeY verification tool as both of these tools use Java as their programming language with JML as their specification language. Alt-Ergo and Z3 will be used as the SMT-solver back-end for all three tools. Why3 and OpenJML have command line tools to support their tool, however we will focus specifically on the recommended IDE’s supplied by the developers.

# Case Study 1

## Binary Search

The first case study chosen was the Binary Search algorithm (*binary search algorithm (2016))*, as the implementations and specifications had already been created by the developers and could therefore provide some initial analysis of the tools without encountering major complexity. This would allow us to get a feel for the how the tools operated with a standard example and show how their implementation and specification strategies differed as well as how their verification processes worked. It also provided a simple comparison of the JML syntax used by each tool.

### Krakatoa

#### Code and Specification

Krakatoa’s version (Figure 8: Lines of Code (LOC) = 55) proved to have the most complex implementation and specification due to its use of predicates, lemmas and pragmas from Lines 2-20. However in the program specification itself, only the predicate (/\*@ predicate is\_sorted{L}(int[] t)) is used with the lemmas there to provide assistance to the provers as stated by Claude Marche “Lemmas are additional properties that can be added usually to give hints to provers” *(Marché, C. (2009))*.

//@+ CheckArithOverflow = yes

/\* lemma mean\_property1 :

@ \forall integer x y; x <= y ==> x <= (x+y)/2 <= y;

@\*/

/\* lemma mean\_property2 :

@ \forall integer x y; x <= y ==> x <= x+(y-x)/2 <= y;

@\*/

/\* lemma div2\_property :

@ \forall integer x; 0 <= x ==> 0 <= x/2 <= x;

@\*/

/\*@ predicate is\_sorted{L}(int[] t) =

@ t != null &&

@ \forall integer i j;

@ 0 <= i && i <= j && j < t.length ==> t[i] <= t[j] ;

@\*/

The specification for the binary\_search method begins on Line 25 and uses the standard JML annotations to setup the general contract with the precondition and postcondition using the usual ‘*requires*’ and ‘*ensures*’ clauses on Lines 25 and 26. This contract requires the contract to be non-null and ensures that either a value is found during the search or -1 is returned. The ‘*behavior*’ keyword is then used to specify further contract requirements with a successful behaviour specified on Lines 27-28 and a deviation from normal behaviour specified on Lines 29-32; the ‘assumes’ statement used on Line 30 helps the SMT-solver when resolving that section of the proof. Note, the words used after the ‘*behavior*’ keyword can be changed to suit the users preference and hold no syntaxial value.

/\*@ requires t != null;

@ ensures -1 <= \result < t.length;

@ behavior success:

@ ensures \result >= 0 ==> t[\result] == v;

@ behavior failure:

@ assumes is\_sorted(t);

@ ensures \result == -1 ==>

@ \forall integer k; 0 <= k < t.length ==> t[k] != v;

@\*/

The loop invariants are setup on Lines 36-40 and are specified by the ‘*loop\_invariant*’ clause. The statement on lines 36-37 must hold before, during and after a successful execution and termination of the while loop on Line 44-51. The successful termination of this while loop depends on the loop variant using the ‘*loop\_variant*’ clause on Lines 41-42;) which ensures ‘*u*’ decreases with each loop iteration. An additional inductive invariant is setup on Lines 38-40 that must hold under the behavior setup on Lines 29-32.

/\*@ loop\_invariant

  @ 0 <= l && u <= t.length - 1;

@ for failure:

  @ loop\_invariant

  @ \forall integer k; 0 <= k < t.length ==> t[k] == v ==> l <= k <= u;

  @ loop\_variant

  @ u-l ;

  @\*/

#### Verification

The verification of program was done using the Why3 IDE which had a choice of two SMT-Solvers setup, Alt-Ergo and Z3. Once the program was loaded into Why3, the translation of the program to the Jessie IVL caused the LOC to grow to 1106 for the proof (Figure 9). From here, the user could select individual methods for proving, which proof strategy and rules to employ, as well as what prover to use for each method. Alternatively the user could automatically verify the program as a whole using the Auto-Level 2 option which selects the best solvers and rules to apply for each individual method and if an error occurred, the user can split the VC that resulted in the error and apply a different solver or rule to the ones used initially. If an error still occurs, then an error in specification or a vagueness in the proof is occurring and the program must be edited to rectify those errors.

During our verification process, we chose to use Auto-Level 2 option on the entire program as a whole and it resulted in 5 of the 6 proof VC’s being discharged with the fourth VC ensuring safety in the binary\_search method being unproven. This VC was split and it was determined from Figure 10 that the postcondition resulted in an Arithmetic Overflow. The pragma on Line 2 was then changed to //@+ CheckArithOverflow = no which resolved the issue and the program verified as seen in Figure 11. Ideally a bound would be placed on the variables highlighted by the Why3 tool in Figure 10 causing this overflow error, however as this case study was only for familiarity we decided this solution was sufficient.

## KeY

### Code and Specification

KeY’s implementation and specification of the Binary Search algorithm is done in a similar fashion to Krakatoa however it is achieved with 20 few lines of code achieving its goal in 35 LOC. It uses the universal ‘*requires*’ and ‘*ensures*’ clauses to set the initial contract as in Krakatoa however it uses far more complex quantifier statements to replace the predicates and lemmas. This is achievable in KeY due to the VC’s being created using Symbolic Execution which can create a tree structure of these quantifier conjunctions used on Lines 4 and 5. The ‘*requires*’ and ‘*ensures*’ clauses are part of the ‘*public normal\_behaviour*’ block, Lines 3-6, and are equivalent to the contract and ‘*behavior success:*’ block in Krakatoa. Note the different spellings of behavior and behaviour based on the tool used, these minute differences can lead to extended periods of debugging and is prime example of a need for agreement on a standard JML syntax that is used universally. The method is also declared ‘*pure*’ which states that the search method cannot does not and cannot have any side-effects on other m=methods or variables within the class BinarySearch.

static /\*@pure@\*/ int search(int[] a, int v)

An issue with this implementation is that on Lines 11 and 12, the code implementation checks to see if the array lengths are greater than zero and therefore valid or equal to one and see if the value is found in that single array index.

if(a.length == 0) return -1;

if(a.length == 1) return a[0] == v ? 0 : -1;

The specification should be checking that an array length is legal in the precondition statement of the method therefore adhering to the non-redundancy principle to reduce defensive programming as set out by Meyer *(Meyer, B. (1992))* , as well as perhaps changing the loop implementation in Line 20-29 to include the first array index, however the latter not being essential but merely preference.

The ‘*loop\_invariant*’ clause is on Lines 14-16 and again this specification conjoins multiple quantifiers and assertions into one statement. An ‘*assignable*’ clause is used on Line 17 to state nothing can be assigned in this loop and it is side-effect free and the ‘decreases’ clause, replaceing the ‘*loop\_variant*’ clause used in Krakatoa, is the loop variant used to ensure loop termination.

/\*@ loop\_invariant 0 <= l && l < r && r < a.length

@ && (\forall int x; 0 <= x && x < l; a[x] < v)

@ && (\forall int x; r < x && x < a.length; v < a[x]);

@ assignable \nothing;

@ decreases r - l;

@\*/

### Verification

Verification with the KeY tool was initially planned to be executed in the Eclipse plugin alongside its OpenJML equivalent, however the KeY plugin source could not be located as the website (<https://www.key-project.org/>) and documentation did not provide any current link, despite mentioning it numerous times and the majority of their documentation and tutorials being based on this plugin. Links were available on the old KeY website (<http://i12www.ira.uka.de/key/download/index.html#eclipse>), however they were based on much older Eclipse and Java versions and with the requirements of OpenJML being to use JDK8 with newer versions of Eclipse, we decided to use the KeY IDE provided via an executable file that could be downloaded. Once the KeY IDE loads, a java file must be selected however the file must be in their own specific folder as the KeY IDE loads all java files within the folder as opposed to only one that was selected. Figure 12 shows the KeY IDE once a proof has been loaded with a dropdown to choose your preferred solver as well as numerous verification options, although the standard defaults already set are for contracts similar to what we need to be verified so no changes were required.

We chose the Z3 solver for our proof and clicked the Start button to run the automatic verification process. Normally some interactive steps are required to complete the proof of a program however in this instance the automatic verifier complete the proof in just over 5 seconds creating over 4500 rules during the verification process (Figure 13). An example of these rules can be seen in Figure 14 which shows the complexity in applying the different rules per proof obligation and could prove very difficult if one of the goals failed to be verified automatically and required interactive application of these rules.

## OpenJML

### Code and Specification

OpenJML’s implementation and specification is taken from the rise4fun website <https://rise4fun.com/OpenJMLESC/BinarySearch> and has a very similar styling to the KeY version. The contract is once again stated with the ‘*requires*’ and ‘*ensures*’ clauses on Lines 4-6 ensuring if a match is found it is a positive value and returns -1 otherwise.

//@ requires (\forall int i, j; 0 <= i && i < j && j < arr.length; arr[i] <= arr[j]);

//@ ensures \result == -1 ==> (\forall int i; 0 <= i && i < arr.length; arr[i] != key);

//@ ensures 0 <= \result && \result < arr.length ==> arr[\result] == key;

Conversely, the OpenJML implementation also checks to see if the array length is greater than zero on Lines 8-9 as opposed to introducing a ‘*requires*’ clause to put that emphasis onto the client as well as not doing a non-null check for the array values themselves. They did however include the first array index into the loop implementation reducing that section of defensive programming.

The loop invariants are introduced on in Lines 14-16 and replace the ‘loop\_invariant’ keyword with the ‘*maintaining*’ keyword. The loop variant in introduced on Line 17 and along with KeY used the ‘*decreases*’ keyword.

//@ maintaining 0 <= low && low <= high && high <= arr.length && mid == low + (high - low) / 2;

//@ maintaining (\forall int i; 0 <= i && i < low; arr[i] < key);

//@ maintaining (\forall int i; high <= i && i < arr.length ==> key < arr[i]);

//@ decreases high - low;

### Verification

Verification for the OpenJML tool was carried out in the Oxygen version of the Eclipse IDE with JDK8 using the plugin supplied by OpenJML and the installation instructions, both found at <http://www.openjml.org/documentation/plugin.shtml> . A toolbar with the OpenJML tools is supplied to the user once installed however the ESC button does not currently operate as required and has to be set using the preferences section of the tool, which starts every time a change is made to the file and then saved. The SMT solver to be used is also set in the preferences section (Pictures?).

The supplied code with its specification verified with no issues, not a guarantee after the previous examples, and a detailed description of the results was displayed to the user (Figure 14). The RAC button functioned as expected with the results highlighted in blue (Figure 15). A type error was introduced into the program and the type-checking button selected resulting in the relevant errors being caught and highlighted in red. Finally, a change to the specification was made to the loop invariant on Line 16 by changing the last less than sign to a greater than sign as highlighted in the code snippet below.

//@ maintaining (\forall int i; high <= i && i < arr.length ==> key **>** arr[i]);

This caused the ESC to now fail and return the details in the console that one proof was invalid. On the Static Checker for the program the section where the issue occurred was highlighted in orange and once clicked brought up the counter example that both highlighted the code where the errors were occurring and the data model that could be used to reproduce the error (Figure 16). Once the code was changed back to its original version and saved, the ESC ran and both methods were once again valid.

## Analysis

The first thing to note is the somewhat unnecessarily changing of the ‘*loop\_invariant*’ keyword to ‘*maintaining*’ in OpenJML which in our opinion it serves no additional purpose as opposed to making it more difficult to reconnect the JML subsets into one version. KeY has the ability to use both keywords for describing loop invariants so we decided to try substituting in the ‘loop\_invariant’ keyword into OpenJML to see if it was supported, however it resulted in an error. This now let the systems split, with Kraktoa supporting ‘*loop\_invariant*’, OpenJML supporting ‘*maintaining*’ and KeY supporting both.

(CHECK DOES KRAKATOA SUPPORT MAINTAINING)

While the implementations are all very similar across the tools, the specifications differed quite vastly for such a small, relatively simple algorithm. Krakatoa employed predicates and lemmas to help the prover during verification while KeY uses quantifier conjunctions to provide a complex specification, in alliance with the symbolic execution process to create its VC’s. OpenJML using the VCG method, like Krakatoa, seems to have developed a much simpler and programmer friendly specification that is easier to understand and walkthrough. However this simplistic approach would be tested to a far greater extend in future case studies.

Also of note is the somewhat strange absence to adhere to standard formal programming rules such as the non-redundancy principle, violated by both the KeY and OpenJML examples by performing defensive programming within their method implementations as opposed to putting the responsibility on the client through the precondition, where it belongs.

The syntaxial differences at this stage are manageable with only minor spelling differences and the use of different keywords for the same operations proving the main difficulties.

# Case Study 2

## PrefixSum

### Goal

Our goal for this case study was to create a specification for the PrefixSum challenge from the VerifyThis 2012 challenge in OpenJML using the KeY recursive implementation as a starting point. We would also try to implement this challenge in Krakatoa simply as a terms of comparison. We chose the KeY implementation as this algorithm was significantly more difficult and required a team of experts from the KeY development team to complete this challenge, along with a substantial period of time. However this team confirmed that the majority of their recursive implementation could indeed be verified in the KeY tool with the required expertise and wrote a paper on the creation of the specification confirming this point *(Burns, D., Mostowski, W. & Ulbrich, M. (2015))*. Therefore we deemed this paper along with its code a valid starting point to begin the OpenJML specification.

#### OpenJML Updates

RAC

Stop Proofs – Introduce Timeouts

#### Attempt 1

##### Code and Specification

The first decision we made was that we would not alter the implementation for the Java code for this algorithm as it achieves what the algorithm sets out, and therefore would only be lloking at altering the specification used by the KeY team. The first task for this conversion was to remove all aspects of native KeY code from the specification and later try to replicate this missing functionality with OpenJML if required. The specification on Line 7 had to be removed, which stated implicitly that only the singleton set consisting of array ‘*a*’ is accessible as the \inv: and \singleton clauses are not supported in OpenJML. OpenJML can specify this constraint on the array ‘*a*’ using class invariants or within the individual method contracts themselves using the ‘*\accessible*’ clause. All further ‘*\singleton*‘ clauses were also removed or replaced.

//@ accessible \inv: \singleton(a);

Line 127-128 and 156-157 contained the ‘*\infinite\_union*‘ clause which was part of the set expressions introduced to JML in 2011 by Benjamin Weiβ *(Weiß, B. (2011))* and has been translated to JavaDL for use in KeY. This clause is “ *a set comprehension operator that binds the variable of any type and has a location set expression in the body*” (*Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))* , and is used in this instance to state that the value of ‘*k*’ is only dependant on the current array index ‘*k*’ that lies within the scope set out by the Leftmost method and additional bounds. We were unsure as to how to model this in OpenJML specification, so the decision was made to comment it out until required.

@ assignable \infinite\_union(int k; leftMost(left,right) <= k

@ && k <= right && !even(k); \singleton(a[k]));

@ assignable \infinite\_union(int k; leftMost(left,right) <= k

@ && k <= right; \singleton(a[k]));

Additional syntax changes were required for the type checker to pass for OpenJML. All ‘strictly\_pure’ and \strictly\_nothing’ notations were changed to ‘pure’ and \nothing respectively. The former clauses are extensions of JML in KeY that provide stronger constraints on the method functionality. The ‘/strictly\_nothing’ clause means that no location may be changed, even those newly created within the method scope, while the ‘*/strictly\_pure*’ clause states that no new location is allowed to be altered or created in the method (*Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))*. Also KeY allows each methods specifications to call another methods specification anytime, however in OpenJML, ‘*spec\_public*’ must be added to a method’s specification if it is to allow access to its specification.

@ assignable \strictly\_nothing; **🡪** @ assignable \nothing

@ strictly\_pure **🡪** @ pure spec\_public

The PrefixSum class also had to be made public if an object of its type is to be used in a specification, as it is on Lines 171-174. The specification on Lines 171-174 was removed until all the ESC verification was complete on the PrefixSum class itself. The labels on Lines 81 and 103 were also removed as this functionality is not used in OpenJML.

//ß\label{lst:min-begin}ß

//ß\label{lst:eff-begin}ß

##### Verification

Once the code and specification was made to match the OpenJML syntax and pass the type-checker and RAC tests we could begin the first phase of verification for this algorithm using Z3 as our back-end STM-Solver. We ran the ESC tool and the process hung, stuck on 0% Progress for over 25 minutes before we tried cancelling the process. Even though we cancelled the proof using the GUI provided, the tool continued running for another 15 minutes before we had to forcefully kill the Eclipse IDE itself. On the next opening of the Eclipse IDE, the ESC process resumed and once again could not be cancelled, however on examination of the console logs it was determined that the ‘binWeight’ method was the reason for the failure. Once the ESC eventually stop executing, we removed the ‘*binWeight’* method as it was only used as part of the ‘*downSweep’* specification which was previously commented out by the KeY developers.

Once we ran the ESC, after the binWeight method was removed, we received a multitude of errors from almost every method within the PrefixSum class so we decided it was best to work on a modular basis and develop the specifications on a method by method basis and not move forward until all previous methods had been verified.



Figure 8: OpenJML - PrefixSum - ESC errors

##### Analysis

We encountered issues that we expected in this phase, such as the differences in syntax between the KeY version and OpenJML as well as the JML extensions that are translated in to JavaDL for the KeY tool that are simply not implemented in OpenJML. However, we also encountered issues that we did not see coming such as the inability to cancel out of proofs that are taking too long with even the forcible closure of the Eclipse IDE not stopping the tool, with it simply resuming the process on relaunch of the IDE. Eclipse also initially would not recognise the Z3 back-end solver even when specifically set in the preferences section of OpenJML. The list of solvers had to be altered so that Z3 was set as the default version in order for the system to recognise it.

The main advances we made in this attempt was merely realising we could not specify this class as a whole and had to modularise our approach and focus our efforts on small pieces of code, which was not unexpected. This initial attempt was initially assumed to be the most time consuming procedure we would encounter during the verification process due to having to understand the KeY versions functionality and JML extensions and we believed we could advance at a respectable pace from hence forth.

#### Attempt 2

##### Code and Specification

The first method we chose from the PrefixSum class was ‘*evenSumLemma*’ (Lines 15-21). Line 21 of the ‘*evenSumLemma*’ specification states that if ‘x’ is even and ‘y’ is even, then that implies ‘x+y’ is also even.

/\*@ normal\_behavior

@ ensures \result == (\forall int x, y; even(x) == (even(y) == even(x+y)));

@ ensures \result;

@ accessible \nothing;

@ strictly\_pure helper

@\*/

private static boolean evenSumLemma() { return true; }

This required the ‘*even’* method (Lines 66-74) to be verified beforehand.

/\*@ normal\_behavior

@ ensures \result == (\exists int y; y\*2 == x);

@ ensures \result != (\exists int y; y\*2 == x+1);

@ accessible \nothing;

@ strictly\_pure helper

@\*/

private static boolean even (int x) {

return x%2==0;

}

Line 8 is an axiom which tells the prover to assume the predicate named is true, which is the ‘*evenSumLemma*’ method.

//@ axiom evenSumLemma();

##### Verification

When we ran the ESC with the ‘*evenSumLemma’* and ‘*even’* methods unaltered, we received an error “*z3 does not support evaluation of quantified formulas*”. This was an error in the ‘*even*’ method specification on Lines 67-68, but was an error as a result of the SMT-solver chosen as opposed to an OpenJML error itself. Therefore the ‘*even*’ method specification required changes and Lines 67-68 were replaced with a simpler clause taken from the method implementation itself which preserved the overall constraint.

~~@ ensures \result == (\exists int y; y\*2 == x);~~

~~@ ensures \result != (\exists int y; y\*2 == x+1);~~

@ ensures \result == (x%2==0);

Once the changes were made, both methods verified.

##### Analysis

The z3 error encountered in this attempt would prove troublesome due to the specification developed by the KeY team making use of this technique on a regular basis, as Symbolic Execution is able to resolve these problems through the creation of models. The ability of Why3 to automatically switch back-end solvers, for different sections of the code and specification, would have proven very helpful for this issue with the restriction to one solver per proof allied with the use of VCG proving restrictive in the development of specifications for OpenJML. However, it could be argued that the simplification of specifications is one of the goals of OpenJML and would greatly ease the development process for novice users even if it is at the expense of more advanced users.

#### Attempt 3

##### Code and Specification

For the next section we introduced the '*div2*' method which simply returned a value divided by two. No additional alterations were required for this method. We then introduced the ‘*leftMost’* method which required additional specifications to bound the variables involved.

/\*@

     @ requires 0 <= a.length && a.length <= Integer.MAX\_VALUE;

     @ requires 0 <= left && left <= a.length;

     @ requires 0 <= right && right <= a.length;

     @ pure spec\_public

     @\*/

     private /\*@ helper @\*/ int leftMost(int left, int right) {

        return 2\*left - right + 1;

     }

##### Verification

ESC Verification of the ‘*div2*’ method resulted in an *ArithmeticOperationRange* error for Line 56, however there appeared no issue with this clause and in the resulting ESC process, all the clauses were deemed valid.

@ ensures \result\*2 == x;

The 'leftMost' method new specifications also had an *ArithmeticOperationRange* error during initial ESC verification checks with the having no upper bound resulting in the upper bound of the integer type being broken after the multiplication operation on Line 78. Restricting the upper bound of a.length to Integer.MAX\_VALUE / 2 solved the issue.

~~@ requires 0 <= a.length && a.length <= Integer.MAX\_VALUE;~~

@ requires 0 <= a.length && a.length <= Integer.MAX\_VALUE / 2;

##### Analysis

The ESC verification of the 'div' and 'leftMost' methods was the easiest in the program. The (error) for the 'div' method was worrying behaviour since it was returning invalid results for specifications that were valid which may potentially mean there is a flaw in the proof system and invalid results could being returned valid. As this was the only occurrence of this type of action happening, we believed it was an anomaly but still should be noted. The 'leftMost' methods upper bound issue was easy to spot and initially fix with an upper bound of 1000, and later, after conversations with David R.Cok on other similar issues, resolved to using the Integer.MAX\_VALUE variable.

#### Attempt 4

##### Code and Specification

We were making good progress up to this point with the majority of the KeY specifications being usable in the OpenJML, and we moved forward introducing the 'pow2' method accordingly. This was implemented as a recursive method to return the 2 to the power of the variable 'x' passed in by the user. The recursive implementation was replaced with an interative version after it became clear during verification that the recursive method employed by the KeY developers did not work in the same fashion using OpenJML. As an example, we set the variable 'x' to be 3 which should have returned the value 8 but instead returned a value 2147483646 which is the Integer.MAX\_VALUE - 1 value as can be seen from the trace in Figure (xxx). It was discovered that recursion is not fully built-in to the tool yet with the measured\_by clause not implemented to ensure termination of the recursive method, therefore we introduced the iterative implementation.

##### Verification

ESC Verification first determined, as stated above, that the recursive implementation along with its specification were not going to work with OpenJML so we used an iterative implementation. The verification of this approach however, also brought up a lot of issues. The first of which was on Line … , specifically the \product quantifier. This product quantifer should have returned a result a result to match 2 to the power of 'x' but was instead returning seemingly random values, many of which were not even multiples of 2 or were below 0 despite a precondition stating 'x' must be equal to or greater than 0. Another issue was that the variables uses in the method appeared to be not holding their values in memory and would change from line to line. The count variable was set to 1 on entering the method and if 'x' is 0, the returned value should still be 1, however the resulting value was returning 2147483647 (Integer.MAX\_VALUE). Another example of this was when 'x' was 0 but value changed by the time the while loop came so it executed the operation on count returning 1073741824.

##### Analysis

All of the OpenJML issues combined to make the verification of the 'pow2' method almost impossible and resulted in mass amounts of time being wasted in pursuit of these issues and determining a resolution. It was later learned from David that although the \product quantifer is used in OpenJML, its implementation on the back-end has not been completed so is running an unknown process returning random values. The memory issues are a far greater problem and have been addressed to David for resolution. The investigation into these issues is still ongoing as of writing this report, with no resolution available as of yet. As a result, all future methods that wither relied on the specification or implementation of the 'pow2' method could no longer be adequately proven. This memory issue in effect proved terminal as all future methods use 'pow2' directly or indirectly within their specifications resulting in the end of the verification process for the PrefixSum algorithm.

# **Chapter four: The Solution**

## Summary

The purpose of this chapter is to clearly identify, discuss, and justify the decisions you made.

“**Solution” (often the name of your solution) – Details what you have done and how you have done it.**

* + **Objectives –** A single sentence that describes the purpose of this section.
  + Provide an analysis of the problem, motivating your approach to answering the research question.
  + Explain your approach by describing exactly what you have done.
  + Explain how you have achieved your solution. Examples: explain how a process improvement was implemented, how a mathematical technique was derived, or how an algorithm was implemented.

## Depending on your type of project, you may not need to include all of these:

## 4.1 Analytical Work

E.g. Equations, etc. that describe your solution

## 4.2 Architectural Level

E.g. Implementation Diagrams

## 4.2 High Level

## E.g. Packages, Class Diagrams, etc.

## 4.2 Low Level

## E.g. Method specifications, Algorithms, etc.

## 4.2 Implementation

Discuss anything interesting here; put full source code in an appendix or attachment

# **Chapter five: Evaluation**

## Summary

Chapter 5 describes……..

**Evaluation – Evaluates your work (both in absolute terms, and compared to other solutions)**

* + **Objectives –** A single sentence that describes the purpose of this section.
  + Explain what was evaluated or validated.
  + Experimental setup – Detail how you evaluated and validated your work.
  + Present your results clearly and objectively, without interpretation - ideally with graphs (data)
  + Explain your results - ideally with explanatory text (analysis) to both explain the meaning of these results, and provide the reasons for why these particular results were obtained
  + Critically analyze your results. Identify the contents in which your results are relevant and any threats are to the validity of your results. Show how well you have answered the research question.
  + Critically analyze your results with respect to the “Related Work” presented earlier.

## 5.1 Solution Verification

## E.g. use your equations to verify the correctness of your solution

## 5.2 Software Design Verification

How did you show that your design worked properly?

Using a model of your solution. E.g. use UML interaction diagrams to verify each scenario.

## 5.3 Software Verification

How did you demonstrate your software worked properly?

If you have not tested your software, then you cannot rely on your results. Clearly describe:

### 5.3.1 Your test approach (i.e. unit testing, sub-system testing, system testing)

### 5.3.2 Your tests (e.g. scenarios, test cases, test data, etc.)

### 5.3.3 Your test results

### 5.3.4 An interpretation of the results

## 5.4 Validation/Measurements

How did you measure how well your solution solved the problem.

### 5.4.1 Results

### 5.4.2 Explanation of Results

### 5.4.3 Analysis of Results

### 5.4.4 Comparison with previous solutions (if relevant)

**Chapter five: Conclusion**

**Summary**

Chapter 5 identifies and discuss the implications of your work.

**Draws conclusions and identifies potential future work**

* + **Objectives –** A single sentence that describes the purpose of this section.
  + Summarize your results. Provide your conclusions (limitations & recommendations) based on the results obtained. Detail the implications of your results with respect to the wider community.
  + Assess how well you have met your project goals. Identify the contributions made by this work.
  + Critically analyze your approach to solving the research question by explaining what was effective in your approach, and what you could have been improved upon.
  + Present possible future work - How could you/others build on your research to advance it further?

5.1 Contribution to the state-of-the-art

If you made a contribution to the state-of-the-art, clearly identify it here.

5.2 Results discussion

Discuss whether your results are general, potentially generalizable, or specific to a particular case. Identify threats to the validity of your results (e.g. limitations, risks introduced by your approach, etc.)

5.3 Project Approach

Discuss your project approach

5.3 Future Work

Discuss future work, based on what you have done (and not done)

# **References**

*Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016). Deductive Software Verification – The KeY Book: From Theory to Practice. 10.1007/978-3-319-49812-6.*

*Ahrendt, W., Beckert, B., Hähnle, R., Rümmer, P. & Schmitt, P.H. (2007) "Verifying Object-Oriented Programs with KeY: A Tutorial" in Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 70-101.*

*Beckert, B., Hähnle, R., Schmitt, P.H. (2007), Chalmers University of Technology, Institutionen för data- och informationsteknik, Datavetenskap (Chalmers), Chalmers tekniska högskola & Department of Computer Science and Engineering, Computing Science (Chalmers) 2006;, Verification of object-oriented software: the KeY approach, Springer, New York;Berlin;.*

*Bernardeschi, C. & Domenici, A. (2016), "Verifying safety properties of a nonlinear control by interactive theorem proving with the Prototype Verification System", Information Processing Letters, vol. 116, no. 6, pp. 409-415.*

*Biere, A., Bloem, R. & SpringerLink (Online service) (2014) Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, Springer International Publishing, Cham.*

*binary search algorithm (2016), , 7th edn, Oxford University Press.*

*Bobot, F., Filliâtre, J., Marché, C. & Paskevich, A. (2015) "Let's verify this with Why3", International Journal on Software Tools for Technology Transfer, vol. 17, no. 6, pp. 709.*

*Bormer, T. (2014) Advancing deductive program-level verification for real-world application: lessons learned from an industrial case study, ProQuest Dissertations Publishing.*

*Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J.R., Leavens, G.T., Leino, K.R. & Poll, E. (2005), "An overview of JML tools and applications", International Journal on Software Tools for Technology Transfer, vol. 7, no. 3, pp. 212-232*

*Burns, D., Mostowski, W. & Ulbrich, M. (2015) "Implementation-level verification of algorithms with KeY", International Journal on Software Tools for Technology Transfer, vol. 17, no. 6, pp. 729-744.*

*Catano, N., Barraza, F., García, D., Ortega, P., Rueda, C. (2009) A case study in JML-assisted software development. In: Proceedings of the Eleventh Brazilian Symposium on Formal Methods (SBMF 2008). ENTCS, July 2009, vol. 240, pp. 5–21*

*Cok, D.R. & Kiniry, J.R. (2005) "ESC/Java2: Uniting ESC/Java and JML: Progress and Issues in Building and Using ESC/Java2, Including a Case Study Involving the Use of the Tool to Verify Portions of an Internet Voting Tally System" in Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 108-128.*

*Cok, D.R. (2014) "OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse", Electronic Proceedings in Theoretical Computer Science, vol. 149, pp. 79-92.*

*Cok, D.R., (2016) “Does your software do what is should?” Specification and verification with the Java Modelling Language and OpenJML. The OpenJML User Guide*

*Cok, D.R., Leavens, G.T., & Ulbrich, M. (2018), Java Modelling Language Reference Manual*

#### Cs.ru.nl. (2018a). [online] Available at: https://www.cs.ru.nl/E.Poll/talks/jml\_tutorial/1\_intro\_jml.pdf [Accessed 1 Jun. 2018].

#### Cs.ru.nl. (2018b). [online] Available at: https://www.cs.ru.nl/E.Poll/talks/jmlrac.pdf [Accessed 1 Jun. 2018].

#### Cs.ru.nl. (2018c). [online] Available at: https://www.cs.ru.nl/E.Poll/talks/jml\_tutorial/2\_tool.pdf [Accessed 1 Jun. 2018].

*De Angelis, E., Fioravanti, F., Pettorossi, A. & Proietti, M. (2017);2016;, "Semantics-based generation of verification conditions via program specialization", Science of Computer Programming, vol. 147, pp. 78-108.*

*de Gouw, S., de Boer, F., Ahrendt, W. & Bubel, R. 2016, "Integrating deductive verification and symbolic execution for abstract object creation in dynamic logic", Software & Systems Modeling, vol. 15, no. 4, pp. 1117-1140.*

*de Moura, L. & Bjørner, N. (2008), "Z3: An Efficient SMT Solver" in Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 337-340.*

#### Drops.dagstuhl.de. (2018). [online] Available at: http://drops.dagstuhl.de/opus/volltexte/2017/7259/pdf/LIPIcs-ECOOP-2017-9.pdf [Accessed 13 May 2018].

#### Eecs.ucf.edu. (2018b). The Java Modeling Language (JML) Home Page. [online] Available at: http://www.eecs.ucf.edu/~leavens/JML//index.shtml [Accessed 13 May 2018].

*for software verification, Maynooth University*

*Felleisen, M., Gardner, P. & SpringerLink (Online service) (2013), Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg.*

*Filliâtre, J. (2011), "Deductive software verification", International Journal on Software Tools for Technology Transfer, vol. 13, no. 5, pp. 397-403.*

#### Formal.iti.kit.edu. (2018). [online] Available at: https://formal.iti.kit.edu/beckert/pub/keytutorial2016.pdf [Accessed 13 May 2018].

*Furia, C.A., Meyer, B. & Velder, S. 2014, "Loop invariants: Analysis, classification, and examples", ACM Computing Surveys (CSUR), vol. 46, no. 3, pp. 1-51.*

*Ganzinger H., Hagen G., Nieuwenhuis R., Oliveras A., Tinelli C., (2004) DPLL(T): fast decision procedures, in: R. Alur, D. Peled (Eds.), Proceedings of the 16th Conference on Computer Aided Verification, Lecture Notes in Computer Science, vol. 3114, pp. 175–188*

*Giacobazzi, R., Berdine, J., Mastroeni, I. & ebrary, I. 2013, Verification, model checking, and abstract interpretation: 14th International Conference, VMCAI, 2013, Rome, Italy, January 20-22, 2013 : proceedings, Springer, Heidelberg.*

*Giorgetti, A., Groslambert, J., Julliand, J. & Kouchnarenko, O. (2008), "Verification of class liveness properties with Java modelling language", IET Software, vol. 2, no. 6, pp. 500-514.*

*Giorgetti, A., Marché, C., Tushkanova, E. & Kouchnarenko, O. 2010, "Specifying generic Java programs: two case studies", ACM, , pp. 1.*

#### Hal.inria.fr. (2018a). [online] Available at: https://hal.inria.fr/hal-01344110/document [Accessed 13 May 2018].

#### Hal.inria.fr. (2018b). [online] Available at: https://hal.inria.fr/hal-00789533/document [Accessed 13 May 2018].

*Healy, A. (2016) Predicting SMT solver performance*

*Hoare, C. (1983), An axiomatic basis for computer programming, ACM*

*Huisman, M., Klebanov, V. & Monahan, R. 2015, "VerifyThis 2012 - A Program Verification Competition", International journal on software tools for technology transfer, vol. 17, no. 6, pp. 647-657.*

*Jacobs, B., Smans, J. & Piessens, F. 2015, "Solving the VerifyThis 2012 challenges with VeriFast", International Journal on Software Tools for Technology Transfer, vol. 17, no. 6, pp. 659-676.*

*Kandziora, J., Huisman, M., Bockisch, C. & Zaharieva-Stojanovski, M. (2015), "Run-time assertion checking of JML annotations in multithreaded applications with e-OpenJML", ACM, , pp. 1.*

*Kassios, I.T., Müller, P. & Schwerhoff, M. (2012), "Comparing Verification Condition Generation with Symbolic Execution: An Experience Report" in Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 196-208.*

#### Key-project.org. (2018a). [online] Available at: https://www.key-project.org/wp-content/uploads/2017/10/slides-pp.pdf [Accessed 13 May 2018].

#### Key-project.org. (2018b). The KeY Project. [online] Available at: https://www.key-project.org/ [Accessed 13 May 2018].

#### Kindsoftware.com. (2018). [online] Available at: http://kindsoftware.com/documents/talks/KSU\_ESCJava2\_Object\_Logic.pdf [Accessed 13 May 2018].

*Kiniry, J., Morkan, A. & Denby, B. 2006, "Soundness and completeness warnings in ESC/Java2", ACM, , pp. 19.*

#### Krakatoa.lri.fr. (2018a). [online] Available at: http://krakatoa.lri.fr/krakatoa.pdf [Accessed 13 May 2018].

*Leavens, G.T., & Cheon, Y. (2003). Design by Contract with JML.*

#### Krakatoa.lri.fr. (2018b). Krakatoa and Jessie: verification tools for Java and C programs. [online] Available at: http://krakatoa.lri.fr/#krakatoa [Accessed 28 May 2018].

*Kuhtz, L. & Finkbeiner, B. (2011), "Weak Kripke Structures and LTL" in Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 419-433.*

*Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C. & Cok, D.R. (2005), "How the design of JML accommodates both runtime assertion checking and formal verification", Science of Computer Programming, vol. 55, no. 1, pp. 185-208.*

*Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Muller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Werner, D., (2013) “JML Reference Manual: JML Reference Manual.” [online] Available at: http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman.html [Accessed 13 May 2018]*

*Leavens, G.T., Kiniry, J.R. & Poll, E. (2007), "A JML Tutorial: Modular Specification and Verification of Functional Behavior for Java" in Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 37-37.*

*Leavens, G. T. , Baker, A. L. & Ruby, C. (1999) JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, Behavioral Specifications of Businesses and Systems, pages 175–188. Kluwer Academic Publishers, Boston*

*Maidi, M. (2000), "The common fragment of CTL and LTL", , pp. 643.*

*Marché, C., Paulin-Mohring, C. & Urbain, X. (2004), "The KRAKATOA tool for certificationof JAVA/JAVACARD programs annotated in JML", Journal of Logic and Algebraic Programming, vol. 58, no. 1, pp. 89-106.*

*Marché, C. (2009), “The Krakatoa tool for Deductive Verification of Java Programs”, INRIA Saclay – F-91893 Orsay cedex*

*Meyer, B. (1992), "Applying 'design by contract'", Computer, vol. 25, no. 10, pp. 40-51*

*Nieuwenhuis, R., Oliveras, A. & Tinelli, C. (2006), "Solving SAT and SAT Modulo Theories: From an abstract Davis--Putnam--Logemann--Loveland procedure to DPLL( T )", Journal of the ACM (JACM), vol. 53, no. 6, pp. 937-977.*

#### Nipkow, T., Paulson, L.C., Wenzel, M. & SpringerLink (Online service) 2002;2000;2006;, Isabelle/HOL: a proof assistant for higher-order logic, Springer, New York;Berlin;

#### Pdfs.semanticscholar.org. (2018). [online] Available at: https://pdfs.semanticscholar.org/ce71/23d4388ea2b776f31967377b10d4ff11698e.pdf [Accessed 13 May 2018].

*Pedersen, J.B. & Welch, P.H. 2018, "The symbiosis of concurrency and verification: teaching and case studies", Formal Aspects of Computing, vol. 30, no. 2, pp. 239-277.*

*Pek, E. 2015, Automated deductive verification of systems software, ProQuest Dissertations Publishing.*

*Philippaerts, P., Muhlberg, J.T., Penninckx, W., Smans, J., Jacobs, B. & Piessens, F. 2014, "Software verification with VeriFast: Industrial case studies", Science of Computer Programming, vol. 82, pp. 77.*

#### Pm.inf.ethz.ch. (2018). VerifyThis Competition. [online] Available at: http://www.pm.inf.ethz.ch/research/verifythis.html [Accessed 13 May 2018].

*Poll, E. (2009), "Teaching Program Specification and Verification Using JML and ESC/Java2", Teaching Formal Methods : Second International Conference, TFM 2009, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings, vol. 5846, pp. 92-104.*

*Sánchez, J. & Leavens, G. (2014), "Static verification of ptolemyrely programs using openJML", ACM, , pp. 13.*

*Segal, L. & Chalin, P. (2012), "A Comparison of Intermediate Verification Languages: Boogie and Sireum/Pilar" in Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 130-145.*

*Schmitt, P., Tonin, I., Wonnemann, C., Jenn, E., Leriche, S. & Hunt, J. 2006, "A case study of specification and verification using JML in an avionics application", ACM, , pp. 107.*

#### Shonan.nii.ac.jp. (2018). [online] Available at: http://shonan.nii.ac.jp/shonan/wp-content/uploads/2011/09/No.2013-3.pdf [Accessed 13 May 2018].

#### Toccata.lri.fr. (2018a). Binary Search, Java version . [online] Available at: http://toccata.lri.fr/gallery/BinarySearch.en.html [Accessed 28 May 2018].

#### Toccata.lri.fr. (2018b). Binary search . [online] Available at: http://toccata.lri.fr/gallery/binary\_search.en.html [Accessed 28 May 2018].

#### Toccata.lri.fr. (2018c). Computing the number of solutions to the N-queens puzzle . [online] Available at: http://toccata.lri.fr/gallery/queens.en.html [Accessed 28 May 2018].

*Weiß, B. (2011). Deductive Verification of Object-Oriented Software — Dynamic Frames, Dynamic*

*Logic and Predicate Abstraction. PhD thesis, Karlsruhe Institute of Technology, Karlsruhe.*

#### Why3.lri.fr. (2018a). [online] Available at: http://why3.lri.fr/tallinn-2013/notes.pdf [Accessed 13 May 2018].

#### Why3.lri.fr. (2018b). The Why3 platform . [online] Available at: http://why3.lri.fr/doc-0.86/ [Accessed 13 May 2018].

#### Why3.lri.fr. (2018c). Why3 . [online] Available at: http://why3.lri.fr/ [Accessed 13 May 2018].

#### Wiki.portal.chalmers.se. (2018). [online] Available at: http://wiki.portal.chalmers.se/cse/uploads/Research/WAVR.pdf [Accessed 13 May 2018].

*Yang, K.H., Olson, D. & Kim, J. (2004), "Comparison of first order predicate logic, fuzzy logic and non-monotonic logic as knowledge representation methodology", Expert Systems With Applications, vol. 27, no. 4, pp. 501-519.*

*Yi, J., Qi, D., Tan, S.H. & Roychoudhury, A. 2013, "Expressing and checking intended changes via software change contracts", ACM, , pp. 1.*

**Appendices**

# Chapter xyz:

## Symbolic Execution



In Figure x, a ‘path condition’ is created by the branching statement ‘*if(x < y)*’ but instead of executing all possible concrete paths, it constructs a tree based on the abstract structure of the program. The left and right branch both execute until they both return a value, resulting in the program termination. If a loop was used in such a program, a similar branching mechanism would occur, however a loop\_invariant and a loop\_variant may be required to ensure termination of the loop branches.

Figure SE: Symbolic Execution Tree: 'min' method

*(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))*.

# Chapter xyz

## Case Study 1

### Binary Search Examples

###### Krakatoa

Figure 9: Krakatoa Binary Search



|  |
| --- |
|  |
| Figure 10: Krakatoa - Jessie Model – Binary Search |
| Figure 11: Krakatoa - Jessie Model - Binary Search Safety    Figure 12: Krakatoa - Jessie Model - Verified |

###### KeY



Figure 13: KeY IDE



Figure 14: KeY IDE - Binary Search - Proof





Figure 15: KeY IDE- Binary Search - Rules

###### OpenJML



Figure 16: OpenJML - Binary Search

Figure 17: OpenJML - Eclipse - Valid Verification





Figure 18: OpenJML - Eclipse - RAC



Figure 19: OpenJML - Eclipse - TypeCheck

Figure 20: OpenJML - Eclipse - ESC Error



## Case Study 2

### PrefixSum

#### KeY Implementation

Figure 21: KeY - PrefixSum implementation



## Appendix 5 Taught M.Sc. Dissertation Guidelines (valid from Oct 2015)

**Taught M.Sc. Dissertation Guidelines**

**(valid from October, 2015)**

This document provides guidelines for your M.Sc. level dissertation for modules CS640 and CS645. There is no standard layout (except for the cover page), as the details may be determined by the project topic and the approach you have taken. You should read a number of other dissertations (available on Moodle or from ePrints) to get an idea of the accepted norms. Your supervisor will be able to advise you further.

Your dissertation won't necessarily be *organised* as shown here, but it MUST *contain* the following information:

* Title
* Abstract
* Introduction
* Related Work
* “Solution” (i.e. title of your work)
* Evaluation
* Conclusions
* References
* Appendices

How you present the research question and your solution will depend to a certain extent on the nature of your project. You need to show that you are aware of other research in the area, and show the relationship of at least one other publication to your own work.

The dissertation absolute limit is 22,000 words (using size 12 Times New Roman font and single line spacing, and not including the appendices). A suggested format for your report is detailed on the next page. Supporting documentation such as your documented code should be uploaded separately as directed by your course co-ordinator. **The submission must be all your own work**. Please read the Maynooth University policy on Plagiarism and ensure that your reference material correctly. The minimum penalty for plagiarism is a failed grade in your thesis.

Recommendation: agree on a “model” report with your supervisor that you can base your approach and layout on. The following diagram shows the ‘flow’ or ‘argument’ you should use in presenting your work.



**Title Page** - Template on next page (replace the highlighted text).

**Abstract -** This is a summary of the research question, your results, and your contribution in 200 words or less.

**Category, Terms, Keywords:** reference [www.**acm**.org/sigs/**publications**/pubform.doc](http://www.acm.org/sigs/publications/pubform.doc)for details

**Suggested sections and sub-sections**

* **Introduction – a high level description of the research question and the problem domain that can be understood by somebody new to the subject area.**
  + **Objectives –** A single sentence that describes the purpose of this section.
  + **Research Question** – State the technical problem that you have focused on in your project in the form of a question which you address.
  + **Motivation** – Discuss the reasons for solving this problem. Detail the problem domain and who would be interested in the solution. Describe the likely impact of your work. Address both why it is an interesting technical problem, and also the value of solving it in more general terms.
  + **Aims and Objectives** – State the aims and objectives of your project. The **aims** of your project are the overall goal, and the **objectives** are the stepping stones in reaching that goal. Identifying the objectives helps the reader to understand your overall project approach.
  + **Report Structure** - Outline the structure of the report summarizing each chapter in one sentence.
* **Related Work – Details what others have done that is relevant to your work.**
  + **Objectives –** A single sentence that describes the purpose of this section.
  + Describe the context of the research question in detail, defining terminology, and with references.
  + Explain how the problem, or related problems, has been solved previously. Critically analyze existing solutions. Discuss how your approach compares to these solutions.
  + Explain other techniques that you have used to: help understand and analyze the research question; motivate your own work; evaluate your solution.
* “**Solution” (often the name of your solution) – Details what you have done and how you have done it.**
  + **Objectives –** A single sentence that describes the purpose of this section.
  + Provide an analysis of the problem, motivating your approach to answering the research question.
  + Explain your approach by describing exactly what you have done.
  + Explain how you have achieved your solution. Examples: explain how a process improvement was implemented, how a mathematical technique was derived, or how an algorithm was implemented.
* **Evaluation – Evaluates your work (both in absolute terms, and compared to other solutions)**
  + **Objectives –** A single sentence that describes the purpose of this section.
  + Explain what was evaluated or validated.
  + Experimental setup – Detail how you evaluated and validated your work.
  + Present your results clearly and objectively, without interpretation - ideally with graphs (data)
  + Explain your results - ideally with explanatory text (analysis) to both explain the meaning of these results, and provide the reasons for why these particular results were obtained
  + Critically analyze your results. Identify the contents in which your results are relevant and any threats are to the validity of your results. Show how well you have answered the research question.
  + Critically analyze your results with respect to the “Related Work” presented earlier.
* **Conclusions – Draws conclusions and identifies potential future work**
  + **Objectives –** A single sentence that describes the purpose of this section.
  + Summarize your results. Provide your conclusions (limitations & recommendations) based on the results obtained. Detail the implications of your results with respect to the wider community.
  + Assess how well you have met your project goals. Identify the contributions made by this work.
  + Critically analyze your approach to solving the research question by explaining what was effective in your approach, and what you could have been improved upon.
  + Present possible future work - How could you/others build on your research to advance it further?

**References –** Proper full complete citations for all referenced documents (NOT a URL). For a master’s level document one would expect up to 30 good references. Use peer-reviewed papers or books: not websites.

**Appendix** – Details of: source code, protocols, data, results, etc.

**Note** – Use a repository to store your code & build procedure.