# Can OpenJML provide a simpler, viable Software Verification process for Software Developers ?

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

[Can OpenJML provide a simpler, viable Software Verification process for Software Developers ? i](#_Toc516842534)

[Title i](#_Toc516842535)

[Declaration i](#_Toc516842536)

[Acknowledgements ii](#_Toc516842537)

[Abstract iii](#_Toc516842538)

[List of Figures iv](#_Toc516842539)

[List of Tables iv](#_Toc516842540)

[Chapter One: Introduction 1](#_Toc516842541)

[Summary 1](#_Toc516842542)

[1.1 Overview 1](#_Toc516842543)

[1.2 Motivation 1](#_Toc516842544)

[1.3 Objectives 2](#_Toc516842545)

[1.4 Approach 2](#_Toc516842546)

[1.5 Metrics 3](#_Toc516842547)

[1.6 Project 3](#_Toc516842548)

[Chapter Two: Related Work 4](#_Toc516842549)

[2.1 Deductive Software Verification 4](#_Toc516842550)

[2.2 Model Checking 4](#_Toc516842551)

[2.3 Logics 4](#_Toc516842552)

[2.4 Design by Contract 4](#_Toc516842553)

[2.5 Runtime Assertion Checking (RAC) 5](#_Toc516842554)

[2.6 Extended Static Checking (ESC) 5](#_Toc516842555)

[2.7 Java Modelling Language (JML) 6](#_Toc516842556)

[2.7.1 JML Description 6](#_Toc516842557)

[2.7.2 JML Syntax 6](#_Toc516842558)

[2.7.3 Ghost and Model 7](#_Toc516842559)

[2.7.4 Quantifiers 7](#_Toc516842560)

[2.8 Intermediate Verification Languages (IVL’s) 8](#_Toc516842561)

[2.9 Verification Condition Generators (VCG’s) 8](#_Toc516842562)

[2.10 Symbolic Execution (SE) 8](#_Toc516842563)

[2.11 Verification Conditions 8](#_Toc516842564)

[2.12 Theorem Provers 9](#_Toc516842565)

[2.13 Satisfiability Solvers (SAT) 9](#_Toc516842566)

[2.14 Satisfiability Modulo Theories (SMT) 9](#_Toc516842567)

[Chapter Three: Tools 10](#_Toc516842568)

[3.1 Why3 Verification Tool 10](#_Toc516842569)

[3.2 KeY Verification Tool 12](#_Toc516842570)

[3.3 OpenJML Verification Tool 15](#_Toc516842571)

[Chapter Four: Case Studies 16](#_Toc516842572)

[4.1 Overview 16](#_Toc516842573)

[4.2 Case Study – Binary Search 16](#_Toc516842574)

[4.2.1 Goal 16](#_Toc516842575)

[4.2.2 Krakatoa 16](#_Toc516842576)

[4.2.3 KeY 18](#_Toc516842577)

[4.2.4 OpenJML 19](#_Toc516842578)

[4.3 Case Study – PrefixSum 21](#_Toc516842579)

[4.3.1 Goal 21](#_Toc516842580)

[4.3.2 Algorithm 21](#_Toc516842581)

[4.3.3 Attempt 1 21](#_Toc516842582)

[4.3.4 Attempt 2 23](#_Toc516842583)

[4.3.5 Attempt 3 24](#_Toc516842584)

[4.3.6 Attempt 4 25](#_Toc516842585)

[4.3.7 Attempt 5 27](#_Toc516842586)

[4.4 Longest Repeating Substring 28](#_Toc516842587)

[4.4.1 Algorithm 28](#_Toc516842588)

[4.4.2 Attempt 1 28](#_Toc516842589)

[Chapter Five: Analysis 30](#_Toc516842590)

[5.1 Overview 30](#_Toc516842591)

[5.2 Case Studies - Analysis 31](#_Toc516842592)

[5.2.1 Binary Search 31](#_Toc516842593)

[5.2.2 PrefixSum 33](#_Toc516842594)

[5.2.3 Longest Repeated Substring 40](#_Toc516842595)

[5.3 Verification Tools - Analysis 40](#_Toc516842596)

[5.3.1 JML Dialects 40](#_Toc516842597)

[5.4 Tool Recommendations 43](#_Toc516842598)

[Chapter Six: Evaluation 44](#_Toc516842599)

[6.2 Overview 44](#_Toc516842600)

[6.3 BinarySearch 44](#_Toc516842601)

[6.4 PrefixSum 44](#_Toc516842602)

[6.5 Longest Repeated Substring 45](#_Toc516842603)

[6.6 OpenJML Tool 45](#_Toc516842604)

[6.7 Project Approach and Assessment 46](#_Toc516842605)

[Chapter Seven: Conclusion 47](#_Toc516842606)

[Summary 47](#_Toc516842607)

[5.1 Contribution to the state-of-the-art 47](#_Toc516842608)

[5.2 Results discussion 47](#_Toc516842609)

[5.3 Project Approach 47](#_Toc516842610)

[5.3 Future Work 47](#_Toc516842611)

[References 48](#_Toc516842612)

[Appendices 55](#_Toc516842613)

[Chapter xyz: 55](#_Toc516842614)

[Symbolic Execution 55](#_Toc516842615)

[Chapter xyz 56](#_Toc516842616)

[Case Study 1 56](#_Toc516842617)

[Case Study 2 68](#_Toc516842618)

[PrefixSum 68](#_Toc516842619)

[Appendix 5 Taught M.Sc. Dissertation Guidelines (valid from Oct 2015) 89](#_Toc516842620)

## Title

## 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. This tool 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**

## 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

LINE

## 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, developed with the overall goal to reduce defensive programming and increase reliability by introducing mathematical proofs into a methods specification, therefore enforcing the clients and suppliers compliance *(Meyer, B. (1992))*.

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 (*Huisman, M., Klebanov, V. & Monahan, R. (2015))*.

The developers of these tools do not communicate or collaborate regularly with each other, focusing primarily on developing their own tool’s functionality which leaves the users without any standard tool or process within the verification field. Novice users, just coming into the formal verification domain, have a steep learning curve with having to learn the separate libraries and syntax variables employed by each tool as well as to embrace the core concepts of Programming by Contract.

OpenJML aims to bridge this gap by allowing its freely available tool to be integrated into the Eclipse IDE directly and using only the basic dialect of the JML specification language (Section 2.6) 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 this tool functions in comparison to its competitors, KeY and Krakatoa for this project, and if the stripping down to just the basics of JML with Java would be viable for real-life industrial systems.

## 1.3 Aims and Objectives

The main aim of this project is to determine how effective OpenJML can be 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 achieve this aim by solving programs from the VerifyThis competition, specifically the PrefixSum and Longest Repeating Substring questions from the 2012 competition, as they have been specified and verified by other tools with a clear benchmark in place for comparison.

With respect to the OpenJML Deductive Verification process we hope 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 aim to provide feedback and data to the developers 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 is to determine if the OpenJML tool is complete enough to replace all other verification tools and streamline the formal verification process, reducing complexity for users and with the hope of widespread adoption in both academia and industry alike.

## 1.4 Approach

We planned to achieve our objectives by comparing OpenJML with the KeY and Krakatoa verification tools in regards to three case studies. The first case study, Binary Search (Chapter) was used to determine the differences in JML syntax and execution of the verification process between these three tools. The following two case studies (Chapters ) were examples taken from the VerifyThis competitions, with a KeY implementation and specification already applied. The goal was to create OpenJML and Krakatoa specifications, using the same KeY implementations as a code skeleton, in order to provide a comparative analysis of the capabilities of the tools as well as document the specification process as it happened. We analyse (Chapter) the approaches taken by each of the tools on a modular basis using each method’s specification as a basis for comparison with an overall determination made once verification was either valid or unable to be satisfied.

## 1.5 Contributions

We have determined that the OpenJML tool is not ready to be used in industry or academia due to its lack of recursive elements with method specifications, recursive model method specifications and quantifier implementations restricting the specification process resulting in constraints on the implementations themselves..

We have also found that the OpenJML tool within the Eclipse IDE, as well as command line, is not fully functional as of yet and requires further development with many bugs and issues still present. The lack of complete documentation for the OpenJML tool has also proven troublesome with only a very small group of case study papers to work from.

We also determined that the JML dialects have splintered across varying verification tools and a definitive version must be established to enable novice users an easier entry into the verification domain.

Also of note is the application of transitional rules in the KeY interactive tool is non-intuitive and there is no substantial documentation or tutorials to assist the user adequately.

# Chapter Two: Related Work

LINE

## 2.1 Deductive Software Verification

Deductive Software Verification is the process of developing specifications that can be mathematically 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.

## 2.2 Model Checking

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 branching logic where “*every temporal operator has to be preceded by a path quantifier, and hence 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 temporal logic patterns. The KeY verification 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.

## 2.3 Logics

There are many types of logic available today however the main logics we focus on regarding deductive verification are first-order predicate 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 Programming” *(Hoare, C. (1983))*, and introduced applying deductive reasoning as a way to formally reason about and develop software programs, that could be mathematically 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.

## 2.4 Design by Contract

Bertrand Meyer published his paper, “Design by Contract” *(Meyer, B. (1992))*, which introduced creating contract specifications for each method through the use of pre and postconditions. This introduced 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 satisfied 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 invariants 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 should not be strengthened and postconditions should not 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.

## 2.5 Runtime Assertion Checking (RAC)

Runtime Assertion Checking’s main application is for 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 (Section 2.6).

## 2.6 Extended Static Checking (ESC)

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 absence 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

HOW DOES ESC RELATE TO TOHER VERIFIERS

## 2.7 Java Modelling Language (JML)

### 2.7.1 JML Description

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 mathematical 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 mathematical operators such as the quantifiers \exists and \forall*” whose inclusion incorporated the first-order predicate logic into the language and “*hides mathematical 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))*.

### 2.7.2 JML Syntax

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 REFERENCE. This requires incorporating as many JML versions 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 switching 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**



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

Line 4: requires indicates 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 indicates 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))*.

Figure x: KeY Array-Search Loop Example

Line 5: \result stores 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 invariant (called loop\_invariant in Krakatoa 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 constraint 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 REWRITE.

Line 14: decreasing indicate the loop variant (called loop\_variant in Krakatoa 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.

### 2.7.3 Ghost and Model

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 EXAMPLE. The model type can be either an abstraction only variable/method used to help the specification or represent 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 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 they are 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))*.

### 2.7.4 Quantifiers

Another major adaption was the introduction of additional quantifiers to represent a subset of commonly used mathematical operations. The additions were the \product, \sum, \max and \min quantifiers which could be used in conjunction with the first-order predicate quantifiers \exists and \forall to ease the specification process. However, not all verification tool currently implement these additions to the JML library resulting in specifications being much more complicated to construct within these tools. Also the conjunction of quantifiers 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.

## 2.8 Intermediate Verification Languages (IVL’s)

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 properties of the original program*” (*Segal, L. & Chalin, P. (2012))*. IVL’s are used to create an abstraction of the program, regardless of the programming language used, that can then generate Verification Conditions to be discharged by the theorem provers. Translating the programming and specification 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 in as input and translates them to WhyML, in the Why3 Tool, to be further translated to VCs using VCGs *(Krakatoa.lri.fr. (2018b)).*

## 2.9 Verification Condition Generators (VCG’s)

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 theorem provers either automatically or interactively from the user (*Burns, D., Mostowski, W. & Ulbrich, M. (2015)*.

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 theorem provers can apply optimization techniques to the VC, the theorem provers may be unable to process them adequately without some interactive direction from the user *(Kassios, I.T., Müller, P. & Schwerhoff, M. (2012))*.

SMT-LIB – SMT-LIB2

## 2.10 Symbolic Execution (SE)

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)).*

## 2.11 Verification Conditions

“*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))*. 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.

## 2.12 Theorem Provers

One of the original standalone theorem provers is Isabelle/HOL which was implemented in the functional programming language ML (Meta Language) which focused on interactive theorem proving in higher-order logics *(Nipkow, T., Paulson, L.C., Wenzel, M. (2006))*. Another standalone interactive theorem 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 standalone theorem provers then gave way to the more popular Satisfiability (SAT) solvers and SMT solvers which were used by verification tools as external provers, allowing the user to create programs with specifications in non-functional languages such as Java and C++.

## 2.13 Satisfiability Solvers (SAT)

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 determining 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 symbolic execution process working on multiple paths is attached in Appendices Figure SE.

## 2.14 Satisfiability Modulo Theories (SMT)

Satisfiability Modulo Theories (SMT) 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 functions, 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 theories 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)).

SMT-LIB and SMT-LIB2

# Chapter Three: Tools

## 3.1 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 (See Chapter 2.13 and 2.14 – EXPLAIN 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 with Verification Condition Generators (See Section 2.9) being the process used to create proof obligations.

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 syntax 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.

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, we focused on 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.

## 3.2 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)).*.

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 EXAMPLE FROM BOOK. 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.

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

Figure 3: The KeY Verification Workflow

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. Taclets are a theory formalization language representing 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 logic 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))*.

Chapter 7: JML KeY version with Libraries

:Quantifiers with libraries

One of the KeY tools main advantages over other deductive verifiers is its ability 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 combination 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 combination 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 specification 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))*.

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 proof strategy for each section of code based on the SMT solver selected. 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 termination witness variable that uses the keyword ‘measured\_by’ that ensures total correctness for the recursive method by decreasing at each method call to itself *(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))*.

The construction of proofs in KeY is done differently to most other deductive verifiers. Instead of using the popular Verification Condition Generation (VCG) technique (Section 2.9), it uses the symbolic execution technique (Section 2.10). MOVE 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 4: 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

## 3.3 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))*. MOVE TO ANALYSIS Documentation is scarce 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 SOURCE. 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.

SMT-LIB

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))*.

DIAGRAM OF FLOW

# Chapter Four: Case Studies

LINE

## 4.1 Overview

The main objective of this thesis was to use OpenJML to determine 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. We present three case studies to analyse the differences between the tools used and the viability of specifications using OpenJML.

## 4.2 Case Study – Binary Search

### 4.2.1 Goal

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.

### 4.2.2 Krakatoa

#### 4.2.2.1 Code and Specification

Krakatoa’s version (Figure 8: Lines of Code (LOC) = 55) seemed to have the most complex implementation and specification due to its use of predicates, lemmas and pragmas (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 SMT 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 (Line 25-xx) uses the standard JML annotations to setup the general contract with the precondition and postcondition using the usual ‘*requires*’ and ‘*ensures*’ clauses (Lines 25 and 26). This contract requires the array 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 (Lines 27-28) and a deviation from normal behaviour specified (Lines 29-32); the ‘assumes’ statement (Line 30) helps the theorem prover 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 (Lines 36-40) and are specified by the ‘*loop\_invariant*’ clause. The statement (Lines 36-37) must hold before and after a successful execution and termination of the while loop (LineS 44-51). The successful termination of this while loop depends on the loop variant (Lines 41-42) which checks that ‘*u*’ decreases with each loop iteration. An additional inductive invariant is setup (Lines 38-40) that must hold under the behavior specified (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 ;

  @\*/

#### 4.2.2.2 Verification

The verification of program was done via the Why3 IDE which had a choice of two SMT-Solvers setup, Alt-Ergo and Z3. Once the program is loaded into Why3, the translation of the program to the Jessie IVL causes the LOC to grow to 1106 for the proof (Figure 9). From here, the user can select individual methods to prove, which proof strategy and rules to employ, as well as what prover to use for each method. Alternatively the user can 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 the issue lies in the specification or a vagueness in the proof is occurring and the program must be edited by the user 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 we determined (Figure 10) that the postcondition was resulting 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.

### 4.2.3 KeY

#### 4.2.3.1 Code and Specification

KeY’s implementation and specification of the Binary Search algorithm is similar that to Krakatoa however it is achieved with 20 fewer lines of code. 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 (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 methods or variables within the class BinarySearch.

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

An issue with this implementation is that the code (Line 12,13) checks to see if the array lengths are greater than zero or equal to one and therefore check if the value if found.

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

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

The specification should be improved by 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 *(Meyer, B. (1992))* , as well as changing the loop implementation (Line 20-29) to include the first array index.

The ‘*loop\_invariant*’ clause (Lines 14-16) conjoins multiple quantifiers and assertions into one statement. An ‘*assignable*’ clause (Line 17) to states that nothing can be assigned in this loop and it is side-effect free with the ‘decreases’ clause, equivalent to the ‘*loop\_variant*’ clause used in Krakatoa, is the loop variant used to prove 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;

@\*/

#### 4.2.3.2 Verification

Verification with the KeY tool was initially planned 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 the KeY 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 a 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. It is common for more complex implementations that some interactive steps are required to complete the proof of a program however in this instance the automatic verifier completes 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 applying the different rules per proof obligation and become very difficult if one of the goals required interactive application of these rules.

### 4.2.4 OpenJML

#### 4.2.4.1 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 (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 (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 (Lines 14-16) are indicated with the ‘*maintaining*’ keyword. The loop variant is introduced (Line 17) and, along with the KeY tool, 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;

#### 4.2.4.2 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 and a detailed description of the results was displayed to the user (Figure 14). The RAC 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 (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 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.

## 4.3 Case Study – PrefixSum

### 4.3.1 Goal

Our goal for this case study is to create a specification for the Longest Repeated Substring and the PrefixSum challenges from the VerifyThis 2012 competition in OpenJML using the KeY implementations as a starting point. We also implement the PrefixSum challenge in Krakatoa simply as a terms of comparison, however our goal is to focus on the OpenJML tool throughout. We chose the KeY implementations as these algorithm were significantly more difficult than the Binary Search algorithm and required a team of experts from the KeY development team to complete these challenges. The KeY team confirmed that the majority of their implementations 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.

### 4.3.2 Algorithm

BRIEF DESCRIPTION

### 4.3.3 Attempt 1

#### 4.3.3.1 Code and Specification

We do not alter the implementation for the Java code for this algorithm as it achieved what the algorithm sets out, the task is to alter the specification used by the KeY team to express it in OpenJML. First we remove all aspects of native KeY code from the specification and later try to replicate this missing functionality with OpenJML. The specification (Line 7) is 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 the same constraint on the array ‘*a*’ using class invariants or within the individual method contracts themselves using frame condition with the ‘*\accessible*’ clause. All further ‘*\singleton*‘ clauses throughout the class were also removed.

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

The ‘*\infinite\_union*‘ clause (Line 127-128 and 156-157) was part of the set expressions introduced to JML in 2011 by Benjamin Weiβ *(Weiß, B. (2011))* and has been translated to JavaDL (See Chapter 3.1) 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 required for the type checker to pass for OpenJML are made with all ‘strictly\_pure’ and \strictly\_nothing’ (Lines )notations 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 method’s specifications to call another method’s specification anytime, however in OpenJML, ‘*spec\_public*’ must be added to a method’s specification if a method is to allow access to its specification. We add the ‘*spec\_public*’ to all the necessary method’s as well as to variables that are used by such method specifications.

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

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

The PrefixSum class also has to be made public if an object of its type is to be used in a specification (Lines 171-174), however we do not need this specification until we have completed the full verification of the PrefixSum class so can also be, temporarily, removed. The labels (Lines 81 and 103) were a native KeY feature and as such are removed as their functionality is not used in OpenJML.

//ß\label{lst:min-begin}ß

//ß\label{lst:eff-begin}ß

#### 4.3.3.2 Verification

Once the code and specification match the OpenJML syntax and pass the type-checker and RAC tests we begin the first phase of verification for this algorithm using Z3 as our back-end STM-Solver. We run the ESC tool but the process hung, stuck on 0% progress for over 25 minutes before cancelation. Even though the proof is cancelled using the GUI provided, the tool continues running for another 15 minutes before we forcefully kill the Eclipse IDE itself. On the next opening of the Eclipse IDE, the ESC process resumes and once again cannot be cancelled, however on examination of the console logs it is determined that the ‘binWeight’ method is the reason for the failure. Once the ESC eventually stops executing, we remove the ‘*binWeight’* method as it was only used as part of the ‘*downSweep’* specification which was not needed in our specification.

Once we run the ESC, after the binWeight method is removed, we receive a multitude of errors from almost every method within the PrefixSum class so we decide it is best to work using a modular approach and develop the specifications on a method by method basis, not moving forward until all previous methods have been verified.



Figure 5: OpenJML - PrefixSum - ESC errors

### 4.3.4 Attempt 2

#### 4.3.4.1 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();

#### 4.3.4.2 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 (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 require changes with Lines 67-68 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.

### 4.3.5 Attempt 3

#### 4.3.5.1 Code and Specification

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

/\*@ normal\_behavior

     @ requires x > 0;

     @ requires even(x);

     @ ensures \result\*2 == x;

     @ ensures \result == x/2;

     @ ensures \result < x;

     @ accessible \nothing;

     @ pure helper spec\_public

     @\*/

     private static int div2 (int x) {

     return x/2;

     }

/\*@

     @ 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;

     }

#### 4.3.5.2 Verification

ESC Verification of the ‘*div2*’ method resulted in an *ArithmeticOperationRange* error for Line 56, however there appeared to be 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 has an *ArithmeticOperationRange* error during initial ESC verification checks, no upper bound restriction of the variable on Line XY is resulting in the upper bound of the integer type being broken after the multiplication operation (Line 78). Restricting the upper bound of a.length to Integer.MAX\_VALUE / 2 solves the issue.

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

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

### 4.3.6 Attempt 4

#### 4.3.6.1 Code and Specification

The ‘isPow2’ method (Lines 23-39) is the next method we add into our OpenJML implementation. The specification made use of recursion (Line 25) and uses the ‘measured\_by’ clause as the basis for termination of the recursive method, which relies on parameter ‘x’.

/\*@ public normal\_behavior

@ requires x > 0;

@ ensures \result ==> ((even(x) && isPow2(div2(x))) <=!=> x == 1);

@ ensures \result == (\exists int b; 0 <= b;

@ x == (\product int i; 0 <= i && i < b; 2));

@ measured\_by x;

@ accessible \nothing;

@ pure helper

@\*/

private static boolean isPow2(int x){

if (x==1)

return true;

else if (x % 2 != 0 )

return false;

else

return isPow2(x/2);

}

#### 4.3.6.2 Verification

Verification of the ‘isPow2’ method fails due to the recursion used within its specification. After consultation with the OpenJML developers, we discovered that recursion is not wholly implemented within OpenJML with the ‘measured\_by’ clause having no effect for determining termination of the inductive process. This lead us to instead remove all recursive properties from the KeY implementation of the PrefixSum algorithm, starting with the ‘isPow2’ method. This required creating iterative implementations to replace the inductive processes with iterative implementations and changing the specifications to match.

Verification fails and we initially determined that instead of processing the while loop with the value 'x' passed in the user, the Line 'LoopInvariantBeforeLoop assertion: \_JML\_\_tmp193' changes the value of 'x' to 1, corrupting the original data. Therefore when entering the while loop and succeeding assertions, the results are flawed and therefore could not match the specification used.

/\*@ normal\_behavior

     @ requires x > 0 && x < Integer.MAX\_VALUE;

     @ ensures \result ==> ( even(x) != (x == 1) );

     @ accessible \nothing;

     @ pure helper spec\_public

     @\*/

     private static boolean isPow2(int x){

         /\*@

         @ maintaining x >= 1;

         @ decreases x/2;

             @\*/

            while(x%2 == 0){

                x = div2(x);

            }

            if(x==1){

                return true;

            }

            return false;

     }

This proved to be a false assumption and through consultation with the OpenJML developers once more, it’s determined that the loop invariants are not strong enough and do not bound the loop to a certain number of iterations. The OpenJML developers provided an example solution to such a problem through the use of a Boolean model method ‘\_isPow2’ which has a restricted number of values that if matched return true, else return false. This, along with the introduction of a new loop invariant, bounds the loop invariant ensuring the old and current values are part of the *‘\_isPow2*’ range of values resulting in a valid specification.

However, the restriction of the *‘\_isPow2*’ model method’s values through its post-condition on Line xxx (Figure XYZ) is akin to defensive programming and as such can be deemed to be in breach of the non-redundancy principle (Section 2.xx) and is also not an ideal programming style as it constrains the codes overall capabilities. This is acceptable for guaranteeing the program specifies correctly but cannot be used in real-life systems.

//@ ensures \result == (x==1||x==2||x==4||x==8||x==16||x==32);

   //@ model public pure helper static boolean \_isPow2(int x);

    /\*@ normal\_behavior

   @ requires x > 0 && x < 33;

   @ ensures \result ==> ( even(x) != (x == 1) );

   @ ensures \result <==> \_isPow2(x);

   @ pure helper spec\_public

   @\*/

   private static boolean isPow2(int x){

     /\*@

     @ maintaining x > 0 && x < 33;

     @ maintaining \_isPow2(\old(x)) == \_isPow2(x);

     @ decreases x;

       @\*/

      while(x%2 == 0)     {

        x = div2(x);

      }

      if(x==1){

        return true;

      }

      return false;

   }

### 4.3.7 Attempt 5

#### 4.3.7.1 Code and Specification

Once we complete the ‘isPow2’ verification, we introduce the 'pow2' method accordingly. This is also implemented as a recursive method to return 2 to the power of the variable 'x' passed in by the user. As stated in the previously in the verification of the ‘isPow2’ method, the recursive clause ensuring termination, ‘*measured\_by’*, was not implemented so an iterative version was developed to replace the implementation.

  /\*@ public normal\_behavior

         @ requires x >= 0;

         @ ensures \result > 0 && \result < 33;

@ ensures \_isPow2(\result);

         @ assignable count;

         @ spec\_public

         @\*/

         private static int pow2(int x) {

            count = 1;

         /\*@

             @ maintaining x >= 0;

             @ maintaining \_isPow2(count);

             @ decreases x;

            @\*/

            while(x>0)

            {

             //@ assume x!=0;

            if(count < 33)

             count = mult2(count);

             x--;

            }

            //@ assume x==0;

            return count;

         }

#### 4.3.7.2 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 use an iterative implementation. The verification of this approach, however, also brought up a lot of issues. The first of which is on Line … , specifically the \product quantifier. This product quantifier should have returned a result to match 2 to the power of 'x' but is instead returning seemingly random values, many of which were not even multiples of 2, or the results are below 0 despite a precondition stating 'x' must be equal to or greater than 0 from the beginning.

As we are using the iterative implementation we are again forced to use the model method *‘\_isPow2’* in order to create a valid specification. This had an impact on the implementation code and restricts the count value to those specified within this model method and an ‘if’ statement is required in the while loop to force such a constraint.

We create a new ‘mult2’ method for the purpose of performing the function of multiplying the value by two. The implementation and specifications are similar to the ‘div2’ method and are therefore easy to develop.

  /\*@ requires x > 0 && x < Integer.MAX\_VALUE/2;

   @ ensures \result/2 == x;

   @ ensures \result == x\*2;

   @ ensures \result > x;

   @ accessible \nothing;

   @ pure helper spec\_public

   @\*/

   private static int mult2 (int x) {

   return x\*2;

   }

## 4.4 Longest Repeating Substring

Due to the specification issues restricting the implementations in OpenJML, as discussed earlier, during the verification of the PrefixSum algorithm, we know the likelihood of verifying the Longest Repeated Substring algorithm is minimal. Therefore, hence forth, we set out to perform verification with the goal of finding more unforeseen errors and take a view more suited to system testing as opposed to software verification.

### 4.4.1 Algorithm

BRIEF DESCRIPTION

### 4.4.2 Attempt 1

#### 4.4.2.1 Code and Specification

The code for the Longest Repeating Substring is done in an object oriented fashion, by the KeY developers, with four separate classes interlinked through composition and aggregation, and represents a more real world code example than previous case studies. There were two classes provided, LRS and SuffixArray, without specification by the people running the VerifyThis competition with a further two classes developed by the KeY team, LCP and Lemmas used to help with providing a correct specification. As we are not intending to verify this implementation completely, or most likely correctly, we just did the basics of refactoring the KeY syntax to match the OpenJML syntax in order the pass the type checking, RAC processes and perform ESC to see all available errors and counter-examples provided.

This refactoring passed the type-checker after the modification of some variables visibility, required due to the object oriented make-up of the program, however we encounter an unexpected error when we run the RAC function. When RAC is run on the ‘*LRS*’ class (Figure …), an internal error occurrs on execution as seen in Figure xyz. This was due to the RAC functionality not being fully developed and upon reporting to the OpenJML developers, they developed an OpenJML update, version 0.8.29, that fixed this error.

#### 4.4.2.2 Verification

Verification of the LRS, Lemmas and LCP classes throw up invalid assertions with counter-examples provided, that when traced show the majority of the errors were failings in the ‘*SuffixArray*’ class. As ‘*SuffixArray*’ objects were being used within the other classes, the verification of ‘*SuffixArray*’ must be completed first. However during the verification of this class we encounter a reoccurring issue that also occurred during the PrefixSum verification; the inability to terminate proofs that are taking too long to prove.

This issue seemed to be occurring during the proof of a complex loop, a recursive method where the inductive process must be unfolded or from specifications where multiple quantifiers were concatenated together which the VCG cannot unravel. Forcefully stopping the ESC process does not seem to have any effect and attempting to close Eclipse itself is ineffective as the Eclipse IDE waits for all jobs to complete before closure. A kill command is required from the command terminal to terminate the Eclipse process however upon restarting the Eclipse IDE, the verification process merely starts again therefore rendering the IDE prone and useless.

The OpenJML developers recommended setting a timeout variable in the preferences section however this also seems to have no effect so therefore another update was required. This was developed and supplied with the error fixed in OpenJML version 0.8.31. After this update immediate termination of all ESC processes can no take place however a warning error now appears upon running of any ESC process. This warning does not seem to have any physical ramifications however will require inspection and ratification.

# Chapter Five: Analysis

LINE

## 5.1 Overview

* Discuss Case Study findings (needed again?)
* Discuss Case Study tables (differences between verification techniques and issue)
* JML versions – table with differences (DONE)
* Identified high overhead of automated theorem proving pg 18,19
* Use of IVL’s to assist theorem proving
* Automatic and Interactive levels in tools
* Behaviour vs Behavior in JML syntax
* Changing program implementation to remove defensive programming didn’t work in OpenJML
* Standard Defaults for tools
* OpenJML SMT Solvers (cvc4, z3, yipes2, simplify)
* Specification affecting implementation
* RAC and ESC Tool failures in Eclipse
* How OpenJML tool works, adding in errors for counter-examples
* Same BinarySearch implementations vs different implementations
* Explain all specification and implementation decisions for PrefixSum
* List Recommendations for OpenJML Tool – traceable back to OpenJML
* Explain singleton, label, infinite\_union from KeY and if necessary in OpenJML
* Explain what works in Krakatoa (KML)
* Explain use of Model method in PrefixSum and then as a predicate in Krakatoa
* Implications of using spec\_public, pure, helper etc…
* How to select correct SMT-Solver (A.Healy thesis)
* Difference between SMT-Solver and Theorem prover
* User recommendations – Iterative over Recursive, implement support for model recursion methods from OpenJML developers

We include our analysis on a modular basis for the Case Studies, similar to the specification process itself, in order to capture the level of detail required and provide feedback as it occurred in real-time. In this analysis we will discuss how the verification process proceeded within each tool using the provided tables as a basis for comparison.

We also use this section to discuss the properties of the three main verification tools used throughout this project and attempt to do a meaningful comparison of these tool. No known benchmark or comparison could be discovered within any documentation and therefore we decided to try and bring as much of the core properties together (TableXYZ) as well as discussing the JML dialects used with these tools (TableXYZ).

## 5.2 Case Studies - Analysis

### 5.2.1 Binary Search

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 only supporting ‘*loop\_invariant*’, OpenJML only supporting ‘*maintaining*’ and KeY supporting both.

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. We changed the OpenJML contract to include the array length assertion and the verification no longer held so we believe the issue may be with the tool itself and the implementation was developed to account for this.

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.

Table 1: Binary Search Overview

|  |  |  |  |
| --- | --- | --- | --- |
| Binary Search | OpenJML | KeY | Krakatoa |
| Total Lines of Code | 32 | 32 | 52 |
| Lines of Implementation | 25 | 26 | 15 |
| Lines of Specification | 7 | 6 | 37 |
| Classes | 1 | 1 | 1 |
| Methods | 1 | 1 | 1 |
| Quantifiers | 4 | 5 | 6 |
| Frame Conditions | 0 | 1 | 0 |
| Axioms | 0 | 0 | 0 |
| Predicates | N/A | 0 | 1 |
| Lemmas | N/A | 0 | 3 |
| Pragmas | N/A | 0 | 1 |
| Type Checking Errors | None | N\A | N\A |
| RAC errors | None | N\A | N\A |
| SMT-Solver | z3 | z3 | Alt-Ergo |
| Verification Results | Valid | Valid | Valid |
| Proof Time | 0.7 secs | 5.697 secs | 0.21 secs |
| Proof Obligations | 2 | 1 | 6 |
| Nodes | N/A | 2877 | N/A |
| Branches | N/A | 40 | N/A |
| Total Rules applied | Unknown | 4527 | Unknown |
| Automatic Steps | All | All | All |
| Interactive Steps | N/A | None | N/A |
| Counter Examples | None | None | None |

### 5.2.2 PrefixSum

#### 5.2.2.1 Attempt 1 (See Chapter 4…)

We encounter issues that we expect 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 encounter issues that we did not see coming such as the inability to cancel 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 will not recognise the Z3 back-end solver even when specifically set in the preferences section of OpenJML. The list of solvers has to be altered so that Z3 was set as the default version in order for the system to recognise it.

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

This modularisation approach is also the approach used for specifying the Krakatoa version with verification applied to a method once the same method has passed the verification process in OpenJML. The OpenJML implementations and specifications will be used as the template to begin the Krakatoa code as the versions of JML are somewhat similar, however a number of changes are required to match the KML syntax, as stated in Table XYZ.

#### 5.2.2.2 Attempt 2 (See Chapter 4…)

The Z3 error we encounter in this attempt is troublesome due to the specification developed by the KeY team making use of this quantifier 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, could prove 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 at the expense of more advanced users.

The Krakatoa version of this implementation differs greatly with regards to the ‘even’ method with a predicate (L2-5) required to replace this methods functionality. This is needed as we discover, within Krakatoa, we are unable to call methods from within specifications and therefore the ‘*evenSumLemma*’ method’s specification causes a syntax error. As such, verification is only required for the ‘*evenSumLemma*’ method and verifies correctly once the ‘*even*’ predicate is developed.

The original KeY implementations (LX-Y) of these methods have been proven to verify by the KeY developers however once we run this method the automatic verifier cannot verify it complete leaving four open for both methods. These require interactive verification through the application of the symbolic rules however this process is not intuitive in nature and there are not enough tutorials on this process in order the adequately verify these methods. This is a major problem as the two methods in question, ‘*even*’ and ‘*evenSumLemma*’, are simple functions that will surely be the easier of the verification tasks at hand, so the failure of our attempts is worrying with greater complexity ahead.

Table 3: even / evenSumLemma properties (per Tool)

|  |  |  |  |
| --- | --- | --- | --- |
| even / evenSumLemma | OpenJML | KeY | Krakatoa |
| Total Lines of Code | 8 / 10 | 17 / 9 | 3 / 8 |
| Lines of Implementation | 3 / 3 | 4 / 3 | 0 / 3 |
| Lines of Specification | 5 / 7 | 13 / 6 | 3 / 5 |
| Classes | N/A | N/A | N/A |
| Methods | 2 | 2 | 0 / 1 |
| Preconditions | 0 / 0 | 0 / 0 | N/A / 0 |
| Postconditions | 1 / 2 | 2 / 2 | N/A / 1 |
| Loop Invariants | 0 / 0 | 0 / 0 | N/A / 0 |
| Loop Variants | 0 / 0 | 0 / 0 | N/A / 0 |
| Pure | Yes / Yes | Yes / Yes | N/A |
| Helper | Yes / Yes | Yes / Yes | N/A |
| Spec\_Public | Yes / Yes | N/A | N/A |
| Model Method | No / No | No / No | No / No |
| Model Variables | No / No | No / No | No / No |
| Ghost Variables | No / No | No /No | No / No |
| \forall quantifier | 0 / 1 | 0 / 1 | 0 / 1 |
| \exists quantifier | 0 / 0 | 2 / 0 | 0 / 0 |
| \product quantifier | N/A | 0 / 0 | 0 / 0 |
| \sum quantifier | N/A | 0 / 0 | 0 / 0 |
| \max quantifer | N/A | 0 / 0 | 0 / 0 |
| \min quantifier | N/A | 0 / 0 | 0 / 0 |
| Frame Conditions | 1 / 1 | 1 / 1 | 0 / 1 |
| Axioms | 0 / 1 | 0 / 1 | 0 / 1 |
| Predicates | N/A | 0 / 0 | 1 / 0 |
| Lemmas | N/A | 0 / 0 | 0 / 0 |
| Pragmas | N/A | 0 / 0 | 0 / 0 |
| SMT-Solver | z3 | z3 | Alt-Ergo |
| Verification | Valid / Valid | Incomplete/ Incomplete | Valid / N/A |
| Proof Time | 1000ms | 239ms / 805ms | N/A / 60ms |
| Proof Obligations | 1 / 1 | 2 / 2 | 0 / 3 |
| Nodes | N/A | 89 / 174 | N/A |
| Branches | N/A | 5 / 7 | N/A |
| Total Rules applied | Unknown | 164 / 296 | N/A / 35 |
| Automatic Verification | All | Partial / Partial | N/A / All |
| Interactive Verification | N/A | Required / Required | N/A / None |
| Counter Examples | 0 / 0 | 4 / 4 | N/A / 0 |

#### 5.2.2.3 Attempt 3 (See Chapter 4…)

The ESC verification of the 'div2' and 'leftMost' methods was the easiest in the program. The *ArithmeticOperationRange* error for the 'div' is returning invalid results for specifications that are valid, which may potentially mean there is a flaw in the proof system and also the vice versa could potentially happen; invalid results could being returned valid. As this is the only occurrence of this type of action, we believe it an anomaly but it still should be noted. The 'leftMost' methods upper bound issue is easy to fix with an upper bound of Integer.MAX\_VALUE, and later after further specification errors changed to Integer.MAX\_VALUE/2 variable.

Only minor modifications are required for the Krakatoa versions of these methods (L.XYZ) with the use of the earlier ‘is\_Even’ predicate required for valid verification.

KeY only needs to verify the div2 method, as no contract is specified on the ‘leftMost’ method, however again provides the same issue as previously with the automatic verifier only working so far. Interactive verification is again required to finish the proof with two open goals but we are unable to apply the rules in any fashion to make sufficient progress.

Table 4: div2 / leftMost properties (per Tool)

|  |  |  |  |
| --- | --- | --- | --- |
| div2 / leftMost | OpenJML | KeY | Krakatoa |
| Total Lines of Code | 12 / 9 | 12 / 4 | 8 / 9 |
| Lines of Implementation | 3 / 3 | 3 / 3 | 3 / 3 |
| Lines of Specification | 9 / 6 | 9 / 1 | 5 / 6 |
| Classes | N/A | N/A | N/A |
| Methods | 2 | 2 | 2 |
| Preconditions | 2 / 3 | 2 / 0 | 1 / 3 |
| Postconditions | 3 / 0 | 3 / 0 | 1 / 1 |
| Loop Invariants | 0 / 0 | 0 / 0 | 0 / 0 |
| Loop Variants | 0 / 0 | 0 / 0 | 0 / 0 |
| Pure | Yes / Yes | Yes / Yes | N/A |
| Helper | Yes / Yes | Yes / Yes | N/A |
| Spec\_Public | Yes / Yes | N/A | N/A |
| Model Method | No / No | No / No | No / No |
| Model Variables | No / No | No / No | No / No |
| Ghost Variables | No / No | No / No | No / No |
| \forall quantifier | 0 / 0 | 0 / 0 | 0 / 0 |
| \exists quantifier | 0 / 0 | 0 / 0 | 0 / 0 |
| \product quantifier | N/A | 0 / 0 | N/A |
| \sum quantifier | N/A | 0 / 0 | N/A |
| \max quantifer | N/A | 0 / 0 | N/A |
| \min quantifier | N/A | 0 / 0 | N/A |
| Frame Conditions | 1 / 0 | 1 / 0 | 1 / 1 |
| Axioms | 0 / 0 | 0 / 0 | 0 / 0 |
| Predicates | N/A | 0 / 0 | 0 / 0 |
| Lemmas | N/A | 0 / 0 | 0 / 0 |
| Pragmas | N/A | 0 / 0 | 0 / 0 |
| SMT-Solver | z3 | z3 | Alt-Ergo |
| Verification | Valid / Valid | Incomplete / N/A (No contract) | Valid / Valid |
| Proof Time | 1100ms | 279ms / N/A | 80ms / 40ms |
| Proof Obligations | 1 / 1 | 2 / N/A | 3 / 2 |
| Nodes | N/A | 117 / N/A | N/A |
| Branches | N/A | 4 / N/A | N/A |
| Total Rules applied | Unknown | 198 / N/A | 42 / 28 |
| Automatic Steps | All | Partial / N/A | All / All |
| Interactive Steps | N/A | Required / N/A | None / None |
| Counter Examples | 0 / 0 | 2 / N/A | 0 / 0 |

#### 5.2.2.4 Attempt 4 (See Chapter 4…)

This Is an important method in that we discovered a number of critical faults with the OpenJML environment. Recursive implementations such as the, ‘pow2’ method, do not always hold as the termination clause has not been fully implemented to resolve the inductive process. This means all future recursive methods will require refactoring to iterative implementations in order to guarantee a valid specification.

It also shows that the ability to prove a method is satisfiable may require the use of techniques, such as the model method *‘\_isPow2’*, that restrict the intended usability of the code. This may not be acceptable with certain methods or programming practices and brings forward the idea that specification may not always be a viable option or at least the best option for certain methods. In such situations, systematic testing may provide users with enough assurances for functionality of methods in such situations.

Krakatoa also requires an iterative implementation for ‘isPow2’ however it requires the ‘is\_Pow2’ predicate to be created in order for the specification syntax to be correct. This functionality matches the ‘\_isPow2’ model method created in OpenJML, however the same specification cannot be used as the ‘*\old’* clause is not being recognised as a keyword within this version of Krakatoa, despite the documentation stating it is supported. This results in the specification failing and no valid verification being determined.

KeY, once again, returned an incomplete verification through its automatic verifier however the interactive process is significantly more difficult with 10027 nodes created, 10972 rules applied and still 16 open goals to be further verified. This highlights the issue that, even if we managed to verify the earlier methods, this version would require quite extensive expertise beyond any standard software developers grasp and would be solvable, most likely, only by the KeY development team themselves.

Table 5: isPow2 / \_isPow2 (OpenJML only) properties (per Tool)

|  |  |  |  |
| --- | --- | --- | --- |
| isPow2 / \_isPow2 | OpenJML | KeY | Krakatoa |
| Total Lines of Code | 22 / 2 | 18 / N/A | 24 / N/A |
| Lines of Implementation | 11 / 0 | 8 | 12 |
| Lines of Specification | 11 / 2 | 10 | 12 |
| Classes | N/A | N/A | N/A |
| Methods | 2 | 1 | 1 |
| Preconditions | 1 / 0 | 1 | 1 |
| Postconditions | 2 / 1 | 2 | 1 |
| Loop Invariants | 2 / 0 | 0 | 2 |
| Loop Variants | 1 / 0 | 1 (Recursive) | 1 |
| Pure | Yes / Yes | Yes | N/A |
| Helper | Yes / Yes | Yes | N/A |
| Spec\_Public | Yes / N/A | N/A | N/A |
| Model Method | No / Yes | No | No |
| Model Variables | No / No | No | No |
| Ghost Variables | No / No | No | No |
| \forall quantifier | 0 | 0 | 0 |
| \exists quantifier | 0 | 1 | 0 |
| \product quantifier | N/A | 1 | N/A |
| \sum quantifier | N/A | 0 | N/A |
| \max quantifer | N/A | 0 | N/A |
| \min quantifier | N/A | 0 | N/A |
| Frame Conditions | 0 | 1 | 1 |
| Axioms | 0 | 0 | 0 |
| Predicates | N/A | 0 | 1 |
| Lemmas | N/A | 0 | 0 |
| Pragmas | N/A | 0 | 0 |
| SMT-Solver | z3 | z3 | Alt-Ergo, Z3 |
| Verification | Valid / N/A | Incomplete | Incomplete (\old not allowed) |
| Proof Time | 1400ms | 33252ms | 36220ms |
| Proof Obligations | 1 / 0 | 2 | 3 |
| Nodes | N/A | 10027 | N/A |
| Branches | N/A | 27 | N/A |
| Total Rules applied | Unknown | 10972 | 58 |
| Automatic Steps | All | Partial | All |
| Interactive Steps | N/A | Required | None |
| Counter Examples | 0 / N/A | 16 | 1 |

#### 5.2.2.5 Attempt 5 (See Chapter 4…)

All of the OpenJML issues found in this example implied that the verification of the 'pow2' method could seem impossible and did result in mass amounts of time being wasted in pursuit of a resolution. It was learned from the OpenJML developers that although the \product quantifier is used in OpenJML, its implementation is no completed and as such is running an unknown process returning random values. This means this quantifier must be removed from the specification.

The restrictions imposed on the ‘pow2’ method due to the lack of a fully functioning recursive mechanism resulted in the development of an iterative implementation. This implementation however must rely on the model method *‘\_isPow2’* (L…) in the specification, that also does not have recursive qualities implemented in OpenJML as of yet, and results in the constraining of the codes functionality. This restriction in fcuntionality means the method can only return values set by the model methods specification and severely reduces the codes usability. All the remaining unverified methods in the program use the ‘pow2’ implementation or specification to some degree and as such the issues occurring from ‘pow2’ specification will have a knock-on effect. This means a decision has to be made to determine what to do moving forward and we decide to officially end the verification process for the overall PrefixSum algorithm. This decision was made due to that, even if all the remaining methods could be verified, the codes ability has been handicapped by the earlier ‘pow2’ implementation and thus the overall program would not have been useful or deployable.

Due to the issue with the OpenJML program, we chose to implement the ‘pow2’ method in Krakatoa and stop verifying once complete, as it is there as a form of comparison only. The ‘pow2’ method specification L(…) failed to verify due to the need to use the is\_Pow2 predicate along with the \old clause that, as discovered in the previous attempt, is not supported by this version of KML.

The KeY tool uses the recursive implementation, as it is support, but like the previous results requires user interaction to resolve the one open goal remaining, once the automatic verifier has finished.

Table 6: pow2 / mult2 (OpenJML only) properties per tool

|  |  |  |  |
| --- | --- | --- | --- |
| pow2 / mult2 | OpenJML | KeY | Krakatoa |
| Total Lines of Code | 26 / 10 | 11 / N/A | 23 / N/A |
| Lines of Implementation | 12 / 3 | 3 | 13 |
| Lines of Specification | 14 / 7 | 8 | 10 |
| Classes | N/A | N/A | N/A |
| Methods | 2 | 1 | 1 |
| Preconditions | 1 / 1 | 1 | 1 |
| Postconditions | 1 / 3 | 2 | 1 |
| Loop Invariants | 3 / 0 | 0 | 1 |
| Loop Variants | 1 / 0 | 1 (Recursive) | 1 |
| Pure | No / Yes | Yes | N/A |
| Helper | No / Yes | Yes | N/A |
| Spec\_Public | Yes / Yes | N/A | N/A |
| Model Method | No / No | 0 | No |
| Model Variables | No / No | 0 | No |
| Ghost Variables | No / No | 0 | No |
| \forall quantifier | 0 | 0 | 0 |
| \exists quantifier | 0 | 0 | 0 |
| \product quantifier | N/A | 1 | 0 |
| \sum quantifier | N/A | 0 | 0 |
| \max quantifer | N/A | 0 | 0 |
| \min quantifier | N/A | 0 | 0 |
| Frame Conditions | 1 / 0 | 2 | 1 |
| Axioms | 0 / 0 | 1 | 0 |
| Predicates | N/A | 0 | 0 |
| Lemmas | N/A | 0 | 0 |
| Pragmas | N/A | 0 | 0 |
| SMT-Solver | z3 | z3 | Alt-Ergo |
| Verification | Valid / Valid | Incomplete | Incomplete (\old not allowed) |
| Proof Time | 1800ms | 250ms | 60ms |
| Proof Obligations | 1 / 1 | 4 | 3 |
| Nodes | N/A | 234 | N/A |
| Branches | N/A | 5 | N/A |
| Total Rules applied | Unknown | 404 | 47 |
| Automatic Steps | All | Partial | All |
| Interactive Steps | N/A | Required | None |
| Counter-Examples | 2 / 0 | 1 | 0 |

### 5.2.3 Longest Repeated Substring

Even though we could not verify this program as we had intended starting out, we still managed to discover numerous errors within the OpenJML environment that seemed to enforce a growing theme that the OpenJML tool is simply not ready for distribution or adoption by industry or academia personnel and requires major alterations and JML extensions to keep up with the more mature KeY tool. The OpenJML developers do provide sufficient support when needed and respond in timely fashion however the development team is quite small and the tool requires substantial commitment and resources to mature to a fully-fledged verification tool. All further verification steps for the Longest Repeated Substring were abandoned due to various factors such as time constraints and the constant barriers to verification that these issues provide.

We decided not to implement this program in Krakatoa as we determine there is no real means of comparison for this program against OpenJML due to the issues that have occurred. We also choose to skip the KeY verification process for this program as we can assume interactive verification will be required and our frustration with this tool has only increased with every use.

## 5.3 Verification Tools - Analysis

### 5.3.1 JML Dialects

Table 2: PrefixSum Keywords (per Tool)

|  |  |  |  |
| --- | --- | --- | --- |
| PrefixSum - Keywords | OpenJML | KeY | Krakatoa |
| requires |  |  |  |
| ensures |  |  |  |
| loop\_invariant |  |  |  |
| maintaining |  |  |  |
| loop\_variant |  |  |  |
| decreases |  |  |  |
| invariant |  |  |  |
| measured\_by |  |  |  |
| assume |  |  |  |
| assert |  |  |  |
| assignable |  |  |  |
| assigns |  |  |  |
| accessible |  |  |  |
| behaviour |  |  |  |
| \old |  |  |  |
| \forall |  |  |  |
| \exists |  |  |  |
| \product |  |  |  |
| \sum |  |  |  |
| \max |  |  |  |
| \min |  |  |  |
| \seq |  |  |  |
| axioms |  |  |  |
| predicates |  |  |  |
| lemmas |  |  |  |
| pragmas |  |  |  |
| pure |  |  |  |
| strictly\_pure |  |  |  |
| helper |  |  |  |
| spec\_public |  |  |  |
| \nothing |  |  |  |
| \strictly\_nothing |  |  |  |
| Model method |  |  |  |
| Model variable |  |  |  |
| Ghost variable |  |  |  |
| normal\_behaviour |  |  |  |
| behavior |  |  |  |
| label |  |  |  |
| \infinite\_union |  |  |  |
| \singleton |  |  |  |

#### 5.3.2 Tool Properties

|  |  |  |  |
| --- | --- | --- | --- |
| PROPERTIES | OpenJML | KeY | Why3 (Krakatoa) |
| Language/s | Java 8 | Java 1.2 | Java / WhyML |
| JML version/s | Standard JML | Extended JML | KML (limited version of JML) |
| IVL | Unknown from documentation | JavaDL contracts | Jessie / WhyML |
| Theories and Libraries | No | Yes | Yes |
| Verification | Automated | Interactive and Automated | Interactive and Automated |
| Theorem Solvers allowed | One | One | Multiple |
| Interactive Verification | No | Yes | No |
| Automatic Verification | Yes | Yes | Yes |
| Logics | Hoare  First-Order Predicate  Propositional | Hoare  First-Order Predicate  Propositional  JavaDL | Hoare  First-Order Predicate  Propositional  WhyML |
| Specification Modes | Valid  Invalid  Infeasible  Timeout  Error  Skipped | Closed Goal  Open Goal | Behaviour  Safety |
| IDE’s | Eclipse | KeY IDE | Why3 IDE |
| Standard Defaults |  |  |  |
| IDE Tool Features | * Type Checking * RAC * ESC * Selection of Solver * Set Preferences * Colour Highlighting * Counter-Examples * Log Traces | * Symbolic Debugger * Selection of Solver * Interactive Verification * Set Default Rules * View Open Goals * First-Order logic formula generation | * Automatic Verification options * Code selection for proof * Selection of Solvers * Colour Highlighting * Model creation |
| Proof Obligation creation technique | VCG | SE | VCG |
| Issues | Environment and Tool not complete | Complex interactive verification process | Krakatoa no longer under development.  WhyML unintuitive to non-functional programmers |

## 5.4 Tool Recommendations

# Chapter Six: Evaluation

This chapter outline how we evaluated the specifications of the Case Studies in Chapter 4, specifically focusing on the OpenJML tool and the implications of these case studies on its validity as a viable verification tool for the future.

* Start with KeY implementation vs your own
* Build KeY-JML to OpenJML translator, viable
* Join OpenJML with Why3 for SMT-Solver selection with proper JML

## 6.2 Overview

We set out to verify two major case studies using the OpenJML tool with previously developed and verified KeY implementations as a guideline. We set out to not change any implementation details but to perform refactoring on the specifications themselves, therefore preserving the core functionality and keeping a consistent code skeleton to perform the specifications on. We would use an initial smaller case study as an example of how the different tools differed with regards to JML syntax and implementation styles.

## 6.3 BinarySearch

The Binary Search algorithm provided us with a basis for describing how the changes between the JML syntaxes was in our opinion unnecessary and a hindrance to any future unification of these versions of JML. We also managed to show how the specifications can differ from tool to tool, however we believe now that we should have used a single Binary Search implementation and built the specifications for each tool around this, providing a more stable basis for comparison. We would ratify this mistake by using the KeY implementations for the two larger case studies and try to adhere to the code skeleton as much as is possible with changes made primarily to the specifications where necessary.

This case study also showed how much detail is required to express the inner working of the specifications and with greater difficulty to com, the detail would surely grow. This brought out the main contribution of this case study, the creation of a table (Table 1, Chapter 4) where we could visibly see the difference between each programs properties and with this we believed we provided a basis for comparison that could be used throughout our project.

## 6.4 PrefixSum

The goal of the PrefixSum case study was to show how effective OpenJML can be, even when working with a complex solution with even more complex specifications. We determined that the only way to work through such an example was to work on a modular basisas the errors as a whole were too much too handle. Managing each individual method’s specification as we went proved relatively fruitful with the alterations being made providing valuable data, captured by our analysis.

From this case study we determined:

* z3 solver cannot support quantified formulas and the inability to change solvers in OpenJML results in valid specifications having to be removed.
* The bounding of all variables is crucial to avoid type boundary errors and unnecessary counter-examples
* Method recursion is not fully implemented in the specifications with no loop variant available to determine termination
* Iterative implementations, replacing the recursive versions, still could not be adequately verified with model methods acting as a form of defensive programming being deemed the most viable option
* Only the \forall and \exists first-order quantifiers are available in OpenJML with \product, \min, \max and \sum not yet supported
* Specification affects implementation, constraining program overall usability

The list of issues above resulted in the verification of the PrefixSum case study to falter, however we deem it to be a success none the less as we have provided the OpenJML developers with numerous examples of reproducible errors that can only help further the development of the tool. We have showed that KeY specifications can be used to create OpenJML specifications with certain alterations and have not been forced to change the non-recursive code implementations.

An expertise in specifications does however help, as seen with the isPow2 method in which despite my best efforts I could not get verified. The OpenJML developers were able to provide a valid specification for this within 24 hours and it showed the difference between a computer science student with programming experience and a developer focused purely on verification. This is the same type of expertise that the developers in the VerifyThis competitions also have and the gap in knowledge is hard to bridge due to the use of verification tools not being wholly widespread throughout industry and academia. OpenJML is trying to be in essence the solution to this problem however I have just encountered these same roadblocks using this tool with verification proving beyond my level of expertise.

## 6.5 Longest Repeated Substring

The verification of the Longest Repeated Substring case study was restricted from the beginning due to the errors occured in the PrefixSum case study. We would have liked to provide a full implementation of this cases study due to its use of object oriented programming and the connectivity of the specifications on one class from the others. We still however managed to provide vital feedback to the OpenJML developers regarding unexpected internal errors as well as ESC termination problems.

From this case study we determined:

* Internal RAC error
* Inability to stop ESC during complex loop verifications
* Inability to stop ESC on specifications with concatenated quantifiers

The verification of this algorithm and the use of OpenJML was not explored to the full extent that we had set out to achieve however through this work two new updates, versions 0.8.29 and 0.8.31, were made to the OpenJML tool which will benefit future users within the Eclipse IDE environment.

## 6.6 OpenJML Tool

We have determined from our analysis that the OpenJML tool is not developed to a level where it is a viable competitor to other more established verification tools, as of yet. There are still development bugs within the tool and not all required JML functionality is currently implemented to match other mature verifiers such as KeY. In my/our opinion, after working through the case studies and analysing the benefits and issue, we believe the OpenJML tool could potentially benefit from collaboration with other developers, specifically with regards to a potential integration to Why3. An OpenJML front-end, integrated to work with the Why3 tool replacing the now defunct Krakatoa, could be beneficial providing the wider range of the JML version used in OpenJML combined with the use of multiple back-end solvers using the best know transformations provided by the Why3 automated verifier. Integration to Why3 is a more realistic option due, compared to KeY, due to the similar process in producing the VC’s using VCG (Section 2) with the main bulk of the work being the translation of the Java code and JML specifications to the IVL Jessie or directly to WhyML. However due to the history of a lack of communication between developers within these verification departments and the Why3 developer’s goal being to further their own WhyML language within their own tool, this collaboration does not seem likely. Due to this we cannot say that OpenJML, at this point in development, can provide an alternative to match the current, matured, verification systems that are available.

## 6.7 Project Approach and Assessment

Judging my own work can prove difficult in this project due to the difficult nature of deductive verification and its steep learning curve, particularly when moving from one tool to another with no standard JML version set in any. It appears that the developers of the tools themselves have the best chance of completing these algorithms from the VerifyThis competitions over the years and even then, they struggled with the tasks at hand and required additional time and manpower to complete a fully verified program (Source VerifyThis papers). However, I believe I could have done better with my verification processes, as I did not manage to get any of the two large case studies verified within the OpenJML tool. I could perhaps have made use of the OpenJML developers’s expertise more often, however that would have made the point of this project mute as it would resulted in a verification from a tool developer once more.

I believe my approach to use implementations and specifications already created was the right idea as we wanted to focus on OpenJML’s capabilities primarily and not on the codes functionality. Also the approach to modularise the specification process, I believe, proved correct and allowed myself to collect important data and knowledge as to how the tool operated with the JML version available.

The inability to terminate long proofs cost the project many, many hours of lost time and I should have made this a priority immediately to the OpenJML developers as an earlier update for this would have perhaps allowed me to catch other issues earlier, perhaps get at least the PrefixSum algorithm verified in OpenJML and allow more time to work on the Krakatoa version. A valid verification in OpenJML and Krakatoa would have given more weight to the table developed in Chapter 4 for comparing the case study across the tools and would have provided at least a partial benchmark for future researchers to work within.

# Chapter Seven: 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 5, 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 5: 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 6: Krakatoa Binary Search



|  |
| --- |
|  |
| Figure 7: Krakatoa - Jessie Model – Binary Search |
| Figure 8: Krakatoa - Jessie Model - Binary Search Safety    Figure 9: Krakatoa - Jessie Model - Verified |

###### KeY



Figure 10: KeY - Binary Search

Figure 11: KeY IDE



Figure 12: KeY IDE - Binary Search - Proof





Figure 13: KeY IDE- Binary Search - Rules

###### OpenJML



Figure 14: OpenJML - Binary Search

Figure 15: OpenJML - Eclipse - Valid Verification





Figure 16: OpenJML - Eclipse - RAC



Figure 17: OpenJML - Eclipse - TypeCheck

Figure 18: OpenJML - Eclipse - ESC Error



## Case Study 2

### PrefixSum

#### KeY Implementation

Figure 19: KeY - PrefixSum



#### OpenJML

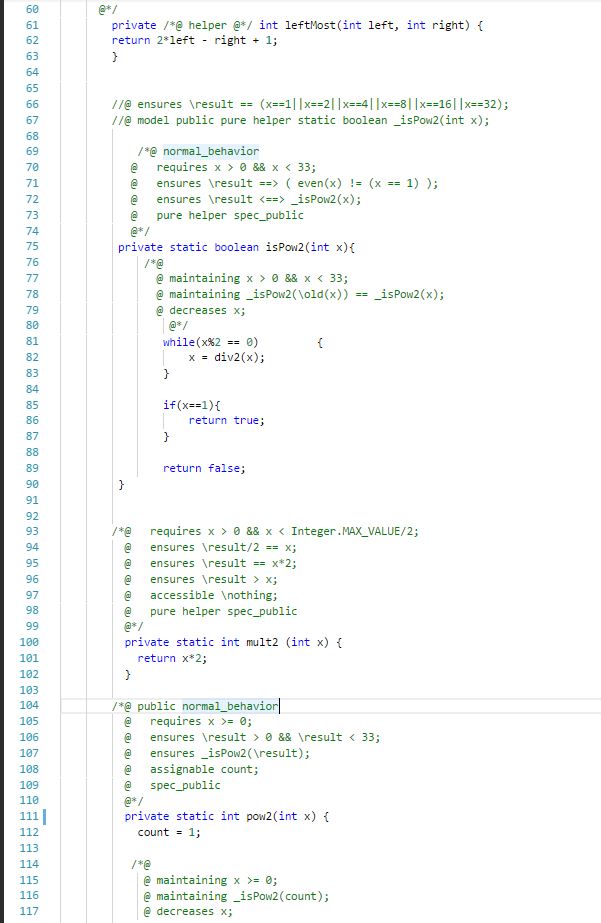
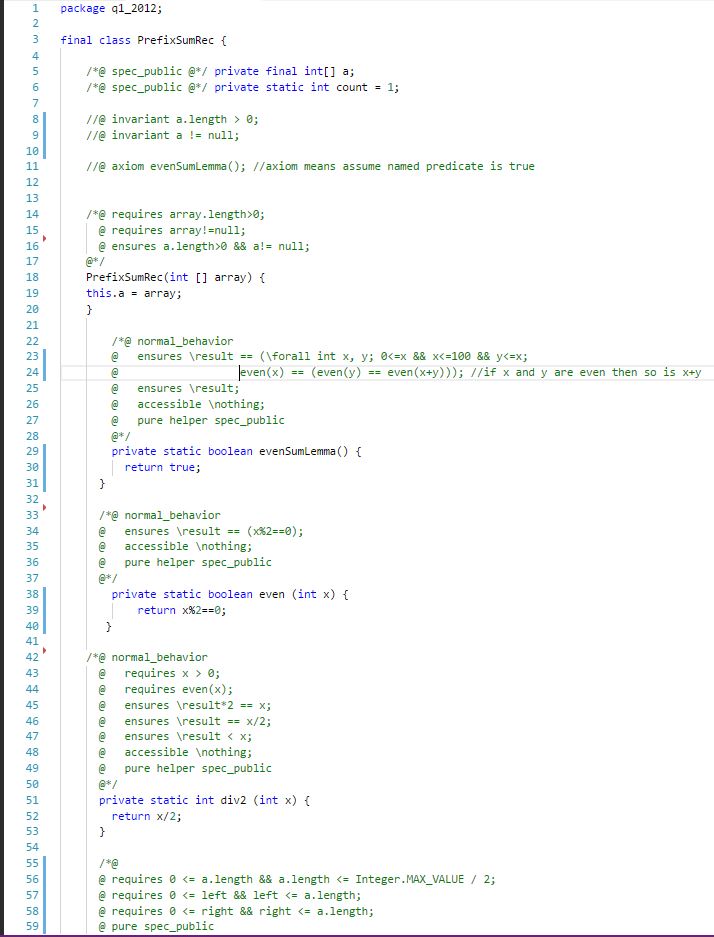


Figure 20: OpenJML - PrefixSum

#### Krakatoa

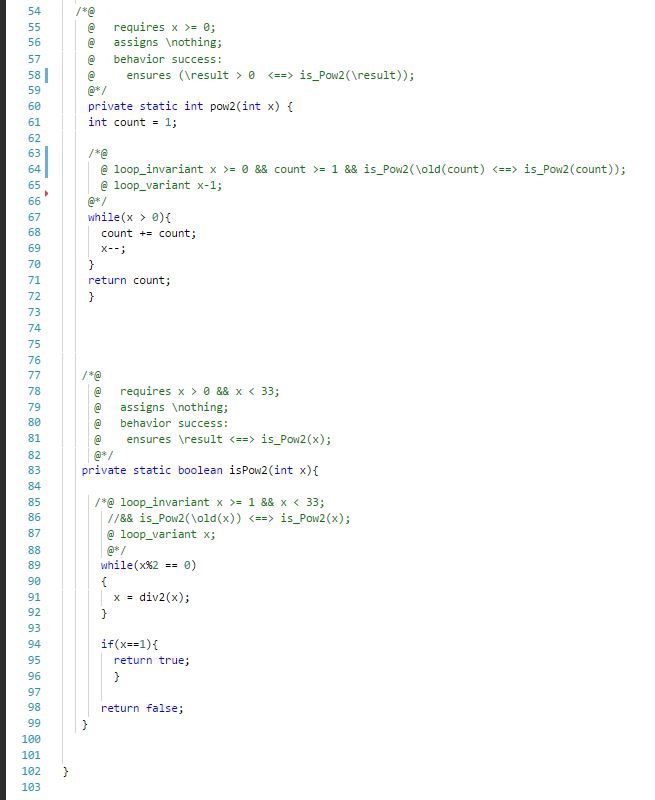
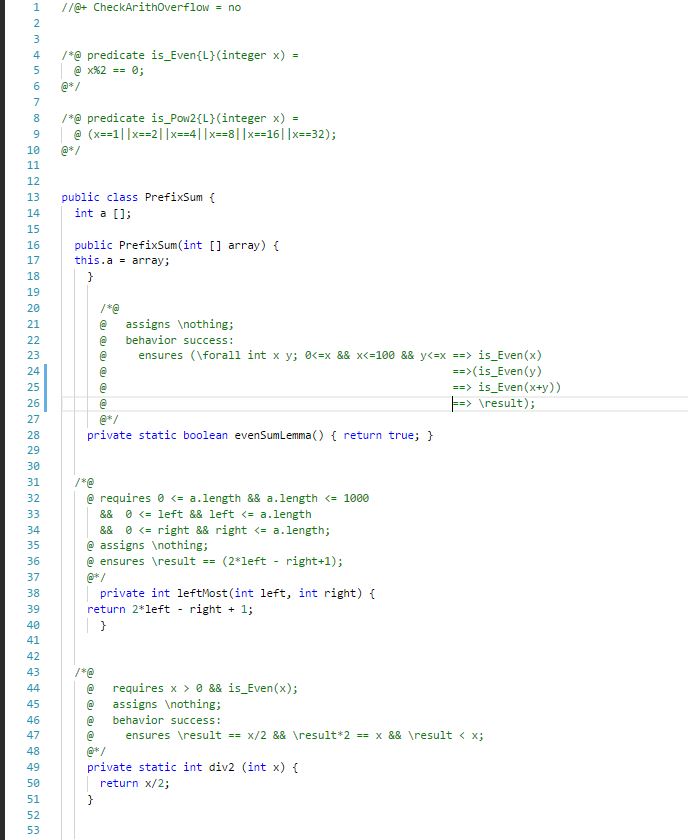


Figure 22: Krakatoa - PrefixSum



Figure 23: KeY- Longest Repeated Substring - Lemmas



Figure 24: KeY- Longest Repeated Substring - LCP



Figure 25: KeY- Longest Repeated Substring - LRS

Figure 26: KeY- Longest Repeated Substring - SuffixArray





Figure 27: OpenJML - Longest Repeated Substring - LCP

Figure 28: OpenJML - Longest Repeated Substring - Lemmas





Figure 29: OpenJML - Longest Repeated Substring - LRS

Figure 30: : OpenJML - Longest Repeated Substring - SuffixArray





Figure 31: OpenJML - RAC internal error

## 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.