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

Enda O'Shea

Dissertation 2018

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: 23375 (including code snippets)

Table of Contents

Title

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

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: 18/06/2018

Acknowledgements

I would like to give special thanks to my supervisor Rosemary Monahan for providing myself with the vast amount of knowledge she has within the Software Verification community and providing myself with constant assistance and feedback throughout this project. I would like to give her special thanks however for taking me on as a student at all, and being my supervisor at such a late stage in the thesis process and for having fate in my abilities to achieve this goal. Without your trust and expertise, this project would not have happened and I am very grateful for your help.
Another special thanks has to go to David R. Cok, developer of the OpenJML tool, who has provided crucial assistance and feedback throughout this project's lifespan. I'd like to give thanks to David for his time and patience when working through example specifications that just would not work and I hope to provide David with some substantial results that he can use to improve the tool in the future.

Finally I want to thank my family and friends who have helped me during my year in Maynooth and provided me with great assistance and company throughout.

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

List of Tables

Chapter One: Introduction

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. (2018a)* 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 (Tobias Nipkow, J. (2018)), Why3 (Why3.lri.fr. (2018c)), KIV *(Reif, W., Schellhorn, G., Stenzel, K. , & Balser. M. (1998))* and Verifast *(Jacobs, B., Smans, J. & Piessens, F. (2010))*, with also KeY (Key-project.org. (2018b)) and Dafny *(Leino, K. R. M. (2010))* 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 (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 (4.2) was used to determine the differences in JML syntax and execution of the verification process between these three tools.

The following two case studies (PrefixSum 4.3, Longest Repeated Substring 4.4) 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 (5.1) 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 (7.1) 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.

Due to our research, two updates were made to the OpenJML environment that fixed an internal RAC error as well as allowing for the interactive cancellation of long complex proofs within the OpenJML plugin for the Eclipse IDE.

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 supporting documentation and tutorials are not substantial, with only one paper providing a detailed description of such a process *(Ahrendt, W., Beckert, B., Hähnle, R., Rümmer, P. & Schmitt, P.H. (2007))*.

Chapter Two: Related Work

This chapter outlines the characteristics of software verification and its key properties as well as detailing the three tools that we will be using throughout out project.

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.

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.

2.2 Logics

The main types of logics used for software verification are first-order predicate logic *(Yang, K.H., Olson, D. & Kim, J. (2004*, propositional logic *(Filliâtre, J. (2011))* and Hoare logic *(Hoare, C. (1983))*.

2.3 Design by Contract

Design by Contract *(Meyer, B. (1992))*, 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.

2.4 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 against 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.

2.5 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 errors that are present, or completeness, may warn of errors that are not possible, as it is not a fully fledged verification tool but rather an additive to RAC and testing procedures used by programmers *(Kiniry, J., Morkan, A. & Denby, B. (2006))*.

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

Verification tools such as OpenJML are based on the design of ESC/Java2 and built upon its structure.

2.6 Java Modelling Language (JML)

2.6.1 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))*.

2.6.2 JML Syntax

The JML syntax and capabilities change based on the verification tool being used, leading to a large subset of JML dialects being designed by developers to be optimised for their specific verification tool. This has reduced the capability of users to easily change verification.

An example of JML is shown in Figure 1 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 it does not cover all of JML's functionality or the adaptions of JML.

JML Example

```
1  public class ArraySearchWhile { // Listing 7.14
2
3      /*@ normal_behavior
4        @ requires a != null;
5        @ ensures \result == (\exists int i;
6        @                           0 <= i && i < a.length; a[i] == val);
7        @*/
8      public boolean search(int[] a, int val) {
9          int i = 0;
10         /*@ maintaining !(\exists int j;  0 <= j && j < i; a[j] == val);
11           @ maintaining 0 <= i && i <= a.length;
12           @ decreasing a.length - i;
13           @*/
14         while (i < a.length) {
15             if (a[i] == val)
16                 return true;
17             i++;
18         }
19         return false;
20     }
21
22 }
```

Figure 1: KeY Array-Search Loop Example

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

L3: normal behavior indicates that if the method functions correctly, the following specifications have to hold

L4: requires indicates the precondition of the contract that must be satisfied by the client for the method to execute correctly

L4: a != null is the constraint placed on the precondition that states the array '*a*' must not be null.

L5: ensures indicates the postcondition that must be satisfied by the execution of the method implemented by the supplier.

L5: \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

L5-6:\result == (\exists int i; 0 <= i && i < a.length; a[i] == val); is the constraint put on the postcondition 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.

L10: 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

L10: !(\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

L11: maintaining 0 <= i && i <= a.length; maintains loop variable values as valid while in the loop and indicates the value that realises the termination of the loop

L14: decreasing indicate the loop variant (called loop_variant in Krakatoa JML version) that ensures that the loop terminates by reducing with each loop iteration

L14: a.length – i; states that with the counter 'i' increasing with every iteration (L17: *i++)* it will eventually break the '*while(i < a.length)'* statement ensuring loop termination if '*val*' is not found.

2.6.3 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. 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))*.

```
Line 1   class SimpleProtocol {
Line 2
Line 3          //@ private ProtocolStack st;
Line 4          //@ boolean model started;
Line 5          //@ represents started <-- (st!=null);
Line 6          //@ requires !started;
Line 7          //@ ensures started;
Line 8          void startProtocol() { ... }
Line 9
Line 10         //@ requires started;
Line 11         //@ ensures !started;
Line 12         void endProtocol() { ... }
Line 13  }
```
(Cs.ru.nl. (2018d))

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))*.

```
Line 1  class SimpleProtocol {
Line 2
Line 3          //@ boolean ghost started;
Line 4          //@ requires !started;
Line 5          //@ ensures started;
Line 6          void startProtocol() {
Line 7          ...
Line 8          //@ set started = true;
Line 9          }
Line 10
Line 11         //@ requires started;
Line 12         //@ ensures !started;
Line 13         void endProtocol() {
Line 14         ...
Line 15         //@ set started = false;
Line 16         }
Line 17   }
```
(Cs.ru.nl. (2018d))

2.6.4 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 becoming 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, however Symbolic Execution can handle such a statement.

2.7 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))*. A Why3 tool plugin called Jessie also exists that takes Java and FramaC languages in as input and translates them to the intermediate language of WhyML which is then further translated to VCs using VCGs *(Kosmatov, N., Marché, C., Moy, Y. & Signoles. J. (2016))*.

2.8 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))*.

2.9 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 *(Nieuwenhuis, R., Oliveras, A. & Tinelli, C. (2006))*.

This technique is sued with the KeY tool and within this, 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.

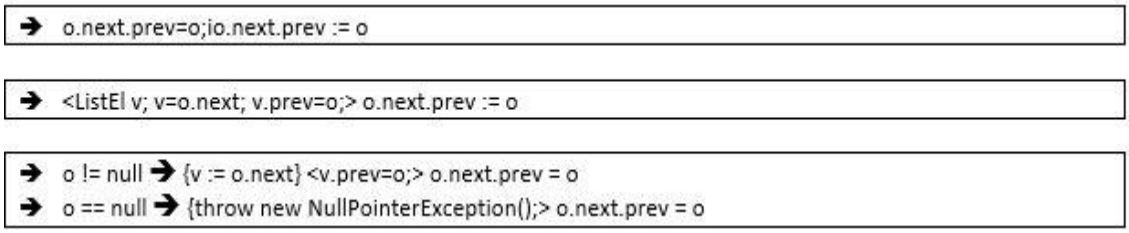


Figure 2: Symbolic Execution with Case Distinction

(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 iterations. '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))*. An example of a symbolic execution process working on multiple paths is attached in Appendices (Figure-7).

2.10 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 generated by certain tools, especially those using first-order arithmetic due to this arithmetic's incompleteness *(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))* resulting in an inability to get sound or complete proofs.

2.11 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++.

2.12 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 un-satisfiable (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))*.

2.13 Satisfiability Modulo Theories (SMT)

Satisfiability Modulo Theories (SMT) solvers extends SAT solvers 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 *(Smtlib.cs.uiowa.edu. (2018))*. 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 types*" (Healy, A.(2016)).

Chapter Three: Tools

This chapter discusses the three verification tools used throughout this project. Each provide the capability to use the Java programming language for implementations and the Java Modelling Language for specifications.

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 (2.12, 2.13).

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))*.

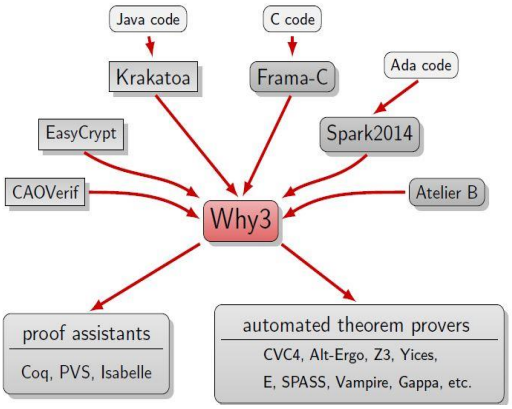


Figure 3: Why3 Platform

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

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))*.

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 (2.8) 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 (2.6)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.

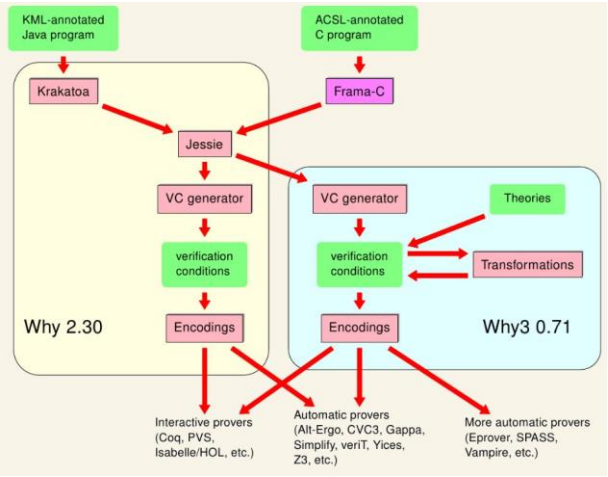


Figure 4: Krakatoa with Why tool

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

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, Wolfram Menzel, and Peter Schmitt 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))*.

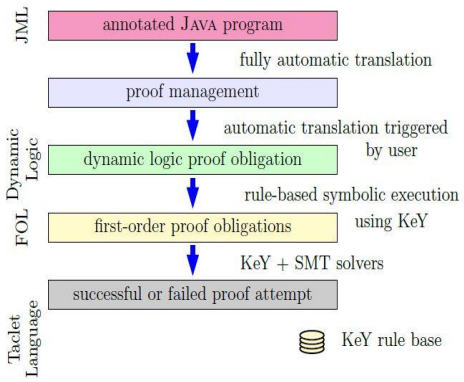


Figure 5: The KeY Verification Workflow

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

Java Dynamic Logic is the basis of the KeY logic system. The syntax of JavaDL extended first-order logic with program variables and program modalities *(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))*, and was designed to match the Java type system to reduce the learning curve required when using the tool. JavaDL uses a Kripke structure to evaluate formulas to determine valid paths and models 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. The modal operator, '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 (Sere Chapter 2.6) due to this arithmetic being incomplete *(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))*. Relative completeness, however, is possible meaning all proofs are capable of being proven with the exception of some proofs that require specific first-order arithmetic operations that are not covered.

The JML specifications (2.6)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 *(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))*. 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))*.

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 libraries 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 link for the Eclipse plugin could not found. The KeY IDE also has an automated feature which will automatically selects 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 deduction steps 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 (2.3) , 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 (2.8), it uses the symbolic execution (2.9).

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))*.

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. The incompleteness of logical theories, such as first-order arithmetic *(Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R. Schmitt, P., & Ulbrich, M. (2016))*, may result in invalid counterexamples being produced by the SMT solvers, particularly when quantifiers are used in the specifications *(Cok, D.R. (2014))* . Static verification is done on a modular basis with each method's specification and feasibility being checked independently.

The OpenJML tool itself checks that the JML specification satisfies the Java implementation using the Design by Contract paradigm and is therefore valid. OpenJML is designed in a similar fashion to other deductive verification systems with it being adapted from the ESC/Java2 system *(Cok, D.R. (2014))* with the process of determining the validity of program specifications started by translating the specifications into assertions and assumptions interleaved with the Java code. These constructs then generate Verification Conditions (VCG), expressed in SMTLIBv2 *(Smtlib.cs.uiowa.edu. (2018))* with a single VC generated for each method in a modular fashion. 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))*.

Chapter Four: Case Studies

The section outlines the specification and verification process for three cases studies all carried out with the tools discussed in Chapter 3. Binary Search is the first case study discussed, followed by PrefixSum and Longest Repeated Substring, both of which are taken from the VerifyThis 2012 competition.

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) seems to have the most complex implementation and specification due to its use of predicates, lemmas and pragmas (L2-20). 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))*.

```
Line 2  //@+ CheckArithOverflow = yes
Line 3
Line 4  /* lemma mean_property1 :
Line 5   @   \forall integer x y; x <= y ==> x <= (x+y)/2 <= y;
Line 6   @*/
Line 7
Line 8  /* lemma mean_property2 :
Line 9   @   \forall integer x y; x <= y ==> x <= x+(y-x)/2 <= y;
Line 10  @*/
Line 11
Line 12 /* lemma div2_property :
Line 13  @   \forall integer x; 0 <= x ==> 0 <= x/2 <= x;
Line 14  @*/
```

```
Line 15
Line 16 /*@ predicate is_sorted{L}(int[] t) =
Line 17  @   t != null &&
Line 18  @   \forall integer i j;
Line 19  @     0 <= i && i <= j && j < t.length ==> t[i] <= t[j] ;
Line 20  @*/
```

The specification for the binary_search method (L25-33) uses the standard JML annotations to setup the general contract with the precondition and postcondition using the usual '*requires*' and '*ensures*' clauses (L25-26). 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 (L27-28) and a deviation from normal behaviour specified (L29-32); the 'assumes' statement (L30) 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.

```
Line 25 /*@ requires t != null;
Line 26      @ ensures -1 <= \result < t.length;
Line 27      @ behavior success:
Line 28      @   ensures \result >= 0 ==> t[\result] == v;
Line 29      @ behavior failure:
Line 30      @  assumes is_sorted(t);
Line 31      @  ensures \result == -1 ==>
Line 32      @    \forall integer k; 0 <= k < t.length ==> t[k] != v;
Line 33      @*/
```

The loop invariants are setup (L36-40) and are specified by the '*loop_invariant*' clause. The statement (L36-37) must hold before and after a successful execution and termination of the while loop (L44-51). The successful termination of this while loop depends on the loop variant (L41-42) which checks that '*u*' decreases with each loop iteration. An additional inductive invariant is setup (L38-40) that must hold under the behavior specified (L29-32).

```
Line 36 /*@ loop_invariant
Line 37  @   0 <= l && u <= t.length - 1;
Line 38  @ for failure:
Line 39  @  loop_invariant
Line 40  @    \forall integer k; 0 <= k < t.length ==> t[k] == v ==> l <= k <= u;
Line 41  @ loop_variant
Line 42  @   u-l ;
Line 43  @*/
```

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-6). 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-7) that the postcondition was resulting in an Arithmetic Overflow. The pragma (L2) was then changed to `//@+ CheckArithOverflow = no` which resolved the issue and the program verified (Figure-8). Ideally a bound would be placed on the variables highlighted by the Why3 tool (Figure-9) 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 (Figure-10) 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 (L4,5). The '*requires*' and '*ensures*' clauses are part of the '*public normal_behaviour*' block, (L3-6), and are equivalent to the contract and '*behavior success:*' block in Krakatoa. This different style of applying the behaviour blocks seems unnecessarily different and is a prime example of the different versions of JML dialect employed by these tools and ideally would be standardised. 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.

```
Line 7  static /*@pure@*/ int search(int[] a, int v)
```

An issue with this implementation is that the code (L11,12) checks to see if the array lengths are greater than zero or equal to one and therefore check if the value if found.

```
Line 11 if(a.length == 0) return -1;
Line 12 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 (L20-29) to include the first array index.

The '*loop_invariant*' clause (L14-16) conjoins multiple quantifiers and assertions into one statement. An '*assignable*' clause (L17) 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.

```
Line 14 /*@ loop_invariant 0 <= l && l < r && r < a.length
Line 15 @                && (\forall int x; 0 <= x && x < l; a[x] < v)
Line 16 @                && (\forall int x; r < x && x < a.length; v < a[x]);
Line 17 @ assignable \nothing;
```

```
Line 18 @ decreases r - l;
Line 19 @*/
```

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 it 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. Once the KeY IDE has loaded the proof (Figure-11) the user has the option of 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-14). An example of these rules can be seen in Figure-15 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 (Figure-16) 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 (L4-6) ensuring if a match is found it is a positive value and returns -1 otherwise.

```
Line 4  //@ requires (\forall int i, j; 0 <= i && i < j && j < arr.length; arr[i] <= arr[j]);
Line 5  //@ ensures \result == -1 ==> (\forall int i; 0 <= i && i < arr.length; arr[i] != key);
Line 6  //@ 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 (L8-9) 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 (L14-16) are indicated with the '*maintaining*' keyword. The loop variant is introduced (L17) and, along with the KeY tool, used the '*decreases*' keyword.

```
Line 14 //@ maintaining 0 <= low && low <= high  && high <= arr.length && mid == low + (high - low) / 2;
Line 15 //@ maintaining (\forall int i; 0 <= i && i < low; arr[i] < key);
Line 16 //@ maintaining (\forall int i; high <= i && i < arr.length ==> key < arr[i]);
```

```
Line 17 //@ 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.

The supplied code, along with its specification, verified with no issues and a detailed description of the results was displayed to the user (Figure-17). The RAC functioned as expected with the results highlighted in blue (Figure-18). 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 (Figure-19). Finally, a change to the specification was made to the loop invariant (L16) by changing the last less than sign to a greater than sign as highlighted in the code snippet below.

```
Line 16 //@ 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-20). 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 (Figure-21) 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.

NOTE:

- All Code snippets and line numbers in the Code and Specification sections are taken from the original KeY implementation in Figure-21 and begin with line numbers coloured as such - Line xyz.
- OpenJML final version is in Figure-22 and code snippets begin with line numbers coloured as such – Line xyz.
- Krakatoa final version is in Figure-23 and code snippets begin with line numbers coloured as such – Line xyz.

4.3.2 Algorithm

The PrefixSum algorithm works on an array structure by storing the sum of the previous indices of the array into the current index. A detailed description of the algorithm is available at the VerifyThis 2012 achieve *(Pm.inf.ethz.ch. (2018b))*, however we are focusing on the specification process as opposed to the algorithmic functionality.

4.3.3 Attempt 1

4.3.3.1 Code and Specification

We do not alter the implementation for the Java code (Figure-21) 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 (L7) 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.

```
Line 7 //@ accessible \inv: \singleton(a);
```

The '*infinite_union*' clause (L127-128 and L156-157) was part of the set expressions introduced to JML in 2011 by Benjamin Weiß *(Weiß, B. (2011))* and has been translated to JavaDL (3.2) 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.

```
Line 127 @   assignable \infinite_union(int k; leftMost(left,right) <= k
Line 128      @              && k <= right && !even(k); \singleton(a[k]));


Line 156 @   assignable \infinite_union(int k; leftMost(left,right) <= k
Line 157      @                            && 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' (L19,129)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.

```
Line 19    @ strictly_pure                    →    @ pure spec_public
Line 97    @ assignable \strictly_nothing;    →    @ assignable \nothing
```

The PrefixSum class also has to be made public if an object of its type is to be used in a specification (L171-174), however we do not need this specification until we have completed the full verification of the PrefixSum class so can also be, temporarily, removed.

```
Line 170   /*@ normal_behavior
Line 171     @   requires \invariant_for(p) && p.a.length > 1;
Line 172     @   ensures (\forall int i; 0 <= i && i < p.a.length;
Line 173     @             p.a[i] == (\sum int j; 0 <= j && j < i;
Line 174     @                           \old(p.a[i])));
Line 175     @*/
```

The labels (L81,101 and L103,112) were a native KeY feature and as such are removed as their functionality is not used in OpenJML. Despite searching through our KeY documentation we could not find the reasoning behind these labels and their use within the tool. We assume they are used to help the prover in some fashion, however we cannot be certain.

```
Line 81  //ß\label{lst:min-begin}ß
Line 101 //ß\label{lst:min-end}ß


Line 103 //ß\label{lst:eff-begin}ß
Line 112 //ß\label{lst:eff-end}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.

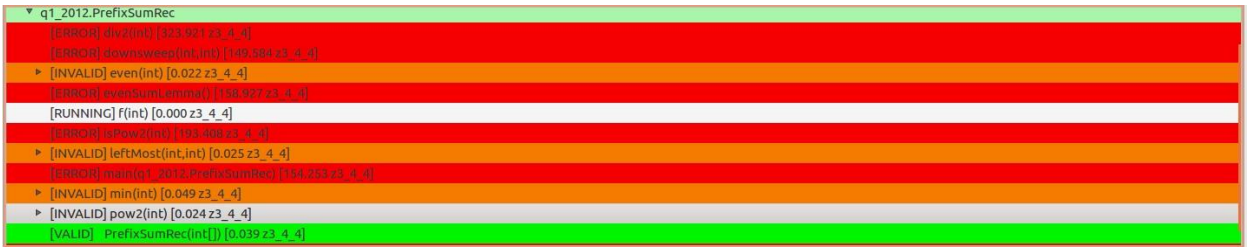


Figure 12: OpenJML - PrefixSum - ESC errors

4.3.4 Attempt 2

4.3.4.1 Code and Specification

The first method we chose from the PrefixSum class was '*evenSumLemma*' (L15-21). Line 21 of the '*evenSumLemma*' specification states that if 'x' is even and 'y' is even, then that implies 'x+y' is also even.

```
Line 15 /*@ normal_behavior
Line 16      @   ensures \result == (\forall int x, y; even(x) == (even(y) == even(x+y)));
Line 17      @   ensures \result;
Line 18      @   accessible \nothing;
Line 19      @ strictly_pure helper
Line 20      @*/
Line 21    private static boolean evenSumLemma() { return true; }
```

This required the '*even*' method (L66-74) to be verified beforehand.

```
 Line 66   /*@ normal_behavior
 Line 67     @   ensures \result == (\exists int y; y*2 == x);
 Line 68     @   ensures \result != (\exists int y; y*2 == x+1);
 Line 69     @   accessible \nothing;
 Line 70     @ strictly_pure helper
 Line 71     @*/
 Line 72   private static boolean even (int x) {
 Line 73       return x%2==0;
 Line 74   }
```

Line-9 is an axiom which tells the prover to assume the predicate named is true, which is the '*evenSumLemma*' method.

```
Line 9  //@ 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 (L67-68), 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 L67-68 replaced with a simpler clause taken from the method implementation itself which preserved the overall constraint.

```
~~Line 67 @   ensures \result == (\exists int y; y*2 == x);~~
~~Line 68 @   ensures \result != (\exists int y; y*2 == x+1);~~
Line 34 @   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.

```
Line 53  /*@ normal_behavior
Line 54      @   requires x > 0;
Line 55      @   requires even(x);
Line 56      @   ensures \result*2 == x;
Line 57      @   ensures \result == x/2;
Line 58      @   ensures \result < x;
Line 59      @   accessible \nothing;
Line 60      @   pure helper spec_public
Line 61      @*/
Line 62      private static int div2 (int x) {
Line 63        return x/2;
Line 64      }
Line 65
Line 66  /*@
Line 67      @ requires 0 <= a.length && a.length <= Integer.MAX_VALUE;
Line 68      @ requires 0 <= left && left <= a.length;
Line 69      @ requires 0 <= right && right <= a.length;
Line 70      @ pure spec_public
Line 71      @*/
Line 72        private /*@ helper @*/ int leftMost(int left, int right) {
Line 73        return 2*left - right + 1;
Line 74        }
```

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.

```
Line 56 @   ensures \result*2 == x;
```

The 'leftMost' method new specifications also has an *ArithmeticOperationRange* error during initial ESC verification checks, no upper bound restriction on the '*a.length*' variable is resulting in the upper bound of the integer type being broken after the multiplication operation (L73). Restricting the upper bound of '*a.length*' to '*Integer.MAX_VALUE / 2'* solves the issue.

```
~~Line 56 @ requires 0 <= a.length && a.length <= Integer.MAX_VALUE;~~
Line 56 @ 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 (L23-39) is the next method we add into our OpenJML implementation. The specification made use of recursion (L25) and uses the 'measured_by' clause as the basis for termination of the recursive method, which relies on parameter 'x'.

```
Line 23  /*@ public normal_behavior
Line 24      @   requires x > 0;
Line 25      @   ensures \result ==> ((even(x)  && isPow2(div2(x))) <=!=> x == 1);
Line 26      @   ensures \result == (\exists int b; 0 <= b;
Line 27      @                       x == (\product int i; 0 <= i && i < b; 2));
Line 28      @   measured_by x;
Line 29      @   accessible \nothing;
Line 30      @   pure helper
Line 31      @*/
Line 32    private static boolean isPow2(int x){
Line 33      if (x==1)
Line 34          return true;
Line 35      else if (x % 2 != 0 )
Line 36          return false;
Line 37      else
Line 38          return isPow2(x/2);
Line 39    }
```

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 program appears to 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-66 (Figure-22) is akin to defensive programming and as such can be deemed to be in breach of the non-redundancy principle (2.3) 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.

```
Line 66   //@ ensures \result == (x==1||x==2||x==4||x==8||x==16||x==32);
Line 67   //@ model public pure helper static boolean _isPow2(int x);
Line 68
Line 69   /*@ normal_behavior
Line 70    @   requires x > 0 && x < 33;
Line 71    @   ensures \result ==> ( even(x) != (x == 1) );
Line 72    @   ensures \result <==> _isPow2(x);
Line 73    @   pure helper spec_public
Line 74    @*/
Line 75  private static boolean isPow2(int x){
Line 76     /*@
Line 77       @ maintaining x > 0 && x < 33;
Line 78       @ maintaining _isPow2(\old(x)) == _isPow2(x);
Line 79       @ decreases x;
Line 80       @*/
Line 81      while(x%2 == 0)     {
Line 82        x = div2(x);
Line 83      }
Line 84
Line 85      if(x==1){
Line 86        return true;
Line 87      }
Line 88
Line 89      return false;
Line 90   }
```

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.

```
Line 41  /*@ normal_behavior
Line 42      @   requires x >= 0;
Line 43      @   ensures \result == (\product int i; 0 <= i && i < x; 2);
Line 44      @   ensures \result > x;
Line 45      @   accessible \nothing;
Line 46      @   measured_by x;
Line 47      @ strictly_pure helper
Line 48      @*/
Line 49    private static int pow2( int x ) {
Line 50      return x==0? 1: 2*pow2(x-1);
Line 51    }
```

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-43, 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.

```
Line 104 /*@ public normal_behavior
Line 105         @   requires x >= 0;
Line 106         @   ensures \result > 0 && \result < 33;
Line 107         @   ensures _isPow2(\result);
Line 108         @   assignable count;
Line 109         @   spec_public
Line 110         @*/
Line 111         private static int pow2(int x) {
Line 112           count = 1;
Line 113
Line 114           /*@
Line 115            @ maintaining x >= 0;
Line 116            @ maintaining _isPow2(count);
Line 117            @ decreases x;
Line 118           @*/
Line 119           while(x>0)
Line 120           {
Line 121            //@ assume x!=0;
Line 122           if(count < 33)
Line 123            count = mult2(count);
Line 124            x--;
Line 125           }
Line 126
Line 127           //@ assume x==0;
Line 128           return count;
Line 129         }
```

As we are using the iterative implementation we are again forced to use the model method *'_isPow2'* (L66-67) 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.

```
Line 93  /*@   requires x > 0 && x < Integer.MAX_VALUE/2;
Line 94    @   ensures \result/2 == x;
Line 95    @   ensures \result == x*2;
Line 96    @   ensures \result > x;
Line 97    @   accessible \nothing;
Line 98    @   pure helper spec_public
Line 99    @*/
Line 100    private static int mult2 (int x) {
Line 101      return x*2;
Line 102    }
```

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.

NOTE:

- All Code snippets and line numbers in the Code and Specification sections are taken from the original KeY implementation in Figure-24, 25, 26 & 27 and begin with line numbers coloured as such - Line xyz.
- OpenJML final version is in Figure-28, 29, 30 ,31 and begin with line numbers coloured as such - Line xyz.

4.4.1 Algorithm

The Longest Repeated Substring algorithm is used for text querying is sued to find repeated text values within a file or array and can be used to determine the longest repeated structure. For a more detail description, please see the VerifyThis 2012 archive *(Pm.inf.ethz.ch. (2018b)).*

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 passes the type-checker after some necessary visibility modifications similar to those taken in the PrefixSum case study (4.3.3.1) such as altering the 'strictly_pure' variables to 'pure' as well as adding '*spec_public*' to variables used in specifications. We encounter an unexpected error when we run the RAC function on the '*LRS*' class (Figure-30), an internal error occurs on execution (Figure-32). 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 (Figure-31). 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 is now possible however a warning symbol (Figure-33) now appears upon ESC execution. This warning symbol does not appear to have any physical ramifications for the tool however still requires investigation.

Chapter Five: Analysis

The analysis chapter discusses the specification and verification processes set out in Chapter 4, as well as discussing the OpenJML tool and its competitors overall properties.

5.1 Overview

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 bring as much of the core properties together into a table (Table-7) as well as discussing the JML dialects used within these tools (Table-1).

We also discuss the advantages and disadvantages of the OpenJML tool along with a list of recommendations for the developers to improve the user experience and usability.

5.2 Case Studies - Analysis

5.2.1 Binary Search

The first thing to note in the Binary Seach case study (4.2) 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 2: 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

We encounter issues that we expect in this phase (4.3.3), 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-3.

5.2.2.2 Attempt 2

The Z3 error we encounter in this attempt (4.3.4.2) 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 (L4-6) 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.

```
Line 4  /*@ predicate is_Even{L}(integer x) =
Line 5   @ x%2 == 0;
Line 6   @*/
```

The original KeY implementations (L15-21 & L66-74) of these methods have been proven to verify by the KeY developers however once we run this method the automatic verifier cannot verify it completely, leaving four open goals for both methods. These require interactive verification through the application of the symbolic rules, however this process is not intuitive in nature and we could not find adequate available tutorials on this process in order to 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 4: 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

The ESC verification of the 'div2' and 'leftMost' methods (4.3.5) 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 (L31-40 & L43-51) with the use of the earlier 'is_Even' predicate required for valid verification.

KeY only needs to verify the div2 method (L53-64), as no contract is specified on the 'leftMost' method (L76-80), 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.

```
Line 76    //@ strictly_pure
Line 77    private /*@ helper @*/ static int leftMost(int left, int right) {
Line 78    return 2*left - right + 1;
Line 79    }
```

Table 5: 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

This Is an important method in that we discovered a number of critical faults with the OpenJML environment (4.3.6). 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 (L8-10) 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 (L85-86) is not being recognised within the loop invariant 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.

```
Line 8  /*@ predicate is_Pow2{L}(integer x) =
Line 9   @ (x==1||x==2||x==4||x==8||x==16||x==32);
Line 10  @*/
```

```
Line 86  /*@ loop_invariant x >= 1 && x < 33;
Line 87  //&& is_Pow2(\old(x)) <==> is_Pow2(x);
```

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 6: 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

All of the OpenJML issues found in this example (4.3.7) 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'* (L66-67) 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 functionality 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 issues with the OpenJML tool halting the verification process, we chose to implement the 'pow2' method in Krakatoa and stop once complete, as it is there as a form of comparison only. The 'pow2' method specification (L54-61) 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 supported, but like the previous results requires user interaction to resolve one open goal remaining, once the automatic verifier has finished.

Table 7: 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 OpenJML Tool Review

5.3.1.1 Advantages

The OpenJML tool plugin for the Eclipse IDE helps developers keep the implementation, specification and verification process all within the one tool. The verification process is also very simple with a toolbar provided to execute type-checking, RAC and ESC without changing environment or compromising on other available IDE functionality. This allows the user to have all their development needs within the one environment, providing the user with the ability to do development, testing and verification within a single IDE.

Another advantage of the OpenJML tool is the simplification of the verification process. The user only has to install one of the four SMT solvers the tool supports and then set the solver as the default solver in the preferences section. Once this is complete, the ESC process takes care of the rest with the user given detailed output for the each specification on a modular basis, along with colour coded counter-examples guiding the user to issues within the specifications along with data models to reproduce such errors. This forces the user to use more simplified specifications however complex specifications can be very difficult to prove as seen within the KeY environment (5.2.2.2) using their interactive system

5.3.1.2 Disadvantages

The main disadvantage of this tool is that it is not fully developed and does not employ all the available qualities of the JML language that are available to it. These include the lack of recursion, model recursion and quantifiers, which resulted in the modification of code implementations (5.2.2.4). This lack of functionality and impact on implementation code restricts the user's ability to use this tool moving forward and needs to be addresses.

Another disadvantage is that 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 the developers *(Cok, D.R. (2016))*, however the document is still in its early stages, with the majority of material yet to be added. As some functionality is not developed as of yet, the users need up to date documentation to determine what is doable and achievable with the tool before deciding to use it for their specifications.

5.3.1.3 Recommendations

We believe the following recommendations for the OpenJML tool are required:

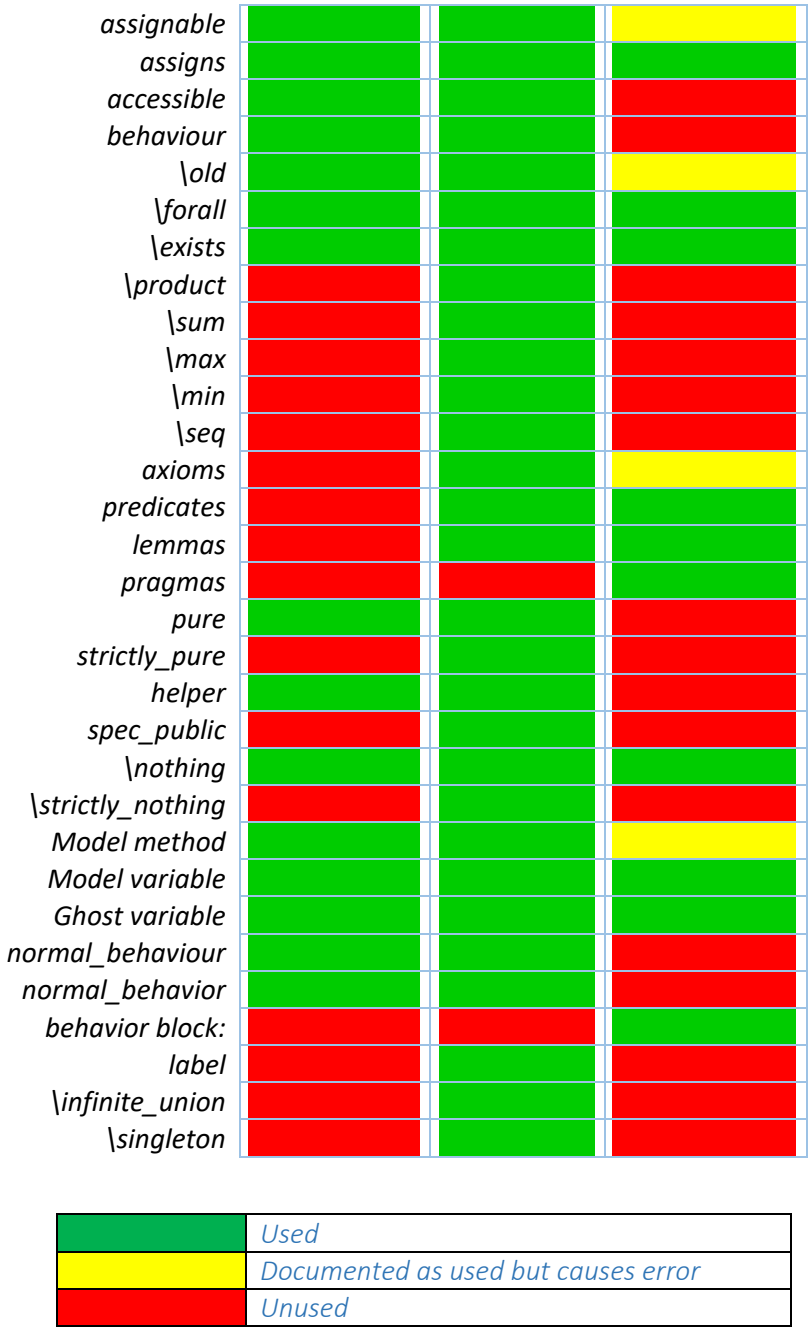
1. Implementation of recursive specification
2. Implementation of recursive model specifications
3. Implementation of all quantifiers
4. ESC toolbar functionality has to be fixed within Eclipse IDE as it currently does not operate and must be set in Preferences section
5. Preferences set by user should stay set upon relaunching of IDE and not restore to default settings
6. Correct RAC internal error (Update 0.8.29 fixed this)
7. Interactive cancellation of long proofs (Update 0.8.31 fixed this)
8. Fix error warning message appearing at ESC execution
9. Collaborate with other verification tool developers and use a set published standard dialect of JML
10. Complete documentation manual for OpenJML
11. Perhaps develop an automatic tool, similar to Why3, that chooses the correct automatic solver for the program/method being proven. Choosing the correct solver is invariably difficult for users as they don't know the internal workings of these systems. A previous Thesis proposed deals with this election issue *(Healy, A. (2016))*.

5.3.2 JML Dialects

Table-8 below shows the correlation between JML dialects across the three tools, with green representing clauses that are usable within said tool, red are unusable and yellow are supported clauses *(Marché, C., Paulin-Mohring, C. & Urbain, X. (2004))* but cause errors when used. These appear to change quite substantially from tool to tool with only the requires, ensures, assume, assert, ghost and model clauses applicable to all three versions. This leads to confusion from a specification standpoint as there is no standard JML dialect for users to work from. The loop invariant (maintaining, loop_invariant) and loop variant (loop_variant, decreases) clauses changing from tool to tool is perhaps the greatest issue as it is part of the crucial design by contract paradigm for method contracts and if standardised provides users with standard full contract clauses for method's throughout with the '*requires*' and '*ensures*' clauses already applicable. It should be noted that this table is a subset of JML clauses within these tools that were used in our case studies and more clauses are available such as signals and type clauses.

Table 9: PrefixSum Keywords (per Tool)

PrefixSum - Keywords	**OpenJML**	**KeY**	**Krakatoa**
requires			
ensures			
loop_invariant			
maintaining			
loop_variant			
decreases			
invariant			
measured_by			
assume			
assert			



5.3.3 Tool Properties

Table-7 below details the some of the important properties of the three main verification tools used within this project. It is of note that we did not find any other table of this type throughout our research and we believe it is vital that the developers of these tools create such a document to provide users and developers alike with the level of instrumental detail required. The information supplied has been gathered from the available documentation, websites and IDE's with all available sources having different levels of detail and as such, various attributes and properties could not be guaranteed for each tool and this is stated as such.

Table 10: OpenJML, KeY and Why3(Krakatoa) 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
Automatic Prover Tools available	z3 yipes2 cvc4 Simplify	Z3 No list of usable provers available	Z3 Alt-Ergo Simplify CVC3 E-Prover Gappa SPASS Vampire VeriT Yices
Interactive Prover tools available	N/A	Sequent Calculus using Taclets	Coq PVS Isabelle/HOL HOL4 HOL Light Mizar
Proof Tools per program	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	Unknown	Setup by User	Automatic Levels (Unknown detail for each process)

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

5.3.4 VerifyThis Competition Winners

Table 11: VerifyThis Competition Winners

VerifyThis Competition *(Pm.inf.ethz.ch. (2018a))*	**Best Team Winners**	**Tool**	**Developers**
2011	Gerhard Schellhorn, Gidon Ernst, Bogdan Tofan / Claude Marche	KIV / Why3	Yes, No, No / Yes
2012	Bart Jacobs, Jan Smalls	Verifast	Yes, Yes
2014	Data not available		
2015	Jean-Christophe Fillatre, Guillaume Melquiond	Why3	Yes, Yes
2016	Bart Jacobs	Verifast	Yes
2017	Jean-Christophe Fillatre	Why3	Yes

Chapter Six: Evaluation

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

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 keep a consistent code skeleton to perform the specifications on. We would use an initial smaller case study as an example of how the verification 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 come, the detail would surely grow. We decided to gather all the main verification properties of each program and display them with a table (Table-1) as a means of comparison as there was no precedent for comparing these tools and we believed this structure could then be useful in future case studies.

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 basis as 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: (5.2.2)

- 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 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 our best efforts we 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 (Table-8) 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 our 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 that occurred 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 (5.2.3):

- 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 (5.3). 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 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 front-end, could be beneficial providing the wider range of the JML dialect used in OpenJML combined with the use of multiple back-end solvers using the best known transformations available provided by the Why3 automated verifier. Integration to Why3 is a more realistic option due, compared to KeY, to the similar process in producing the VC's using VCG (2.8) with the main bulk of the work being the translation of the Java code and JML specifications to the plugin Jessie or directly to WhyML itself. 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 until the recommendations specified in Chapter 5.3.1.3 are implemented by the OpenJML developers.

6.7 Project Approach and Assessment

Judging our approach to this project has to be balanced against 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. The developers of the tools have historically performed the best at specifying the algorithms from the VerifyThis competitions (Table-8) 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 *(Bormer, T., Brockschmidt, M., et al. (2012))*. However, we believe we could have done better with the verification process, as we did not manage to get any of the two large case studies verified within the OpenJML tool. We could perhaps have made use of the OpenJML developers' expertise more often, however that would have made the point of this project mute as it would resulted in a specification from a tool developer once more.

We believe the approach to use implementations and specifications already created was correct as we wanted to focus primarily on OpenJML's capabilities as a specifier and verifier and not on the codes functionality. Also the approach to modularise the specification process , we believe, proved correct and allowed us to collect important data and knowledge as to how the tool operated with the JML dialect employed.

The inability to terminate long proofs cost the project many, many hours of lost time and we should have made this a priority immediately to the OpenJML developers at an earlier stage, for this would have perhaps allowed us to catch other issues earlier and perhaps get at least the PrefixSum algorithm verified in OpenJML. A valid verification in OpenJML and Krakatoa would have given more weight to the table developed in Chapter 4 (Table-1) 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

This chapter discusses the results and contributions of the project as a whole and as well as potential future work in this domain.

7.1 Contribution

Our contributions from this project was showing the deficiencies in the OpenJML verification tool and provide recommendations for further improvement within this tool, from a user's perspective. We provide a series of recommendations for the OpenJML tool (5.3.1.3) that can improve the user experience and the tool overall , two of which resulted in updates to the OpenJML framework itself.We explain how difficult it is for new user's to understand and apply the transitional rules in the KeY interactive tool as well as the limitations of the Krakatoa tool with the KML syntax they employ.

Our main contributions however were showing how difficult it is to get a standard across all the tools despite them all using Java and JML. All three tools used different versions of JML resulting in the specifications, of the same method implementations, being non-transferable and frustrating the user's due to the lack of documentation stating these differences with no sign of a standard being set moving forward.

7.2 Results

The results of the Binary Search case study shows the differences between the different versions of JML employed in each tool as well as the verification difficulties that can occur when involved in software verification (5.2.1). We believe we should have used the same Binary Search implementation when performing this comparison in order to provide unbiased results however we believe the results are still valid.

We have shown how the OpenJML tool is not yet a viable verification tool to compete with the more mature tools due to its lack of full JML functionality (5.2.2), however it does simplify the verification process in comparison to the interactive KeY tool and perhaps would benefit from the automatic solver section employed by Why3 (3.1). Also we conclude that the lack of core JML functionality in OpenJML requires that complete documentation is essential for users in order to understand what is allowable and usable within this structure (Analysis).

A threat to validity of this project is that our own experience with verification tools is minimal and our lack of interactive knowledge meaning we may have a bias against the KeY tool in parts. However, we believe that we represent the majority of the users of these systems in that our expertise is in software development or computer science and not is software verification, so the ability to learn and use these systems in a small period of time with medium to low difficulty is essential. The lack of a joint commitment in the software verification community, allied with the lack of documentation and standards, will cause, we believe, the tools to become more specialised and will hinder their expansion into industry and academia.

7.3 Project Approach

As was said previously in our evaluation (6.7), we believe our approach to modularise the specification process was correct and provided a rich and plentiful supply of comparative data from which to work with. Our main regret was not using the same implementation for the Binary Search case study however we believe the results are still viable and it provided the basis to use the same implementations for the two case studies that followed.

7.4 Future Work

Once the recursive and quantifier JML properties have been implemented by the OpenJML developers, the case studies could be specified once more with a viable opportunity to verify the entire case studies implementations without compromising the integrity of the code itself. We believe the incorporation of these JML segments is essential before any future work is carried out, as well as the creation of complete documentation to accompany all available specification processes.

A further more thorough comparison can be pursued between the tools themselves, not focusing on the specifications and verification but on the properties of the tools themselves. A complete document showing the inner workings of verification tools, along with the standard defaults employed by each would provide users with enormous benefit when starting out.

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.

Bormer, T., Brockschmidt, M., Distefano, D.S., Distefano, D., Ernst, G., Filliatre, J.-., Grigore, R., Huisman, M., Klebanov, V., Marche, C., Monahan, R., Mostowski, W., Poiikarpova, N., Scheben, C., Schellhorn, G., Tofan, B., Tschannen, J., Ulbrich, M., Beckert, B., Damiani, F. & Gurov, D. (2012), "The COST IC0701 Verification Competition 2011", Springer Verlag, , pp. 3.

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].

Cs.ru.nl. (2018d). [online] Available at: https://www.cs.ru.nl/E.Poll/talks/jml_advanced.pdf [Accessed 18 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.

Filliâtre, J. "One Logic To Use Them All." (2013) Maria Paola Bonacina. CADE 24 - the 24th International Conference on Automated Deduction, Jun 2013, Lake Placid, NY, United States. Springer.

Filliâtre, J., Paskevich, P. (2013) "Why3 – Where Programs Meet Provers." ESOP'13 22nd European Symposium on Programming, Mar 2013, Rome, Italy. Springer, 7792, LNCS.

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.

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. (2010), "A Quick Tour of the VeriFast Program Verifier" in Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 304-311.

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.

Kosmatov, N., Marché, C., Moy, Y. & Signoles. J. (2016) "Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014." 7th International Symposium on Leveraging Applications, Oct 2016, Corfu, Greece. Springer, 9952, pp.461–478, Lecture Notes in Computer Science.

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

Leino, K. R. M. (2010) Dafny: An automatic program verifier for functional correctness. In LPAR-16, volume 6355 of LNCS, pages 348–370. Springer,

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

Pm.inf.ethz.ch. (2018b). (2012). [online] Available at: http://www.pm.inf.ethz.ch/research/verifythis/Archive/2012.html [Accessed 18 Jun. 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.

Reif, W., Schellhorn, G., Stenzel, K. , & Balser. M. (1998) Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, Automated Deduction—A Basis for Applications, Vol II.1, pages 13 – 39. Kluwer.

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].

Smtlib.cs.uiowa.edu. (2018). SMT-LIB The Satisfiability Modulo Theories Library. [online] Available at: http://smtlib.cs.uiowa.edu/ [Accessed 18 Jun. 2018].

Tobias Nipkow, J. (2018). Isabelle. *[online] Isabelle.in.tum.de. Available at: https://isabelle.in.tum.de/ [Accessed 18 Jun. 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

Symbolic Execution

```
1 public static int min(int x, int y) {
2    if (x < y) {
3       return x;
4    }
5    else {
6       return y;
7    }
8 }
```

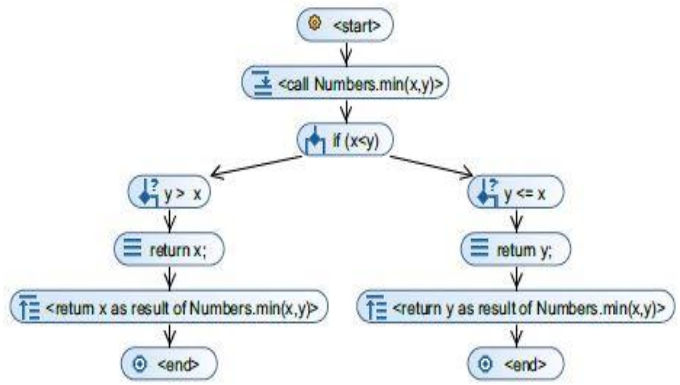


Figure 13: Symbolic Execution - min method

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

In Figure-7, 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.

Chapter 4

Case Study 1

Binary Search Examples

- Krakatoa

BinarySearch.java

```
//@+ CheckArithOverflow = no

/* 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] ;
   @*/


class BinarySearch {

    /*@ 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;
      @*/
    static int binary_search(int t[], int v) {
   int l = 0, u = t.length - 1;
   /*@ 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 ;
      @*/
   while (l <= u ) {
       int m;
           m = (u + l) / 2;
      //@assert l <= m <= u;
       if (t[m] < v) l = m + 1;
       else if (t[m] > v) u = m - 1;
       else return m;
   }
   return -1;
    }

}
```

Figure 14: Krakatoa Binary Search

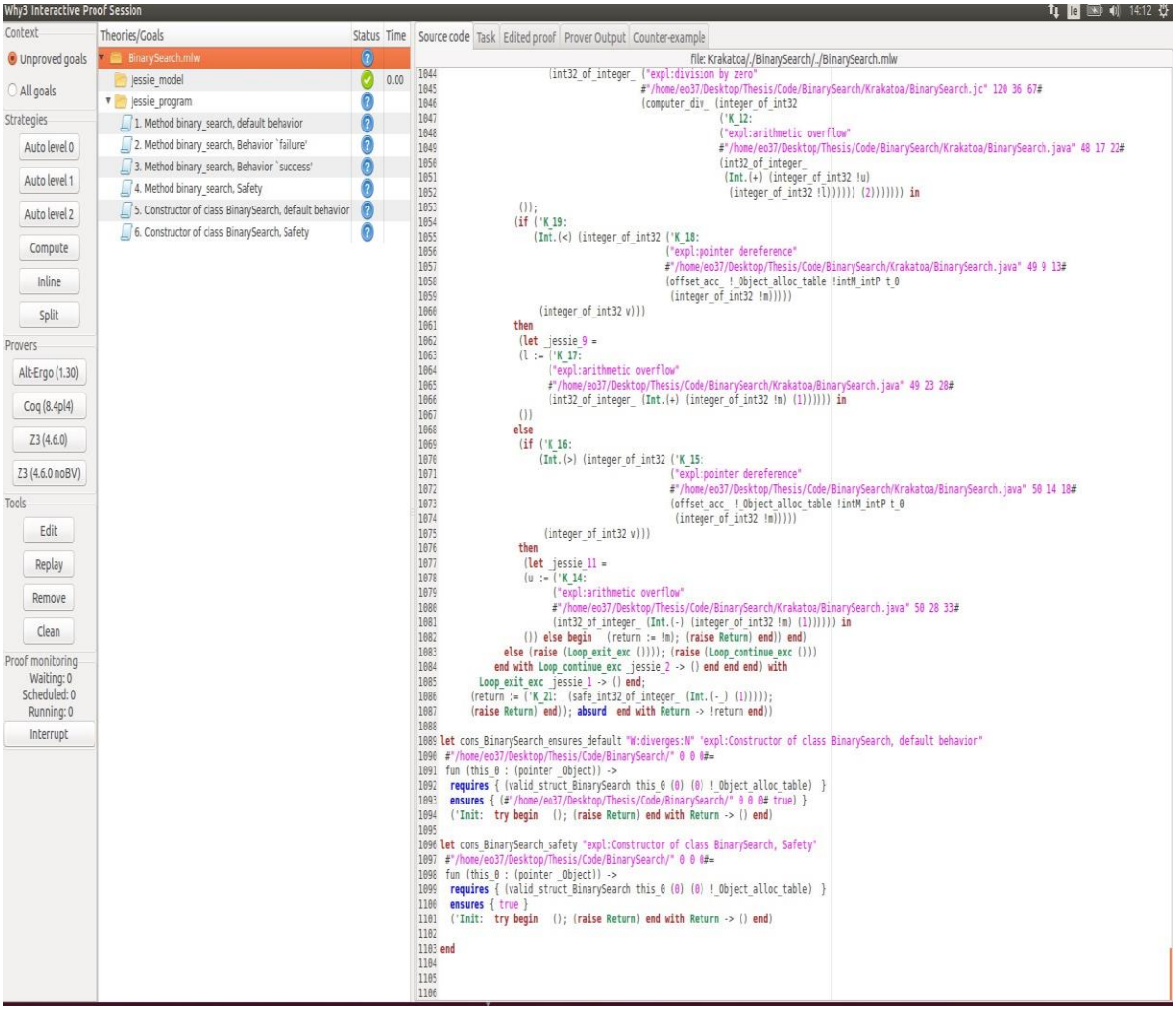


Figure 15: Krakatoa - Jessie Model – Binary Search

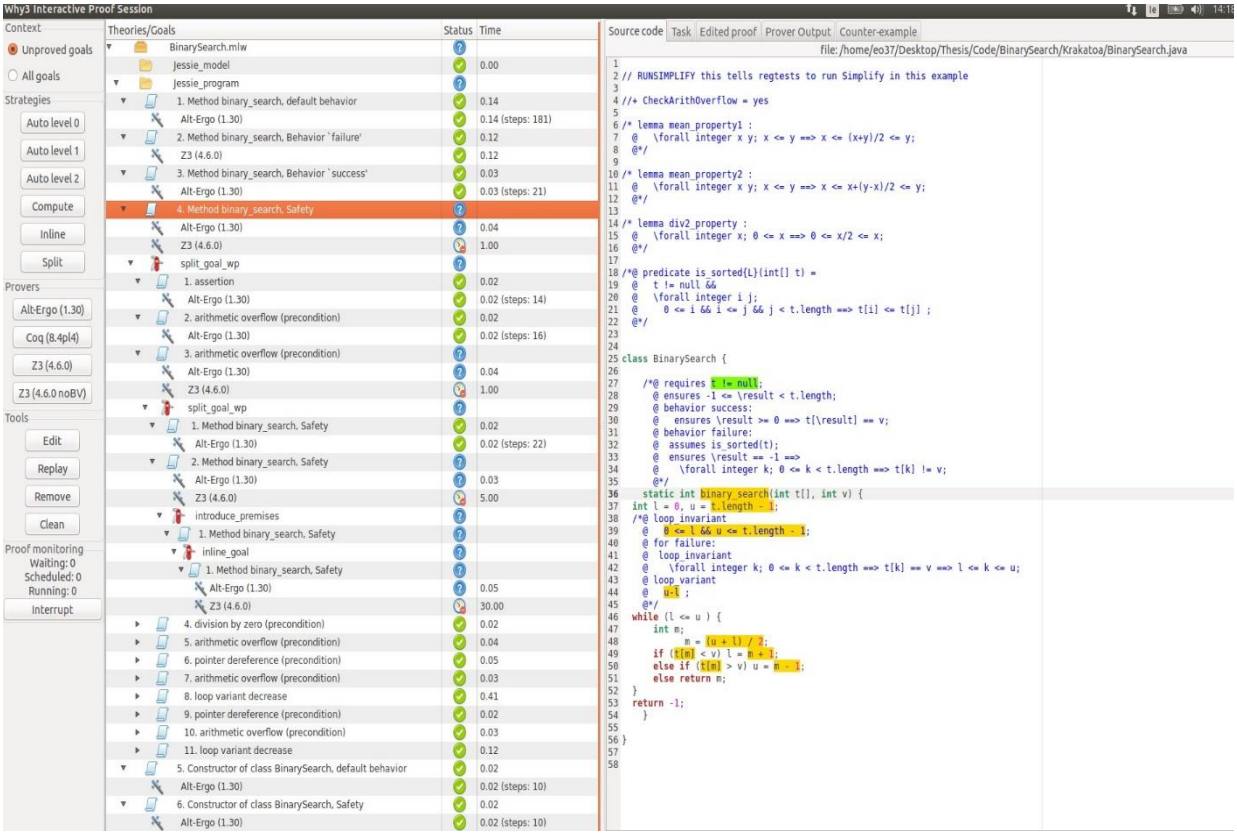


Figure 16: Krakatoa - Jessie Model - Binary Search Safety

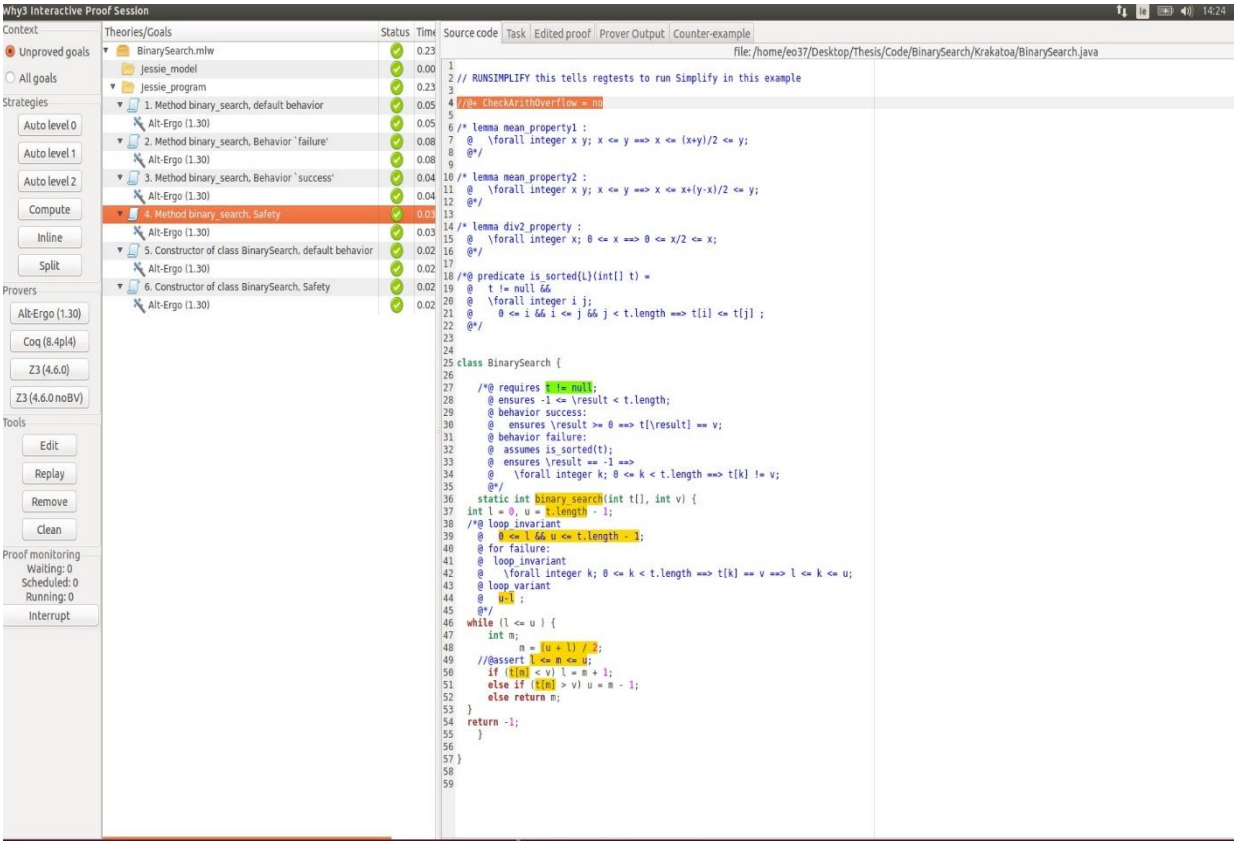


Figure 17: Krakatoa - Jessie Model - Verified

- KeY

BinarySearch.java

```
1  class BinarySearch {
2
3      /*@ public normal_behaviour
4        @   requires (\forall int x; (\forall int y; 0 <= x && x < y && y < a.length; a[x] <= a[y]));
5        @   ensures ((\exists int x; 0 <= x && x < a.length; a[x] == v) ? a[\result] == v : \result == -1);
6        @*/
7      static /*@pure@*/ int search(int[] a, int v) {
8          int l = 0;
9          int r = a.length - 1;
10
11         if(a.length == 0) return -1;
12         if(a.length == 1) return a[0] == v ? 0 : -1;
13
14         /*@ loop_invariant 0 <= l && l < r && r < a.length
15           @                && (\forall int x; 0 <= x && x < l; a[x] < v)
16           @                && (\forall int x; r < x && x < a.length; v < a[x]);
17           @ assignable \nothing;
18           @ decreases r - l;
19           @*/
20         while(r > l + 1) {
21             int mid = l + (r - l) / 2;
22             if(a[mid] == v) {
23                 return mid;
24             } else if(a[mid] > v) {
25                 r = mid;
26             } else {
27                 l = mid;
28             }
29         }
30
31         if(a[l] == v) return l;
32         if(a[r] == v) return r;
33         return -1;
34     }
35 }
```

Figure 18: KeY - Binary Search

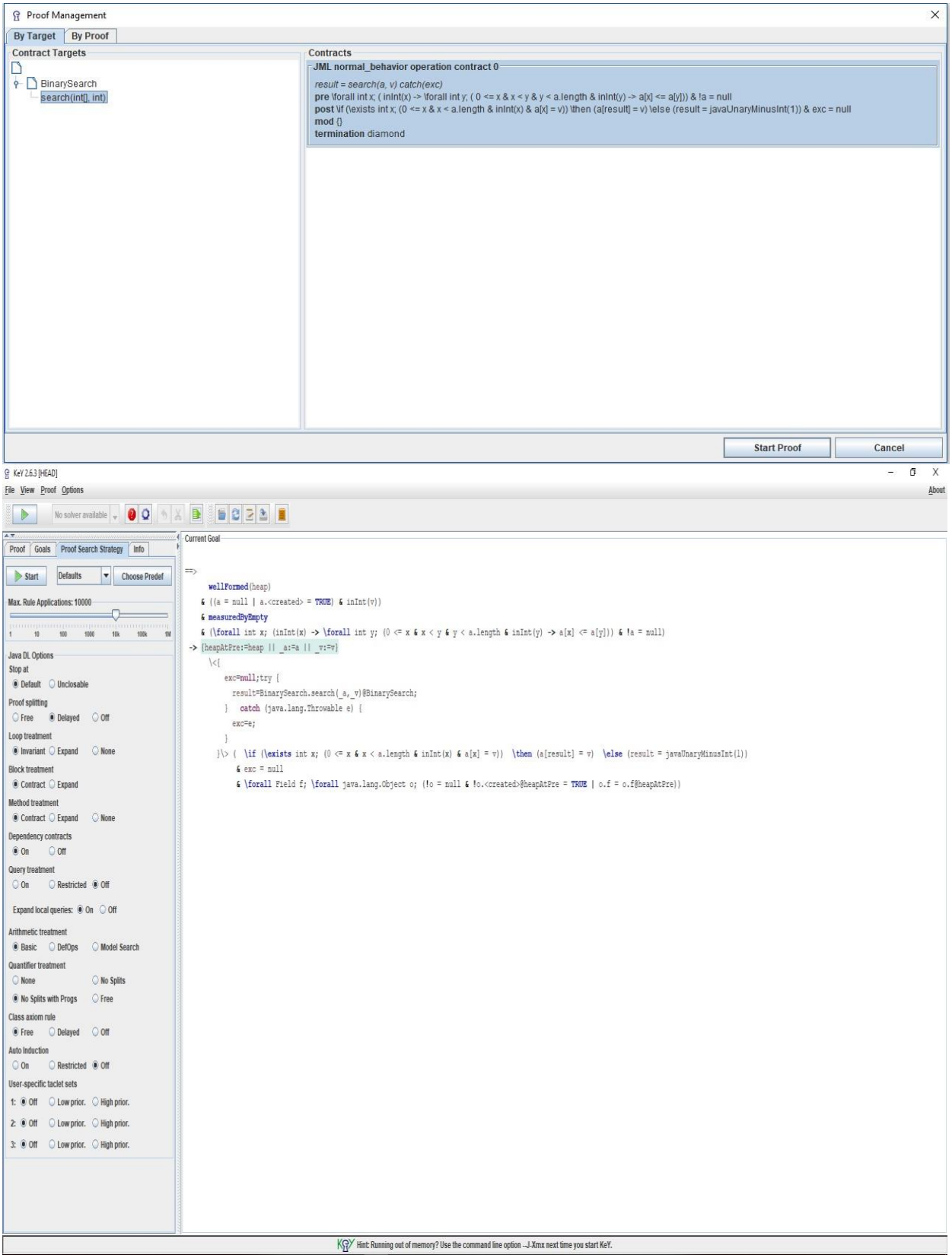


Figure 19: KeY IDE

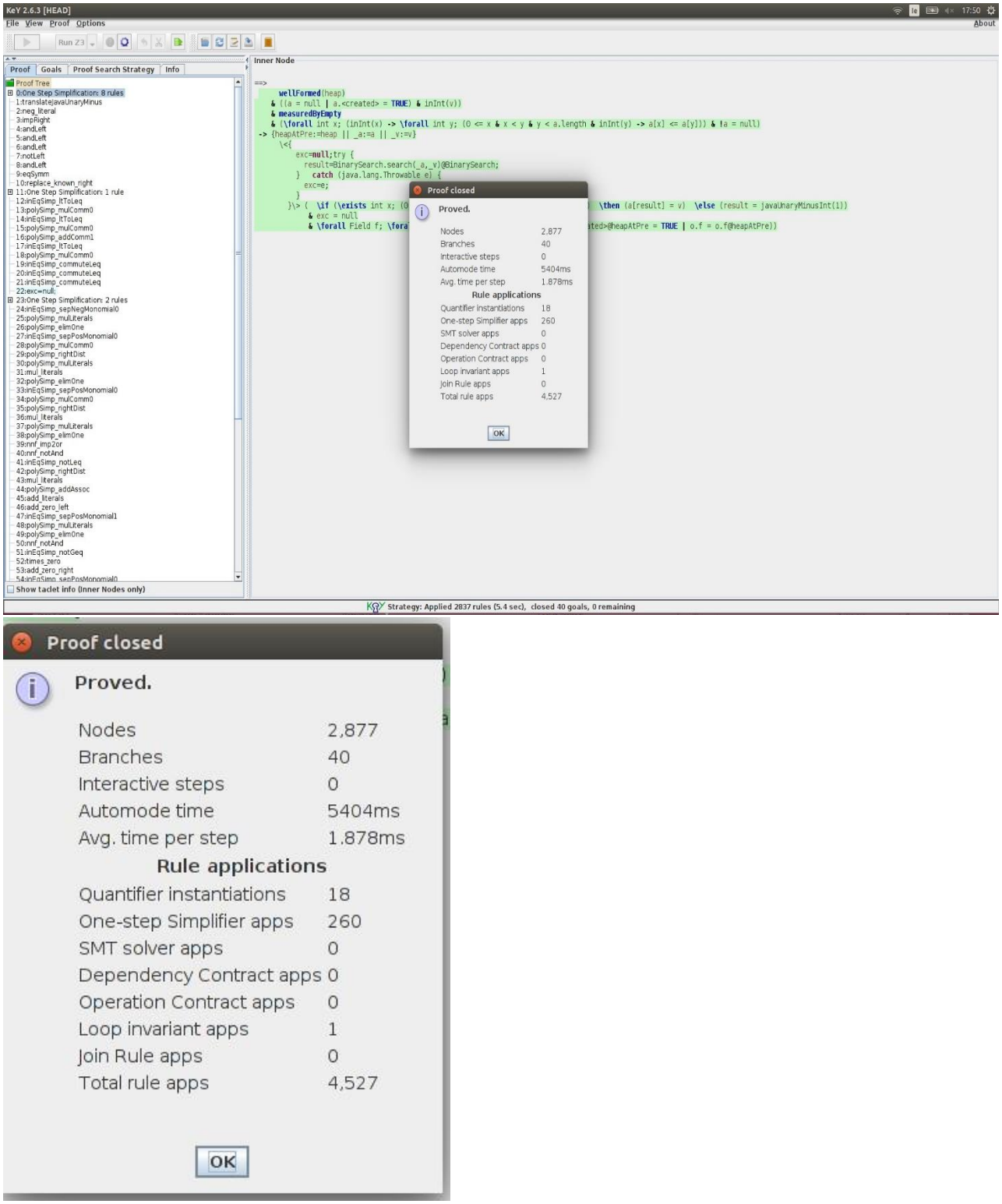


Figure 20: KeY IDE - Binary Search - Proof

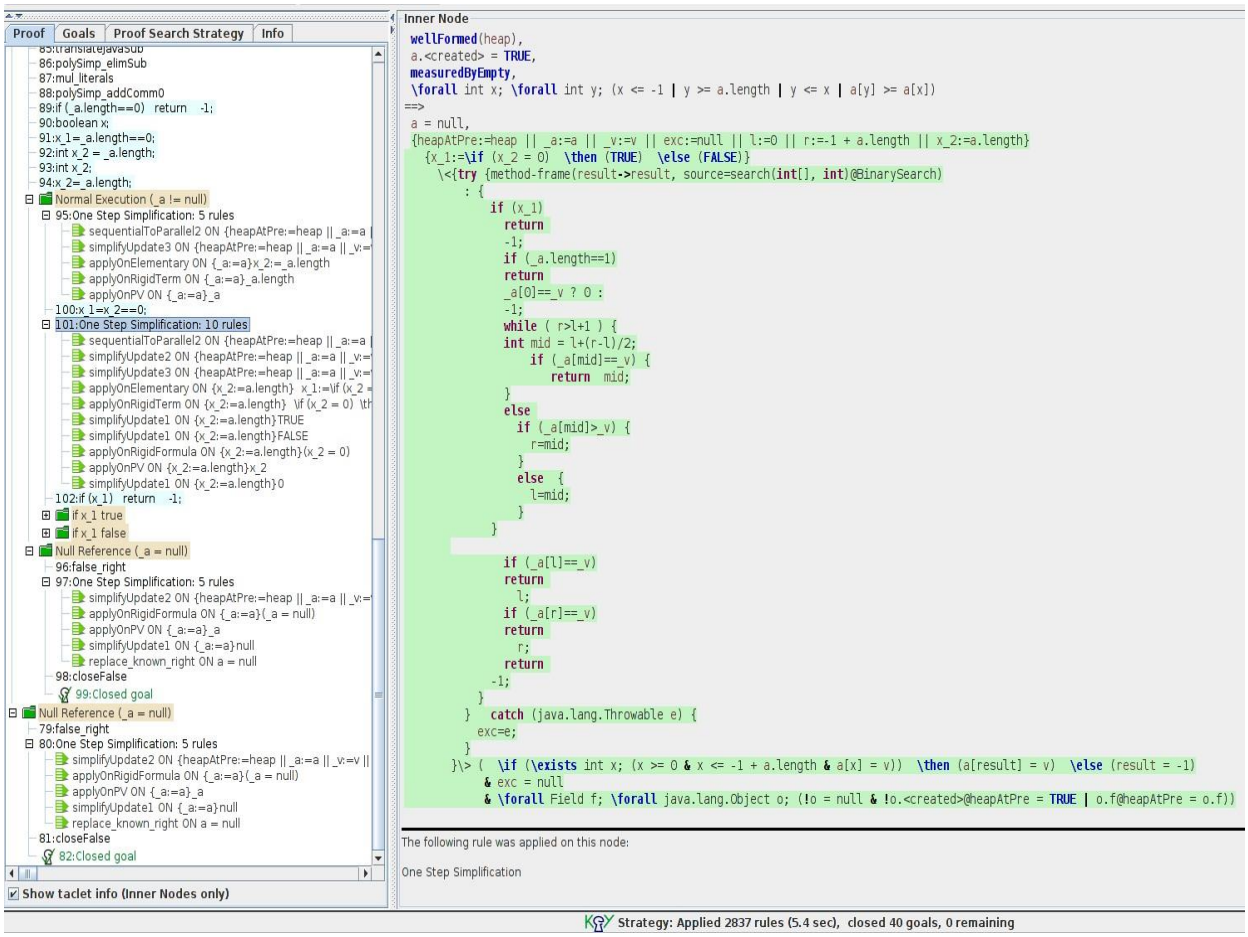


Figure 21: KeY IDE- Binary Search - Rules

- OpenJML

BinarySearch.java

```java
package Programs;

public class BinarySearch {
    //@ 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;
    public static int BinarySearch(int[] arr, int key) {
        if (arr.length == 0) {
            return -1;
        } else {
            int low = 0;
            int high = arr.length;
            int mid = low + (high - low) / 2;
            //@ 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;
            while (low < high && arr[mid] != key) {
                if (arr[mid] < key) {
                    low = mid + 1;
                } else {
                    high = mid;
                }
                mid = low + (high - low) / 2;
            }
            if (low >= high) {
                return -1;
            }
            return mid;
        }
    }
}
```

Figure 22: OpenJML - Binary Search

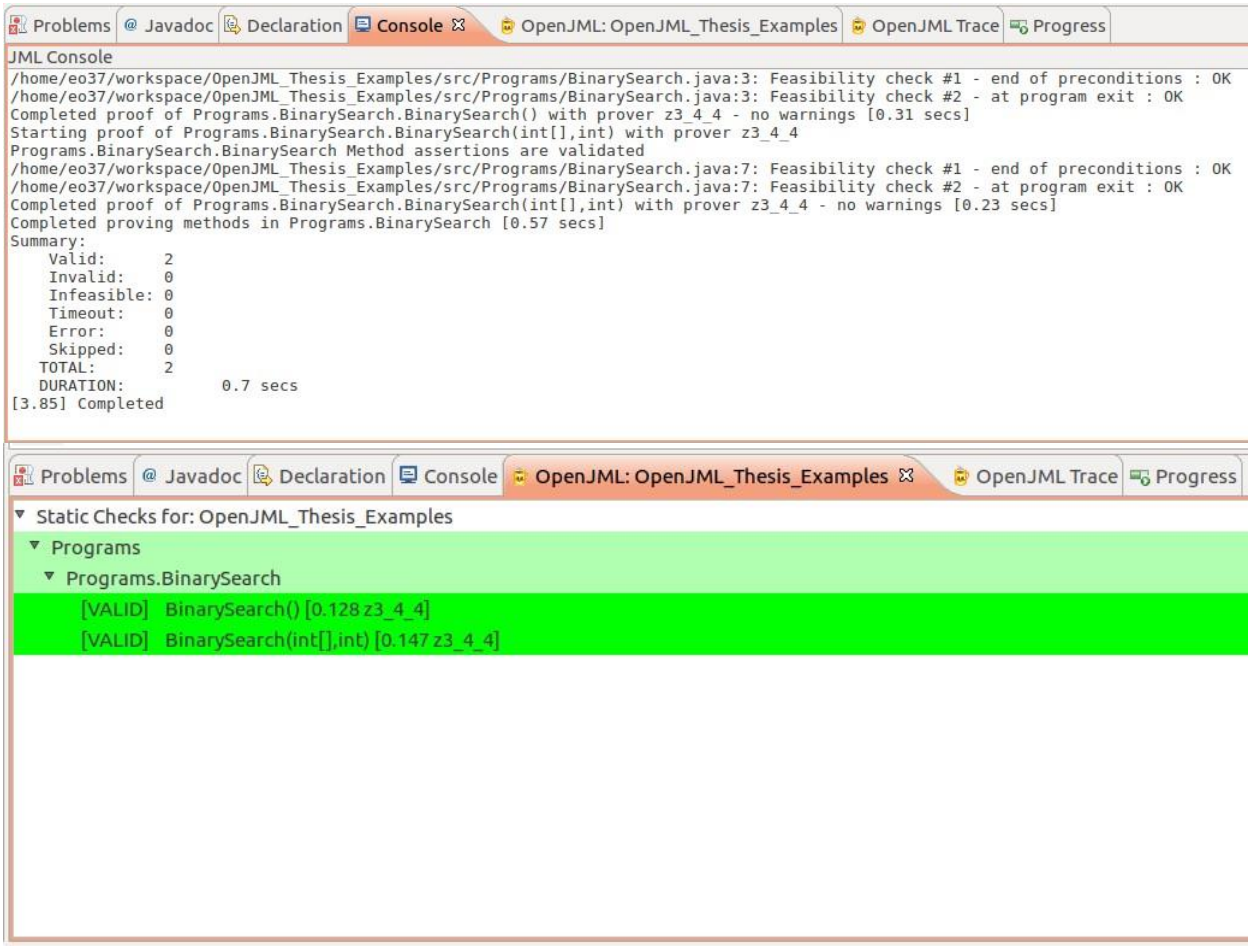


Figure 23: OpenJML - Eclipse - Valid Verification

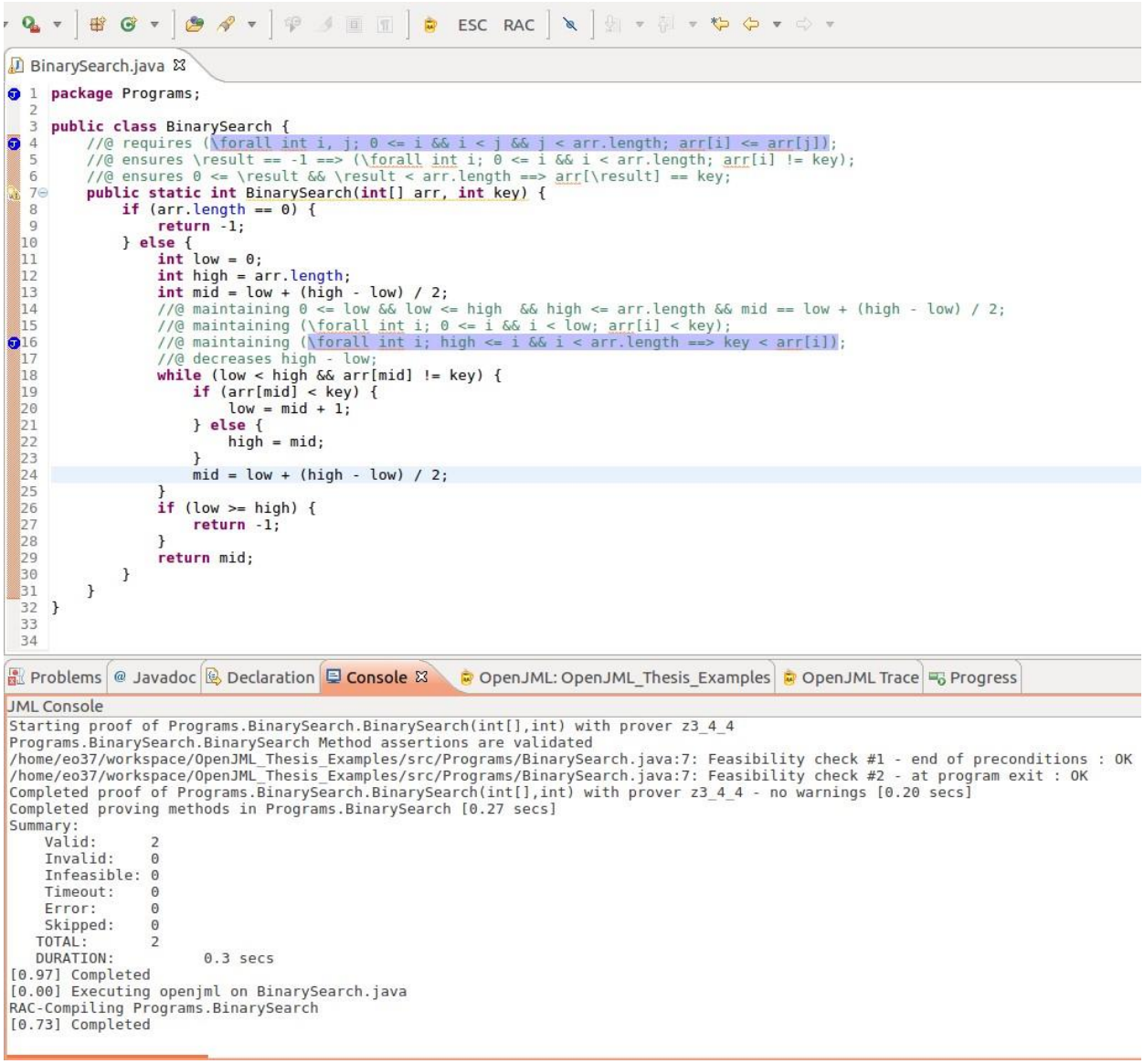


Figure 24: OpenJML - Eclipse - RAC

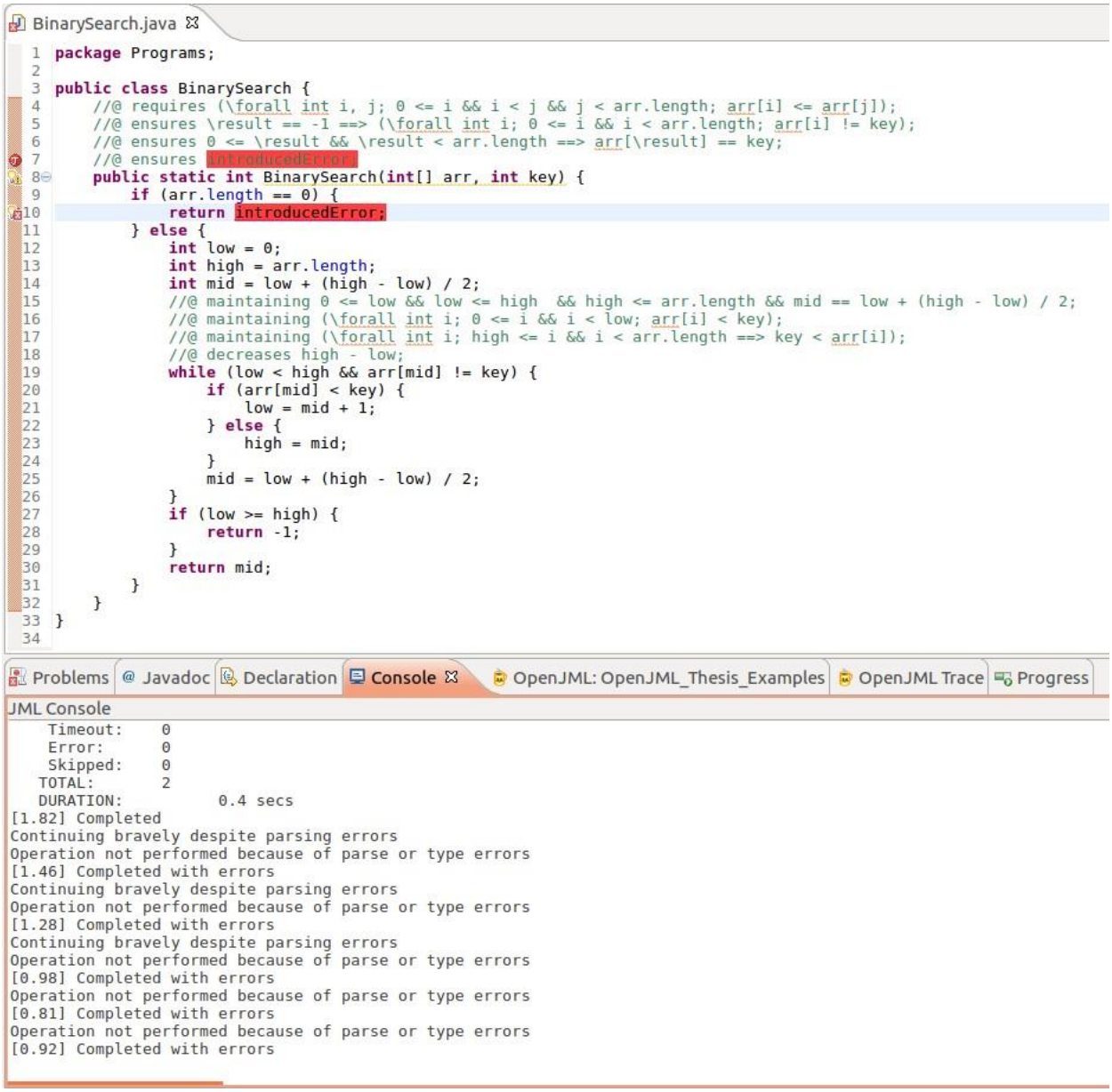


Figure 25: OpenJML - Eclipse - TypeCheck

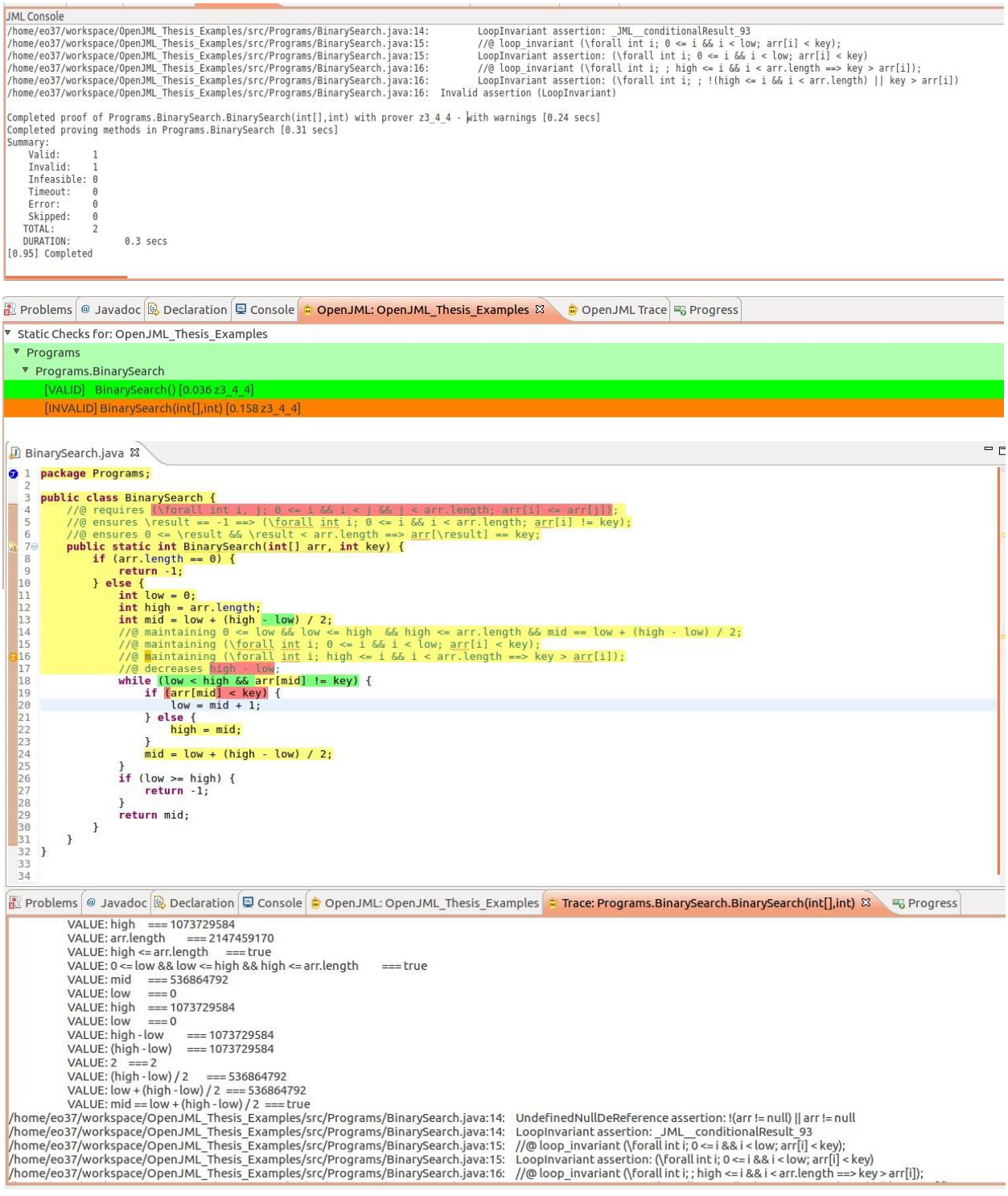


Figure 26: OpenJML - Eclipse - ESC Error

Case Study 2

PrefixSum

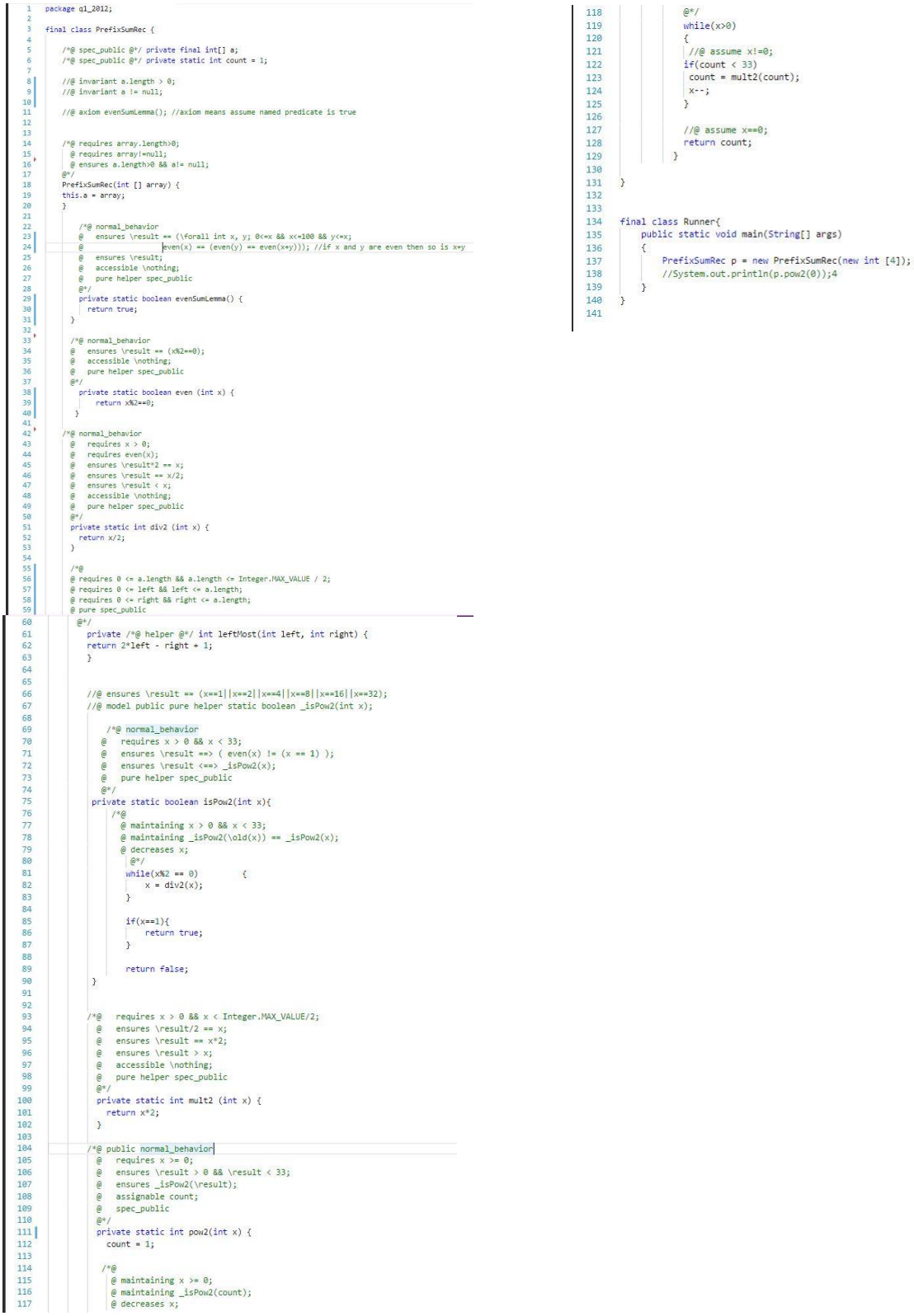
- KeY Implementation

PrefixSumRec.java

```
final class PrefixSumRec {

    private final int[] a;

    //@ invariant a.length > 0;
    //@ invariant isPow2(a.length);
    //@ accessible \inv: \singleton(a);

    //@ axiom evenSumLemma();

    PrefixSumRec(int [] a) {
    this.a = a;
    }

    /*@ 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; }

    /*@ 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;
      @ strictly_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);
    }

    /*@ normal_behavior
      @   requires x >= 0;
      @   ensures \result == (\product int i; 0 <= i && i < x; 2);
      @   ensures \result > x;
      @   accessible \nothing;
      @   measured_by x;
      @ strictly_pure helper
      @*/
    private static int pow2( int x ) {
      return x==0? 1: 2*pow2(x-1);
    }

    /*@ normal_behavior
      @   requires x > 0;
      @   requires even(x);
      @   ensures \result*2 == x;
      @   ensures \result == x/2;
      @   ensures \result < x;
      @   accessible \nothing;
      @ strictly_pure helper
      @*/
    private static int div2 (int x) {
        return x/2;
    }

    /*@ 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;
    }

    //@ strictly_pure
    private /*@ helper @*/ static int leftMost(int left, int right) {
    return 2*left - right + 1;
    }

    /*@ normal_behavior //@\label{lst:min-begin}@
      @   requires k >= 0;
      @   ensures 0 <= \result && \result <= k;
      @   ensures pow2(\result) <= k+1;
      @   ensures k% pow2(\result+1) == pow2(\result)-1;
      @   ensures (\forall int z; k% pow2(z+1) == pow2(z)-1;
      @                z >= \result);
      @   accessible \nothing;
      @ strictly_pure helper
      @*/
    private static int min ( int k ) {
        int n = 0;
        /*@ maintaining (\forall int z; 0 <= z && z < n;
          @                k% pow2(z+1) != pow2(z)-1 );
          @ maintaining 0 <= n && pow2(n) <= k+1;
          @ decreasing k-n+1;
          @ assignable \strictly_nothing;
          @*/
        while ( k% pow2(n+1) != pow2(n)-1 ) n++;
        return n;
    }//@\label{lst:min-end}@

    /*@ normal_behavior //@\label{lst:eff-begin}@
      @   requires 0 <= k;
      @   ensures \result == pow2(min(k));
      @   ensures 0 < \result && \result <= k+1;
      @   measured_by k + 2;
      @   accessible \nothing;
      @*/
    private /*@ helper strictly_pure @*/ static int f ( int k ) {
        return even(k)? 1: f(div2(k-1));
    }//@\label{lst:eff-end}@


    /*@ public normal_behavior
      @   requires right > left;
      @   requires leftMost(left, right) >= 0;
      @   requires right < a.length;
      @   requires isPow2(right-left);
      @   requires !even(right);
      @   requires !even(left) || right-left==1;
      @   ensures (\forall int k; 0 <= k && k < 2*(right-left);
      @            a[k+leftMost(left,right)] ==
      @                    (\sum int i; k-f(k)+1 <= i && i < k+1; \old(a[i+leftMost(left,right)])));
      @   //ensures a[right] == (\sum int i; leftMost(left,right) <= i && i < right+1; \old(a[i]));
      @   measured_by right - left + a.length + 3;
      @   assignable \infinite_union(int k; leftMost(left,right) <= k
      @             && k <= right && !even(k); \singleton(a[k]));
      @*/
    public void upsweep(int left, int right) {
        int space = right - left;
        if (space > 1) {
            upsweep(left-div2(space), left);
            upsweep(right-div2(space), right);
        }
        a[right] = a[left] + a[right];
    }

    private static int binWeight (int i) {
        if (i==0) return 0;
        if (even(i)) return binWeight(div2(i));
        return 1 + binWeight(div2(i-1));
    }

    /*@ normal_behavior
      @   requires right > left;
      @   requires leftMost(left, right) >= 0;
      @   requires right < a.length;
      @   requires isPow2(right-left);
      @   requires !even(right);
      @   requires !even(left) || right-left==1;
      @// ensures (\forall int k; leftMost(left,right) <= k && k <= right;
      @//           a[k] == (\sum int i; 0 <= i && i < binWeight(k-leftMost(left,right));
      @//                                   \old(a[i+leftMost+xxx])) + \old(a[right]));
      @   measured_by right - left + a.length + 3;
      @   assignable \infinite_union(int k; leftMost(left,right) <= k
      @                         && k <= right; \singleton(a[k]));
      @*/
    public void downsweep(int left, int right) {
        int tmp = a[right];
        a[right] = a[right] + a[left];
        a[left] = tmp;
        int space = right - left;
        if (space > 1) {
            downsweep(left-div2(space),left);
            downsweep(right-div2(space),right);
        }
    }

    /*@ normal_behavior
      @   requires \invariant_for(p) && p.a.length > 1;
      @   ensures (\forall int i; 0 <= i && i < p.a.length;
      @            p.a[i] == (\sum int j; 0 <= j && j < i;
      @                       \old(p.a[i])));
      @*/
    public static void main( PrefixSumRec p ) {
        final int l = div2(p.a.length)-1;
        final int r = p.a.length-1;
        p.upsweep(l, r);
        p.a[r] = 0;
        p.downsweep(l, r);
    }
}
```

Figure 27: KeY - PrefixSum

- OpenJML



```
package q1_2012;

final class PrefixSumRec {

    /*@ spec_public @*/ private final int[] a;
    /*@ spec_public @*/ private static int count = 1;

    //@ invariant a.length > 0;
    //@ invariant a != null;

    //@ axiom evenSumLemma(); //axiom means assume named predicate is true


    /*@ requires array.length>0;
      @ requires array!=null;
      @ ensures a.length>0 && a!= null;
    @*/
    PrefixSumRec(int [] array) {
    this.a = array;
    }

        /*@ normal_behavior
        @   ensures \result == (\forall int x, y; 0<=x && x<=100 && y<=x;
        @                       even(x) == (even(y) == even(x+y))); //if x and y are even then so is x+y
        @   ensures \result;
        @   accessible \nothing;
        @   pure helper spec_public
        @*/
        private static boolean evenSumLemma() {
          return true;
       }

       /*@ normal_behavior
       @   ensures \result == (x%2==0);
       @   accessible \nothing;
       @   pure helper spec_public
       @*/
         private static boolean even (int x) {
            return x%2==0;
        }

    /*@ 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 / 2;
      @ 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;
         }


         //@ 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;
          }


         /*@   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;
          }

         /*@ 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;
           }

    }


    final class Runner{
        public static void main(String[] args)
        {
            PrefixSumRec p = new PrefixSumRec(new int [4]);
            //System.out.println(p.pow2(0));4
        }
    }
```

Figure 28: OpenJML - PrefixSum

- Krakatoa

```
//@+ CheckArithOverflow = no


/*@ predicate is_Even{L}(integer x) =
  @ x%2 == 0;
@*/

/*@ predicate is_Pow2{L}(integer x) =
  @ (x==1||x==2||x==4||x==8||x==16||x==32);
@*/


public class PrefixSum {
  int a [];

  public PrefixSum(int [] array) {
  this.a = array;
    }

      /*@
      @   assigns \nothing;
      @   behavior success:
      @     ensures (\forall int x y; 0<=x && x<=100 && y<=x ==> is_Even(x)
      @                                                     ==>(is_Even(y)
      @                                                     ==> is_Even(x+y))
      @                                                     ==> \result);
      @*/
    private static boolean evenSumLemma() { return true; }


  /*@
    @ requires 0 <= a.length && a.length <= 1000
      &&  0 <= left && left <= a.length
      &&  0 <= right && right <= a.length;
    @ assigns \nothing;
    @ ensures \result == (2*left - right+1);
    @*/
      private int leftMost(int left, int right) {
    return 2*left - right + 1;
      }


  /*@
    @   requires x > 0 && is_Even(x);
    @   assigns \nothing;
    @   behavior success:
    @     ensures \result == x/2 && \result*2 == x && \result < x;
    @*/
    private static int div2 (int x) {
      return x/2;
    }
```

```
  /*@
    @   requires x >= 0;
    @   assigns \nothing;
    @   behavior success:
    @     ensures (\result > 0  <==> is_Pow2(\result));
    @*/
    private static int pow2(int x) {
    int count = 1;

    /*@
      @ loop_invariant x >= 0 && count >= 1 && is_Pow2(\old(count) <==> is_Pow2(count));
      @ loop_variant x-1;
    @*/
    while(x > 0){
      count += count;
      x--;
    }
    return count;
    }



  /*@
    @   requires x > 0 && x < 33;
    @   assigns \nothing;
    @   behavior success:
    @     ensures \result <==> is_Pow2(x);
    @*/
  private static boolean isPow2(int x){

    /*@ loop_invariant x >= 1 && x < 33;
      //&& is_Pow2(\old(x)) <==> is_Pow2(x);
      @ loop_variant x;
      @*/
    while(x%2 == 0)
    {
      x = div2(x);
    }

    if(x==1){
      return true;
      }

    return false;
  }


}
```

Figure 29: Krakatoa - PrefixSum

Case Study 3

Longest Repeated Substring

- KeY Implementation

```
final class Lemmas {

    /*@ public normal_behaviour
      @  requires sa.a != null;
      @  requires 0 <= a && a < sa.a.length;
      @  requires 0 <= b && b < sa.a.length;
      @  requires 0 <= c && c < sa.a.length;
      @  requires sa.compare(a,b) > 0;
      @  requires sa.compare(b,c) > 0;
      @  ensures sa.compare(a,c) > 0;
      @  strictly_pure helper
      @*/
    public static boolean compareTrans(SuffixArray sa, int a, int b, int c) {
    return true;
    }

    /*@ public normal_behaviour
      @  requires sa.a != null;
      @  requires 0 <= a && a <= sa.a.length - k;
      @  requires 0 <= b && b < sa.a.length;
      @  requires 0 <= c && c <= sa.a.length - k;
      @  requires sa.compare(a,b) >= 0;
      @  requires sa.compare(b,c) > 0;
      @  requires (\forall int t; 0 <= t && t < k; sa.a[a+t] == sa.a[c+t]);
      @  ensures  (\forall int t; 0 <= t && t < k; sa.a[a+t] == sa.a[b+t]);
      @  ensures  (\forall int t; 0 <= t && t < k; sa.a[b+t] == sa.a[c+t]);
      @  ensures  b < sa.a.length - k;
      @  strictly_pure helper
      @*/
    public static boolean compareRun(SuffixArray sa,
                                     int a, int b, int c, int k){
    return true;
    }

    /*@ public normal_behaviour
      @  requires \invariant_for(sa);
      @  requires 0 <= i && i < j && j < sa.a.length;
      @  ensures sa.compare(sa.suffixes[j], sa.suffixes[i]) > 0;
      @  strictly_pure helper
      @*/
    public static boolean compareSuffixArray(SuffixArray sa, int i, int j) {

    /*@ loop_invariant
      @  sa.compare(sa.suffixes[m], sa.suffixes[i]) > 0 &&
      @  i+1 <= m && m <= j;
      @ decreases j - m;
      @ assignable \strictly_nothing;
      @*/
    for(int m = i + 1; m < j; m++) {
        compareTrans(sa, sa.suffixes[m+1], sa.suffixes[m], sa.suffixes[i]);
    }
    return true;
    }

    /*@ public normal_behaviour
      @  requires sa.a != null;
      @  requires 0 <= i && i < sa.a.length;
      @  ensures sa.compare(i,i) == 0;
      @  strictly_pure helper
      @*/
    public static boolean compareReflex(SuffixArray sa, int i) {
      return true;
    }


    /*@ public normal_behaviour
      @  requires \invariant_for(sa);
      @  requires 0 <= i && i < j && j < sa.a.length;
      @  requires
      @     sa.suffixes[i] + k <= sa.a.length &&  sa.suffixes[j] + k <= sa.a.length &&
      @     (\forall int t; 0 <=t && t < k; sa.a[sa.suffixes[i]+t] == sa.a[sa.suffixes[j]+t]);
      @  ensures
      @     sa.suffixes[i+1] + k <= sa.a.length &&
      @     (\forall int t; 0 <=t && t < k; sa.a[sa.suffixes[i]+t] == sa.a[sa.suffixes[i+1]+t]);
      @  ensures \result;
      @  strictly_pure helper
      @*/
    public static boolean neighbourMax(SuffixArray sa, int i, int j, int k) {

    compareSuffixArray(sa, i, j);
        if(j == i+1) {
            compareReflex(sa, sa.suffixes[i+1]);
        } else {
            compareSuffixArray(sa, i+1,j);
        }
    compareRun(sa, sa.suffixes[j], sa.suffixes[i+1], sa.suffixes[i], k);
        return true;
    }
}

```

Figure 30: KeY- Longest Repeated Substring - Lemmas

```
/** Contains an implementation for the Longest Common Prefix algorithm.
  * <em>FM 2012 Verification Competition, Problem 1 (part a).</em><br>
  * Longest Common Prefix (LCP) is an algorithm used for text querying. In
  * the following, we model text as an integer array. <ul>
  * <li> Input:  an integer array a, and two indices x and y into this array
  * <li> Output: length of the longest common prefix of the subarrays of a
  *     starting at x and y respectively.</ul>
  * @author bruns, woj
  */
final class LCP {


  /*@ normal_behavior
    @ requires 0 <= x && x < a.length;
    @ requires 0 <= y && y < a.length;
    @ requires x != y;
    @ ensures 0 <= \result;
    @ ensures \result <= a.length-x && \result <= a.length-y;
    @ ensures (\forall int i; 0 <= i && i < \result;//ß\label{lst:lcp-post-ex1}ß
    @                         a[x+i] == a[y+i] );//ß\label{lst:lcp-post-ex2}ß
    @ ensures a[x+\result] != a[y+\result]//ß\label{lst:lcp-post-max-start}ß
    @             || \result == a.length-x
    @             || \result == a.length-y;//ß\label{lst:lcp-post-max-end}ß
    @ strictly_pure @*/
    static int lcp(int[] a, int x, int y) {
        int l = 0;
        /*@ maintaining 0 <= l && l+x <= a.length//ß\label{lst:lcp-inv1-start}ß
          @           && l+y <= a.length && x!=y;//ß\label{lst:lcp-inv1-end}ß
          @ maintaining (\forall int z; 0 <= z && z < l;//ß\label{lst:lcp-inv2-start}ß
          @                           a[x+z] == a[y+z] );//ß\label{lst:lcp-inv2-end}ß
          @ decreasing a.length-l; //ß\label{lst:lcp-variant}ß
          @ assignable \strictly_nothing; //ß\label{lst:lcp-loopframe}ß
          @*/
        while (x+l<a.length && y+l<a.length
                && a[x+l]==a[y+l]) l++;
        return l;
    }
}
```

Figure 31: KeY- Longest Repeated Substring - LCP

```
/** Implementation of the Longest Repeated Substring algorithm.
  * <em>FM 2012 Verification Competition, Problem 1 (part b).</em><br>
  * Together with a suffix array, LCP can be used to solve interesting text
  * problems, such as finding the longest repeated substring (LRS) in a text.<br>
  * A suffix array (for a given text) is an array of all suffixes of the
  * text. For the text [7,8,8,6], the suffix array is
  *   [[7,8,8,6],
  *      [8,8,6],
  *        [8,6],
  *          [6]]
  * <p>Typically, the suffixes are not stored explicitly as above but
  * represented as pointers into the original text. The suffixes in a suffix
  * array  are sorted in lexicographical order. This way, occurrences of
  * repeated substrings in the original text are neighbors in the suffix
  * array.</p>
  *
  * For the above, example (assuming pointers are 0-based integers), the
  * sorted suffix array is: [3,0,2,1]
  */
public class LRS {


    private int s = 0;
    private int t = 0;
    private int l = 0;
    private final SuffixArray sa;

    LRS (SuffixArray arr) { sa = arr; }


    /*@ normal_behavior
      @ requires \invariant_for(sa);
      @ requires sa.a.length >= 2;
      @ ensures 0 <= s && s < sa.a.length;
      @ ensures 0 <= t && t < sa.a.length;
      @ ensures 0 <= l && l < sa.a.length;
      @ ensures s+l <= sa.a.length && t+l <= sa.a.length;
      @ ensures (\forall int j; 0 <= j && j < l; sa.a[s+j] == sa.a[t+j]);
      @ ensures s != t || l == 0;
      @ ensures !(\exists int i,k; 0 <= i && i < k && k < sa.a.length-l;
      @            (\forall int j; 0 <= j && j <= l; sa.a[k+j] == sa.a[i+j]));
      @         // there is no LRS of length l+1
      @*/
    public void doLRS() {
        int s = 0; int t = 0; int l = 0;

        /*@ maintaining sa != null && \invariant_for(sa);
          @ maintaining 0 <= s && s < sa.a.length;
          @ maintaining 0 <= t && t < sa.a.length;
          @ maintaining 0 <= l && l < sa.a.length;
          @ maintaining s+l <= sa.a.length && t+l <= sa.a.length;
          @ maintaining s != t || l == 0;
          @ maintaining 0 < x && x <= sa.a.length;
          @ maintaining (\forall int j; 0 <= j && j <l; sa.a[s+j] == sa.a[t+j]);
          @ maintaining !(\exists int w; 0 < w && w < x
          @               && sa.suffixes[w-1] < sa.a.length-l
          @               && sa.suffixes[w]   < sa.a.length-l;
          @               (\forall int j; 0 <= j && j <= l;
          @                sa.a[sa.suffixes[w-1]+j] == sa.a[sa.suffixes[w]+j]));
          @ decreasing sa.a.length-x;
          @ assignable \strictly_nothing;
          @*/
        for (int x=1; x < sa.a.length; x++) {
            int length = LCP.lcp(sa.a,sa.suffixes[x],
                                 sa.suffixes[x-1]);
            if (length > l) {
                s = sa.suffixes[x];
                t = sa.suffixes[x-1];
                l = length;
        }   }
        this.s = s; this.t = t; this.l = l;
    }

}


//Based on code by Robert Sedgewick and Kevin Wayne.

```

Figure 32: KeY- Longest Repeated Substring - LRS

```
public final class SuffixArray {

    final /*@ spec_public @*/ int[] a;
    final /*@ spec_public @*/ int[] suffixes;

    /*@ public invariant
      @           (\forall int i; 0 <= i && i < a.length;
      @           (\exists int j; 0 <= j && j < a.length; suffixes[j]==i));
      @           // suffixes is a permutation on indices
      @ public invariant
      @         (\forall int i; 0 <= i && i < a.length;
      @                 0 <= suffixes[i] && suffixes[i] < a.length);
      @           // indices are in range (follows from above, cannot hurt)
      @ public invariant (\forall int i; 0 < i && i < a.length;
      @                           suffixes[i-1] != suffixes[i]);
      @           // indices are unique (follows from above, cannot hurt)
      @ public invariant (\forall int i; 0 < i && i < a.length;
      @                           compare(suffixes[i],suffixes[i-1]) > 0);
      @           // suffixes is ordered lexicographically
      @ public invariant a.length == suffixes.length;
      @*/

    /*@ normal_behavior
      @ ensures this.a == a;
      @*/
    public /*@ pure @*/ SuffixArray(int[] a) {
        this.a = a;
        suffixes = new int[a.length];
        /*@ maintaining 0 <= i && i <= a.length;
          @ maintaining (\forall int j; 0 <= j && j < i; suffixes[j] == j);
          @ decreasing a.length-i;
          @ assignable suffixes[*];
          @*/
        for (int i = 0; i < a.length; i++) suffixes[i] = i;
        sort(suffixes);
    }

    /*@ normal_behavior
      @ requires a != null;
      @ requires 0 <= x && x < a.length;
      @ requires 0 <= y && y < a.length;
      @ ensures \result <  0 ==>
      @           (\exists int j; 0 <= j && j < a.length-y;
      @               ((j < a.length-x && a[x+j] < a[y+j] ) || j == a.length-x)
      @               && (\forall int k; 0 <= k && k < j; a[x+k] == a[y+k]));
      @ ensures \result == 0 ==> x == y;
      @ ensures \result >  0 ==>
      @           (\exists int j; 0 <= j && j < a.length-x;
      @               ((j < a.length-y && a[x+j] > a[y+j] ) || j == a.length-y)
      @               && (\forall int k; 0 <= k && k < j; a[x+k] == a[y+k]));
      @ ensures \result == -compare(y,x);
      @ accessible a, a[*];
      @ spec_public strictly_pure helper
      @*/
    private int compare(int x, int y) {
        if (x == y) return 0;
        int l = LCP.lcp(a,x,y);
        if (x+l == a.length) return -1;
        if (y+l == a.length) return 1;
        if (a[x+l] < a[y+l]) return -1;
        if (a[x+l] > a[y+l]) return 1;

        throw new RuntimeException();
    }



    private void /*@ helper @*/ sort(final int[] data) {
        /*@ maintaining data.length == a.length;
          @ maintaining 0 <= k && k <= data.length;
          @ maintaining (\forall int i; 0 <= i && i < a.length;
          @             (\exists int j; 0 <= j && j < a.length; data[j]==i));
          @ maintaining (\forall int i; 0 < i && i < a.length;
          @                         i < k? compare(data[i],data[i-1]) > 0
          @                              : data[i] == \old(data[i]));
          @ decreasing data.length - k;
          @ assignable data[*];
          @*/
        for(int k = 0; k < data.length; k++)
            /*@ maintaining 0 <= l && l <= k;
              @ maintaining (\forall int i; l < i && i <= k;
              @                 compare(data[i],data[i-1]) > 0);
              @ maintaining (\forall int i; 0 < i && i < data.length
              @                 && !( l < i && i <= k);
              @                 data[i] == \old(data[i]));
              @ decreasing l;
              @ assignable data[*];
              @*/
            for(int l = k; l > 0 && compare(data[l - 1], data[l]) > 0; l--)
                swap(data, l);
    }

    /*@ normal_behavior
      @ requires 0 < x && x < data.length;
      @ ensures data[x]   == \old(data[x-1]);
      @ ensures data[x-1] == \old(data[x]);
      @ assignable data[x], data[x-1];
      @*/
    private static void /*@ helper @*/ swap(int[] data, int x) {
        final int y = x-1;
        final int t = data[x];
        data[x] = data[y];
        data[y] = t;
    }

}


//Based on code by Robert Sedgewick and Kevin Wayne.
```

Figure 33: KeY- Longest Repeated Substring - SuffixArray

- OpenJML implementation

```
1  package q2_2012;
2
3  /** Contains an implementation for the Longest Common Prefix algorithm.
4   * <em>FM 2012 Verification Competition, Problem 1 (part a).</em><br>
5   * Longest Common Prefix (LCP) is an algorithm used for text querying. In
6   * the following, we model text as an integer array. <ul>
7   * <li> Input:  an integer array a, and two indices x and y into this array
8   * <li> Output: length of the longest common prefix of the subarrays of a
9   *     starting at x and y respectively.</ul>
10  * @author bruns, woj
11  */
12 final class LCP {
13
14
15  /*@ normal_behavior
16    @ requires 0 <= x && x < a.length;
17    @ requires 0 <= y && y < a.length;
18    @ requires x != y;
19    @ ensures 0 <= \result;
20    @ ensures \result <= a.length-x && \result <= a.length-y;
21    @ //ensures (\forall int i; 0 <= i && i < \result; a[x+i] == a[y+i] );
22    @ //ensures (\forall int i; x <= i && i < x+\result; a[i] == a[y+i-x]);
23    @ //ensures a[x+\result] != a[y+\result] || \result == a.length-x || \result == a.length-y;
24    @ pure;
25    @*/
26    static int lcp(int[] a, int x, int y) {
27        int l = 0;
28        /*@ loop_modifies l;
29          @ maintaining 0 <= l && l+x <= a.length && l+y <= a.length && x!=y;
30          @// maintaining (\forall int z; 0 <= z && z < l; a[x+z] == a[y+z] );
31          @ decreasing a.length-l;
32          @*/
33        while (x+l < a.length &&
34               y+l < a.length &&
35               a[x+l]==a[y+l]){
36          l++;
37        }
38
39        return l;
40    }
41 }
```

Figure 34: OpenJML - Longest Repeated Substring - LCP

```
package q2_2012;

final class Lemmas {

    /*@ public normal_behaviour
      @  requires sa.a != null;
      @  requires 0 <= a && a < sa.a.length;
      @  requires 0 <= b && b < sa.a.length;
      @  requires 0 <= c && c < sa.a.length;
      @  requires sa.compare(a,b) > 0;
      @  requires sa.compare(b,c) > 0;
      @  ensures sa.compare(a,c) > 0;
      @  pure helper
      @*/
    public static boolean compareTrans(SuffixArray sa, int a, int b, int c) {
    return true;
    }

    /*@ public normal_behaviour
      @  requires sa.a != null;
      @  requires 0 <= a && a <= sa.a.length - k;
      @  requires 0 <= b && b < sa.a.length;
      @  requires 0 <= c && c <= sa.a.length - k;
      @  requires sa.compare(a,b) >= 0;
      @  requires sa.compare(b,c) > 0;
      @  requires (\forall int t; 0 <= t && t < k; sa.a[a+t] == sa.a[c+t]);
      @  ensures  (\forall int t; 0 <= t && t < k; sa.a[a+t] == sa.a[b+t]);
      @  ensures  (\forall int t; 0 <= t && t < k; sa.a[b+t] == sa.a[c+t]);
      @  ensures  b < sa.a.length - k;
      @  pure helper
      @*/
    public static boolean compareRun(SuffixArray sa,
                                     int a, int b, int c, int k){
    return true;
    }

    /*@ public normal_behaviour
      @  requires \invariant_for(sa);
      @  requires 0 <= i && i < j && j < sa.a.length;
      @  ensures sa.compare(sa.suffixes[j], sa.suffixes[i]) > 0;
      @  pure helper
      @*/
    public static boolean compareSuffixArray(SuffixArray sa, int i, int j) {

    /*
      @ decreases j - m;
      @ assignable \nothing;
      @ loop_invariant sa.compare(sa.suffixes[m], sa.suffixes[i]) > 0 && i+1 <= m && m <= j;
      @*/
    for(int m = i + 1; m < j; m++) {
        compareTrans(sa, sa.suffixes[m+1], sa.suffixes[m], sa.suffixes[i]);
    }
    return true;
    }

    /*@ public normal_behaviour
      @  requires sa.a != null;
      @  requires 0 <= i && i < sa.a.length;
      @  ensures sa.compare(i,i) == 0;
      @  pure helper
      @*/
    public static boolean compareReflex(SuffixArray sa, int i) {
       return true;
    }


    /*@ public normal_behaviour
      @  requires \invariant_for(sa);
      @  requires 0 <= i && i < j && j < sa.a.length;
      @  requires
      @     sa.suffixes[i] + k <= sa.a.length &&  sa.suffixes[j] + k <= sa.a.length &&
      @     (\forall int t; 0 <=t && t < k; sa.a[sa.suffixes[i]+t] == sa.a[sa.suffixes[j]+t]);
      @  ensures
      @     sa.suffixes[i+1] + k <= sa.a.length &&
      @     (\forall int t; 0 <=t && t < k; sa.a[sa.suffixes[i]+t] == sa.a[sa.suffixes[i+1]+t]);
      @  ensures \result;
      @  pure helper
      @*/
    public static boolean neighbourMax(SuffixArray sa, int i, int j, int k) {

    compareSuffixArray(sa, i, j);
        if(j == i+1) {
            compareReflex(sa, sa.suffixes[i+1]);
        } else {
            compareSuffixArray(sa, i+1,j);
        }
    compareRun(sa, sa.suffixes[j], sa.suffixes[i+1], sa.suffixes[i], k);
        return true;
    }
}

```

Figure 35: OpenJML - Longest Repeated Substring - Lemmas

```
1  package q2_2012;
2
3  /** Implementation of the Longest Repeated Substring algorithm.
4   * <em>FM 2012 Verification Competition, Problem 1 (part b).</em><br>
5   * Together with a suffix array, LCP can be used to solve interesting text
6   * problems, such as finding the longest repeated substring (LRS) in a text.<br>
7   * A suffix array (for a given text) is an array of all suffixes of the
8   * text. For the text [7,8,8,6], the suffix array is
9   *   [[7,8,8,6],
10  *      [8,8,6],
11  *        [8,6],
12  *          [6]]
13  * <p>Typically, the suffixes are not stored explicitly as above but
14  * represented as pointers into the original text. The suffixes in a suffix
15  * array  are sorted in lexicographical order. This way, occurrences of
16  * repeated substrings in the original text are neighbors in the suffix
17  * array.</p>
18  *
19  * For the above, example (assuming pointers are 0-based integers), the
20  * sorted suffix array is: [3,0,2,1]
21  */
22 public class LRS {
23
24
25    private /*@ spec_public @*/ int s = 0;
26    private /*@ spec_public @*/ int t = 0;
27    private /*@ spec_public @*/ int l = 0;
28    private /*@ spec_public @*/ final SuffixArray sa;
29
30    LRS (SuffixArray arr) { sa = arr; }
31
32
33    /*@ normal_behavior
34    @ requires \invariant_for(sa);
35    @ requires sa.a.length >= 2;
36    @ ensures 0 <= s && s < sa.a.length;
37    @ ensures 0 <= t && t < sa.a.length;
38    @ ensures 0 <= l && l < sa.a.length;
39    @ ensures s+l <= sa.a.length && t+l <= sa.a.length;
40    @ //ensures (\forall int j; 0 <= j && j < l; sa.a[s+j] == sa.a[t+j]);
41    @ ensures s != t || l == 0;
42    @ //ensures !(\exists int i,k; 0 <= i && i < k && k < sa.a.length-l;
43    @ //           (\forall int j; 0 <= j && j <= l; sa.a[k+j] == sa.a[i+j]));
44    @         // there is no LRS of length l+1
45    @*/
46  public void doLRS() {
47      int s = 0; int t = 0; int l = 0;
48
49      /*@ maintaining sa != null;
50        @//maintaining \invariant_for(sa);
51        @ maintaining 0 <= s && s < sa.a.length;
52        @ maintaining 0 <= t && t < sa.a.length;
53        @ maintaining 0 <= l && l < sa.a.length;
54        @ maintaining s+l <= sa.a.length && t+l <= sa.a.length;
55        @ maintaining s != t || l == 0;
56        @ maintaining 0 < x && x <= sa.a.length;
57        @ //maintaining (\forall int j; 0 <= j && j <l; sa.a[s+j] == sa.a[t+j]);
58        @ //maintaining !(\exists int w; 0 < w && w < x
59        @ //              && sa.suffixes[w-1] < sa.a.length-l
60        @ //              && sa.suffixes[w]   < sa.a.length-1;
61        @ //             (\forall int j; 0 <= j && j <= l;
62        @ //              sa.a[sa.suffixes[w-1]+j] == sa.a[sa.suffixes[w]+j]));
63        @ decreasing sa.a.length-x;
64        @*/
65      for (int x=1; x < sa.a.length; x++) {
66          int length = LCP.lcp(sa.a,sa.suffixes[x],
67                               sa.suffixes[x-1]);
68          if (length > l) {
69              s = sa.suffixes[x];
70              t = sa.suffixes[x-1];
71              l = length;
72      }   }
73      this.s = s; this.t = t; this.l = l;
74  }
75 }
76
77
78
79 //Based on wszcode by Robert Sedgewick and Kevin Wayne.
80
```

Figure 36: OpenJML - Longest Repeated Substring - LRS

```
package q1_2012;

final class PrefixSumRec {

    /*@ spec_public @*/ private final int[] a;
    /*@ spec_public @*/ private static int count = 1;

    //@//**accessible \inv: \singleton(a);
    //@ axiom evenSumLemma(); //axiom means assume named predicate is true


    /*@ requires array.length>0;
      @ requires array!=null;
      @// requires isPow2(a.length);
      @ ensures a.length>0 && a!= null;
    @*/
    PrefixSumRec(int [] array) {
    this.a = array;
    }

    /*@ normal_behavior
      @   ensures \result == (\forall int x, y; 0<=x && x<=100 && y<=x; even(x) == (even(y) == even(x+y)));
      @   ensures \result;
      @   accessible \nothing;
      @   pure helper spec_public
      @*/
    private static boolean evenSumLemma() { return true; }



    /*@ normal_behavior
    @   ensures \result == (x%2==0);
    @   accessible \nothing;
    @   pure helper spec_public
    @*/
    private static boolean even (int x)
    { return x%2==0;}


    /*@ 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;
      }


     /*@ public normal_behavior
       @   requires x == 3;
       @// ensures \result == (\product int i; 0 <= i && i < x; 2);
       @   ensures \result > 0 && \result < Integer.MAX_VALUE;
       @// measured_by x;
       @   assignable count;
       @   spec_public
       @*/
       private static long pow2(int x) {
          // return x==0? 1: 2*pow2(x-1);
         count = 1;

         //@ maintaining count >= 1 && x >= 0;
         while(x>0)
         {
          count += count;
          x--;
         }
         return count;
       }


        /*@ 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;
      }


      /*@
       @ requires 0 <= a.length && a.length <= Integer.MAX_VALUE / 2;
       @ 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;
         }



}


final class Runner{
    public static void main(String[] args)
    {
        PrefixSumRec p = new PrefixSumRec(new int [5]);
        //System.out.println(p.pow2(0));
    }
}
```

Figure 37: OpenJML - Longest Repeated Substring - SuffixArray

OpenJML Errors

RAC Error

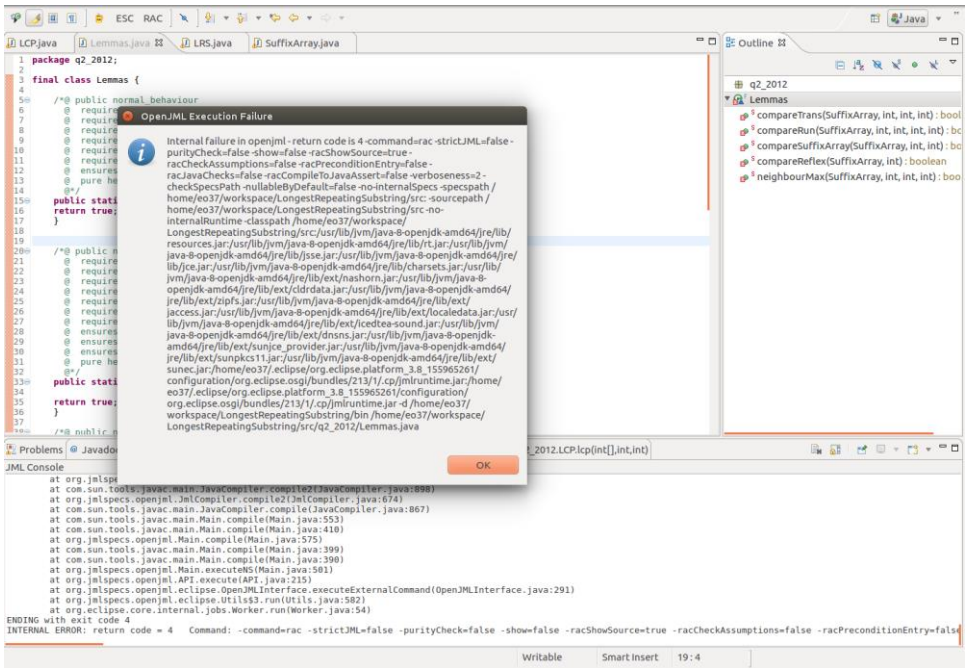


Figure 38: OpenJML - RAC internal error

OpenJML ESC error

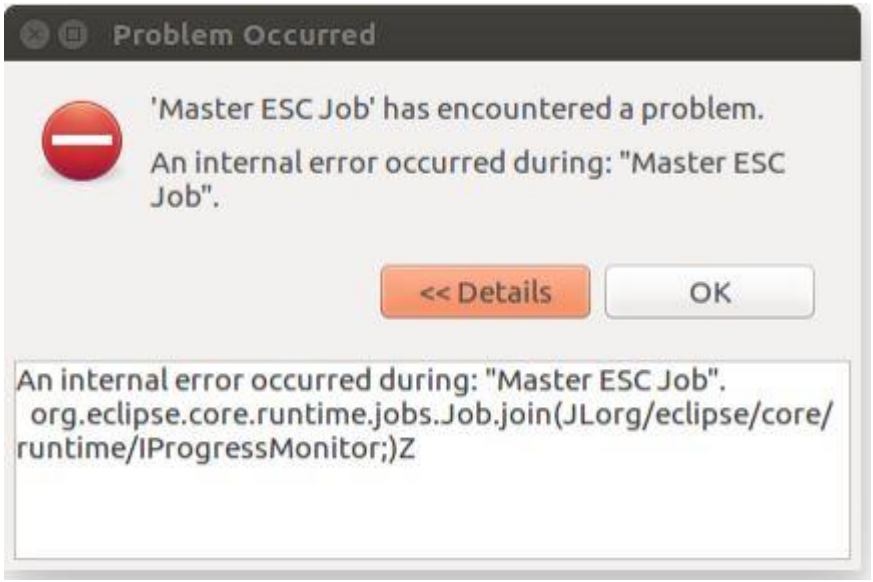


Figure 39: OpenJML Error Warning