



# UNIVERSIDAD DE LA RIOJA

Facultad

Facultad de Ciencias, Estudios Agroalimentarios e Informática

Titulación

Grado en Ingeniería Informática

Título

**Verificación de Programas Java para el Procesamiento de  
Imágenes Digitales**

Autor/es

Rubén Sáenz Francia

Tutor/es

Dr. Julio Rubio García

Departamento

Matemáticas y Computación

Curso académico

2012 - 2013

**Abstract.** Fiji is a Java platform widely used by biologists and other experimental scientists to process digital images. In our research, made together with a biologists team, we use Fiji in some pre-processing steps before undertaking a homological digital processing of images. In a previous work, the team formalised the correctness of the programs which use homological techniques to analyse digital images. However, the verification of Fiji's pre-processing step was missed and NeuronPersistentJ plugin had some working issues that needed to be corrected. In this dissertation, we present a *multi-tool* approach (based on the combination of Why/Krakatoa, Coq and ACL2) filling this gap.

## Table of Contents

|     |   |    |
|-----|---|----|
| 1   | Introduction.....   | 1  |
| 2   | Project Goal Statement (PGS) .....                        | 2  |
| 2.1 | Context .....   | 2  |
| 2.2 | Goals and Objectives .....                                | 2  |
| 2.3 | Deadlines .....   | 2  |
| 2.4 | Key Concepts.....   | 3  |
| 2.5 | Requirements .....  | 3  |
| 2.6 | Existing Verification Tools.....                          | 4  |
| 2.7 | Work Breakdown Structure (WBS) and estimates of time..... | 5  |
| 2.8 | Global estimates of time .....                            | 7  |
| 2.9 | Gantt Diagram.....  | 8  |
| 3   | Analysis .....  | 9  |
| 3.1 | Tools - Krakatoa and ACL2 .....                           | 9  |
| 3.2 | Method .....  | 12 |
| 3.3 | NeuronPersistentJ .....                                   | 14 |
| 3.4 | Java Parser.....  | 19 |
| 4   | Design.....   | 20 |
| 4.1 | Class Unifier.....  | 20 |
| 4.2 | Verification Platform .....                               | 22 |
| 4.3 | MakeLineRadii.....  | 23 |
| 5   | Development .....   | 23 |
| 5.1 | NeuronPersistentJ Plugin.....                             | 23 |
| 5.2 | Verification Platform .....                               | 24 |
| 5.3 | Specifying programs for digital imaging .....             | 26 |
| 5.4 | The role of ACL2 .....                                    | 29 |
| 5.5 | The method in action: a complete example .....            | 30 |
| 5.6 | The role of junit .....                                   | 35 |
| 6   | Final Estimates of Time .....                             | 37 |
| 7   | Conclusions and further work.....                         | 38 |
| A   | Install and Configure Krakatoa using Eclipse .....        | 41 |
| B   | Krakatoa Operators .....                                  | 46 |

## 1 Introduction

Fiji [29] is a Java platform widely used by biologists and other experimental scientists to process digital images. In our research, made together with a biologists team, we use Fiji in some pre-processing steps before undertaking a homological digital processing of images.

Due to the fact that the reliability of results is instrumental in biomedical research, we are working towards the certification of the programs that we use to analyse biomedical images – here, certification means verification assisted by computers. In a previous work, see [17, 19], we have formalised two homological techniques to process biomedical images. However, in both cases, the verification of Fiji’s pre-processing step was not undertaken.

Being a software built by means of plug-ins developed by several authors, Fiji is messy, very flexible (program pieces are used in some occasions with a completely different objective from the one they were designed), contains many redundancies and dead code, and so on. In summary, it is a big software system which has not been devised to be formally verified. So, this endeavour is challenging.

There are several approaches to verify Java code; for instance, proving the correctness of the associated Java bytecode, see [24]. In this dissertation, we use Krakatoa [12] to specify and prove the correctness of Fiji/Java programs. This experience allows us to evaluate both the verification of *production* Fiji/Java code, and the Krakatoa tool itself in an unprepared scenario.

Krakatoa uses some automated theorem provers (as Alt-Ergo [6] or CVC3 [4]) to discharge the proof obligations generated by means of the Why tool [12]. When a proof obligation cannot be solved by means of the automated provers, the corresponding statement is generated in Coq [10]. Then, the user can try to prove the missing property by interacting with this proof assistant.

In this picture, we add the ACL2 theorem prover [22]. ACL2 is an automated theorem prover but more powerful than others. In many aspects, working with ACL2 is more similar to interactive provers than to automated ones, see [22]. Instead of integrating ACL2 in the architecture of Why/Krakatoa, we have followed another path leaving untouched the Why/Krakatoa code. Our approach reuses a proposal presented in [3] to translate first-order Isabelle/HOL theories to ACL2 through an XML specification language called XLL [3]. We have enhanced our previous tools to translate Coq theories to the XLL language, and then apply the tools developed in [3] to obtain ACL2 files. In this way, we can use, unmodified, the Why/Krakatoa framework; the Coq statements are then translated (if needed) to ACL2, where an automated proof is tried; if it succeeds, Coq is only an intermediary specification step; otherwise, both ACL2 or Coq can be interactively used to complete the proof.

The programs and examples that were presented to the Conference on Intelligent Computer Mathematics are available at <http://www.computing.dundee.ac.uk/staff/jheras/vpdims/>.

## 2 Project Goal Statement (PGS)

### 2.1 Context

Fiji [29] is a Java program which can be described as a distribution of ImageJ [28]. These two programs help with the research in life sciences and biomedicine since they are used to process and analyse biomedical images. Fiji and ImageJ are open source projects and their functionality can be expanded by means of either a macro scripting language or Java plug-ins. Among the Fiji/ImageJ plug-ins and macros, we can find functionality which allows us to binarise an image via different threshold algorithms, homogenise images through filters such as the “median filter” or obtain the maximum projection of a stack of images.

In the frame of the ForMath European project [1], one of the tasks is devoted to the topological aspects of digital image processing. The objective of that consists in formalising enough mathematics to verify programs in the area of biomedical imaging. In collaboration with the biologists team directed by Miguel Morales, two plug-ins for Fiji have been developed (SynapCountJ [26] and NeuronPersistentJ [25]); these programs are devoted to analyse the effects of some drugs on the neuronal structure. At the end of such analysis, some homological processing is needed (standard homology groups in SynapCountJ and persistent homology in NeuronPersistentJ). As explained in the introduction, we have verified these last steps [17, 19]. But all the pre-processing steps, based on already-built Fiji plug-ins and tools, kept unverified. This is the gap we try to fill now, by using the facilities presented in the sequel.

### 2.2 Goals and Objectives

- Analyse existing verification tools for Java programs.
- Choose and use the most suitable tool to prove Java code.
- Define a methodology which provides a general process to compile the Java Code in the tool that we choose.
- Automate the methodology.
- Choose some parts of ImageJ source code that are suspicious of working improperly to be proved .
- Debug the ImageJ plugin called NeuronPersistentJ [25] which has some working issues.

### 2.3 Deadlines

In order to make a schedule of the tasks that need to be accomplished, we have to take into account two important events.

- Conferences on Intelligent Computer Mathematics 2013.  
This is a Computing and Mathematics Conference that is taking place in the University of Bath where we hope we can write a paper which will include some parts of this dissertation.

- Submission deadline: 12 March 2013
  - Reviews sent to authors: 5 April 2013
  - Rebuttals due: 8 April 2013
  - Notification of acceptance: 14 April 2013
  - Camera ready copies due: 26 April 2013
  - Conference: 8-12 July 2013
- La Rioja University Project Deadline.
- These are the Deadline dates established by La Rioja University in order to submit the dissertation and be able to defend it and achieve the Degree in Computer Science.
- Submission Deadline June: Second Week of June 2013
  - Defense of the Project June: Last Week of June 2013
  - Submission Deadline July: Second Week of July 2013
  - Defense of the Project July: Third Week of July 2013

## 2.4 Key Concepts

- **Object-oriented**: is a programming paradigm that represents concepts as "objects" that have data fields (attributes that describe the object) and associated procedures known as methods.
- **Java**: is a general-purpose, concurrent, class-based, object-oriented computer programming language that is specifically designed to have as few implementation dependencies as possible.
- **ImageJ**: is a public domain, Java-based image processing program developed at the National Institutes of Health.
- **JML**: The Java Modeling Language (JML) is a specification language for Java programs, using Hoare style pre- and postconditions and invariants, that follows the design by contract paradigm. Specifications are written as Java annotation comments to the source files, which hence can be compiled with any Java compiler.

## 2.5 Requirements

During this section we identify the tasks that go into determining the needs or conditions of our dissertation. We can classify the requirements into these two sections:

**Functional Requirements**, that define the functions that the dissertation should have.

- NeuronPersistentJ plugin needs to detect the neuronal structure from an image without any failure.

- Some parts of the code related to NeuronPersistentJ plugin are needed to be proved using formalisation, therefore we need to select the most suitable ones.
- We must define a procedure to easily formalise Java Code and automate it.

**Non functional requirements**, that specify criteria that can be used to judge the operation of the system, rather than specific behaviors.

- We must use formalisation software in our approach.
- We must verify Java Code, therefore the formalisation software needs to be Java compatible.
- We have to select formalisation software compatible with programming languages that are used in the Department (Coq, Isabelle or ACL2).

## 2.6 Existing Verification Tools

- **jUnit**: JUnit is a unit testing framework for the Java programming language.  
*Pros*: Most used testing framework.  
*Cons*: It is not based on formalisation.
- **KeY**: KeY is a formal software development tool that aims to integrate design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible. At the core of the system is a novel theorem prover for the first-order Dynamic Logic for Java with a user-friendly graphical interface.  
*Pros*: JML/OCL support.  
*Cons*: It does not export to Coq proof assistant.
- **ESC/Java2**: originally developed at Compaq Research, is a programming tool that attempts to find common run-time errors in JML-annotated Java programs by static analysis of the program code and its formal annotations. Users can control the amount and kinds of checking that ESC/Java2 performs by annotating their programs with specially formatted comments called pragmas.  
*Pros*: Standard JML notation, Eclipse plugin.  
*Cons*: Very complex tool and it does not export to the Coq language.
- **Krakatoa**: It is a front-end of the Why platform for deductive program verification.  
*Pros*: Coq-language export, Java and C verification tool, plenty of back-end automated provers, available on Ubuntu repositories.  
*Cons*: KML (Krakatoa Modelling Language) which is a self made JML syntax. It is only available for Unix operating systems.

## 2.7 Work Breakdown Structure (WBS) and estimates of time

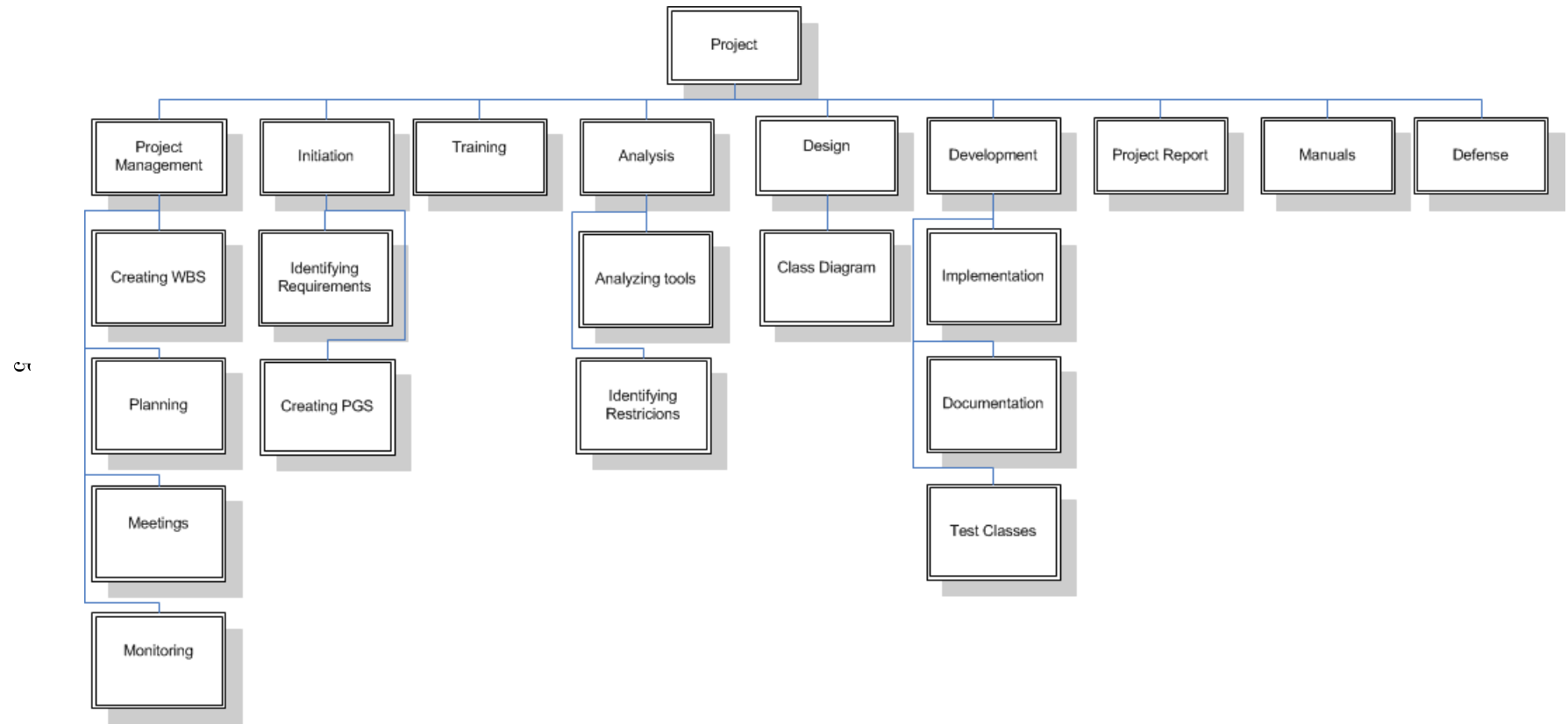


Fig. 1: Work Breakdown Structure (WBS)



**Project Management** This task consists on planning, organizing, motivating, and controlling the resources to achieve specific goals.

*WBS Creation* This section provides the necessary framework for detailed time estimating and control along with providing guidance for schedule development and monitoring. Estimates: **5 hours**

*Planning* This sections consists of estimating, as real as possible, the time that the tasks defined in the previous item will take. Estimates: **4 hours**

*Meeting* This section contains the subtasks associated with the Project meetings such as: preparation of the meeting, writing meeting records... Estimates: **6 hours**

*Monitoring* During this section we ensure that the estimates of hours and the real used hours differ as few as possible measuring it and making decisions concerning the adequacy to the scheduled timetable. Estimates: **2 hours**

**Initiation** It is the phase in which we define our scope and, with a clearly defined target, we can face the complete project.

*Identifying Requirements* During this section we discover the needs of this project. Estimates: **53 hours**

*Creating PGS* This section contains the creation of a work breakdown structure (WBS) which helps us to be both comprehensive and specific when managing the project. Estimates: **10 hours**

**Training** During this section we study and learn a variety of technologies with the purpose of gaining experience on those subjects. Estimates: **178 hours**

**Analysis** This section encompasses those tasks for determining the needs or conditions of the project, taking account of the possibly conflicting requirements of the technologies.

*Analyzing Tools* During this section we analyse different technologies that can match in our project in order to fulfill our needs. Estimates: **54 hours**

*Identifying Restrictions* During this section we analyse the constraints of the technologies and tools. Estimates: **70 hours**

**Design** During this phase, we assemble the information we gathered from the requirements capture phase. We identify what we need in our project according to the selected tools and technologies we have chosen.

*Class Diagram* This section consists of modelling Class Diagrams that are essential to the object modeling process and the static structure of the project. Estimates: **205 hours**

**Development** During this section we build the solution components code as well as documentation and Test Classes.

*Implementation* Given the requirements and Class Diagram we build exactly what was planned taking into account the restrictions of the system. Estimates: **129 hours**

*Documentation* During this phase we detail the design and functionality of the application. Estimates: **42 hours**

*Test Classes* This section consists of building the mechanisms that will help us to detect bugs in the system. Estimates: **27 hours**

**Project Report** During this section we write a summarised report of what we have learnt and made during this project and further work related to this project. Estimates: **99 hours**

**Manuals** During this section we write manuals that step-by-step help people to use some tools. Estimates: **18 hours**

**Defense** Having all the tasks completed, we defend the current project in front of a panel. Estimates: **8 hours**

## 2.8 Global estimates of time

The project Analysis, Design and Development (See Figure 2) are divided into two phases with the purpose of having done the first part before the Submission Deadline (See Section 2.3). The estimated hours that the project is taking is obtained from the formula:

$6.5 \text{ hours/day (Average Active Worktime)} \times 7 \text{ months} \times 20 \text{ hours (Active Working Days)} = 910 \text{ hours.}$

The total number of hours (910) is a much higher value than established by recommendation for the project but we are taking into account that this dissertation is going to be reused and attached to the Documentation of the European Project ForMath.

## 2.9 Gantt Diagram

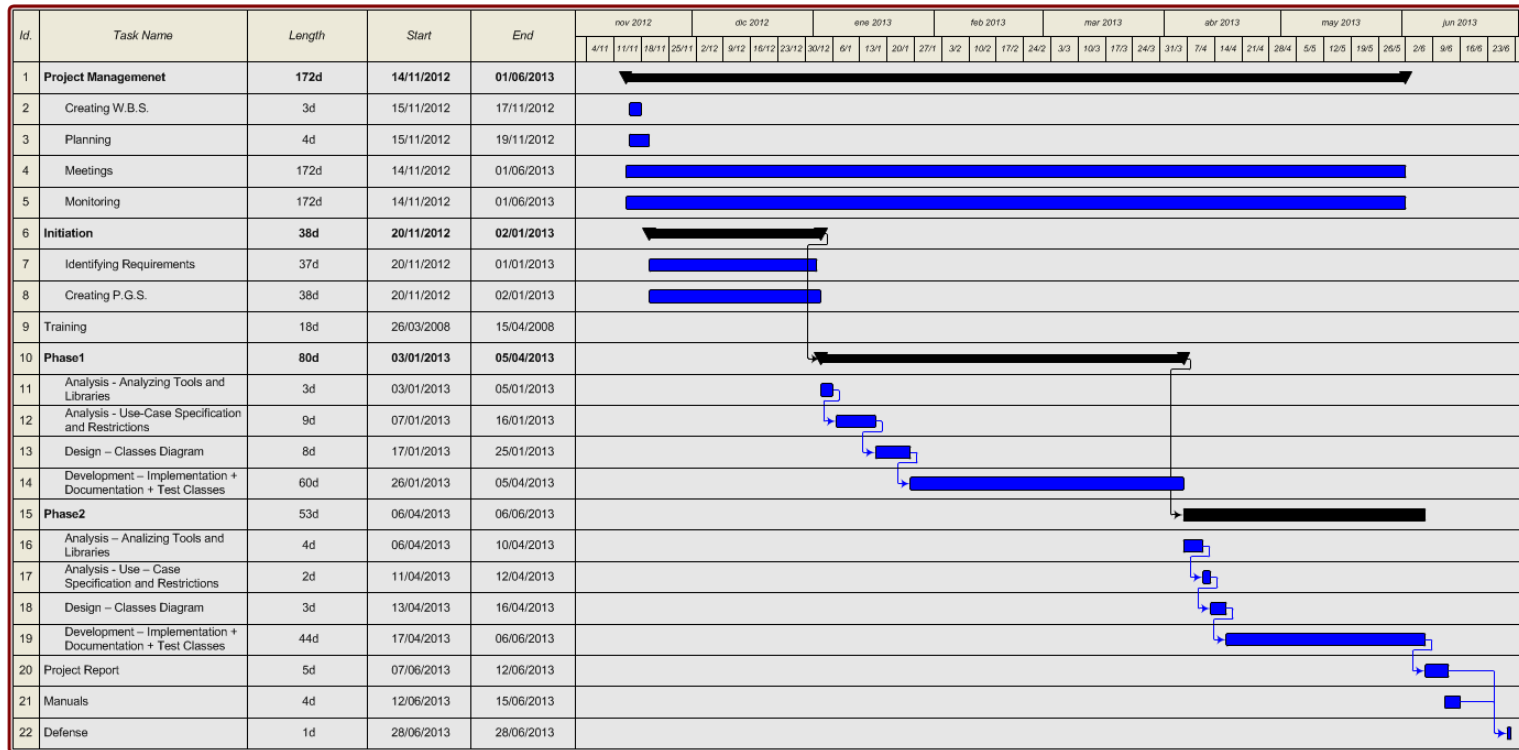


Fig. 2: Gantt Diagram

## 3 Analysis

### 3.1 Tools - Krakatoa and ACL2

**Why/Krakatoa: Specifying and verifying Java code.** The Why/Krakatoa tools [12] are an environment for proving the correctness of Java programs annotated with JML [8] specifications which have been successfully applied in different context, see [5]. The environment involves three distinct components: the Krakatoa tool, which reads the annotated Java files and produces a representation of the semantics of the Java program into Why's input language; the Why tool, which computes proof obligations (POs) for a core imperative language annotated with pre- and post-conditions, and finally several automated theorem provers which are included in the environment and are used to prove the POs. When some PO cannot be solved by means of the automated provers, corresponding statements are automatically generated in Coq [10], so that the user can then try to prove the missing properties in this interactive theorem prover. The POs generation is based on a Weakest Precondition calculus and the validity of all generated POs implies the soundness of the code with respect to the given specification. The Why/Krakatoa tools are available as open source software at <http://krakatoa.lri.fr>.

#### *Krakatoa Semantics*

- **Krakatoa statements:** The comments starting with `/*@` specify Krakatoa clauses.
- **Precondition:** a *requires* statement introduces a precondition. This is a formula that is supposed to hold at the beginning of the method call. The precondition is not something guaranteed by the method itself.
- **Postcondition:** an *ensures* clause introduce a postcondition, which is a formula supposed to hold at end of execution of that method, for any value of its arguments satisfying its precondition.
- **Splitted Postcondition:** a *behavior* statement allows us to split the postcondition in different sections. A normal behavior clause has the form

```
behavior id :  
  assumes A ;  
  ensures E ;
```

The semantics of such a behaviour is as follows. The callee guarantees that if it returns normally, then in the post-state:

$\backslash\text{old}(A) \Rightarrow E$  holds

If  $\backslash\text{old}(A)$  holds, each location remains allocated and unchanged in the post-state.

- **Assertions:** an *assert* clause, it specifies that the given property is true at the corresponding program point.
- **Loop invariants:** a *loop\_invariant* statement, is a property that is true at loop entrance, and it preserves it by loop body and loop exit.
- **Loop variants:** a *loop\_variant* clause establishes an integer value that decreases on each iteration of the loop.
- **Lemmas:** a *lemma* statement defines a proven proposition.
- **Axioms:** an *axiom* clause is a premise or starting point of reasoning. It can only be used in order to define properties of abstract data types.
- **Ghost Variables:** a *ghost* statement is used to define or redefine the value of a variable in Krakatoa.

### Krakatoa Structure

*Krakatoa External Provers Compatibility* Krakatoa is a tool that can support a wide range of provers but we have to install it independent from Krakatoa. Each time a new prover is installed, we must rerun the command `why-config` (in Why 2.xx) and/or `why3config -detect` (in Why3) in order to charge into the Why core. We can classify the provers into two important groups.

- Automated Provers. Figure 3.  
On the one hand, automatic provers provide automation, but only allow first order logic with equality. See the list in Table 1.

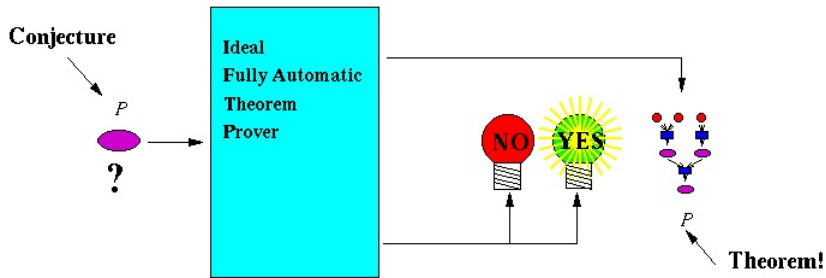


Fig. 3: Automated Prover Structure

|          |         |          |       |          |
|----------|---------|----------|-------|----------|
| Alt-Ergo | CVC3    | E-prover | Gappa | Simplify |
| SPASS    | Vampire | veriT    | Yices | Z3       |

Table 1: Krakatoa Supported Automated Provers

- Interactive Probers. Figure 4.

On the other hand, interactive provers offer users expressive formalisms and flexibility and are suitable for proving theorems on other logics. See the list in Table 2.

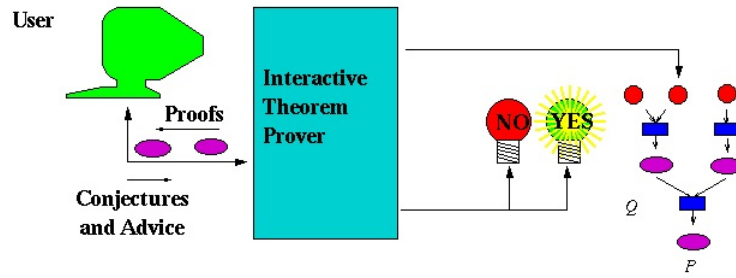


Fig. 4: Interactive Prover Structure

|     |     |              |       |           |       |
|-----|-----|--------------|-------|-----------|-------|
| Coq | PVS | Isabelle/HOL | HOL 4 | HOL Light | Mizar |
|-----|-----|--------------|-------|-----------|-------|

Table 2: Krakatoa Supported Interactive Provers

#### *Krakatoa Restrictions*

1. Annotations are not allowed.  
For instance: *@Before;*
2. Float and Double numbers that match the pattern <number>f or <number>d throws compilation errors.  
For instance: *float f = 5.0f;*
3. Import statements are forbidden if they are included in the file that we want to compile.  
For instance: *import java.lang.String;*
4. Inheritance is not allowed to appear in the file we compile. For this reason *extends* and *implements* clauses are forbidden.  
For instance: *class A extends B implements C;*

5. Else clauses without an if condition usually throws compilation errors.  
For instance: *if(a==b){return a;} else{ return b;}*
6. Numbers like 0x1.fffffeP+127d throw compilation errors.  
For instance: *float a = 0x1.fffffeP+127d;*
7. Double to integer casts are forbidden.  
For instance: *int i = (int) (Double.parseDouble("2.4"));*

**Coq and ACL2: Interactive theorem proving.** Coq [10] is an interactive proof assistant for constructive higher-order logic based on the Calculus of Inductive Construction. This system provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. Coq has been successfully used in the formalisation of relevant mathematical results; for instance, the recently proven Feit-Thompson Theorem [14].

ACL2 [22] is a programming language, a first order logic and an automated theorem prover. Thus, the system constitutes an environment in which algorithms can be defined and executed, and their properties can be formally specified and proved with the assistance of a mechanical theorem prover. ACL2 has elements of both interactive and automated provers. ACL2 is automatic in the sense that once started on a problem, it proceeds without human assistance. However, non-trivial results are not usually proved in the first attempt, and the user has to lead the prover to a successful proof providing a set of lemmas, inspired by the failed proof generated by ACL2. This system has been used for a variety of important formal methods projects of industrial and commercial interest [16] and for implementing large proofs in mathematics.

### 3.2 Method

In this section, we present the method that we have applied to verify Fiji code. This process can be split into the following steps.

1. Transforming Fiji code into compilable Krakatoa code.
2. Specifying Java programs.
3. Applying the Why tool.
4. If all the proof obligations are discharged automatically by the provers integrated in Krakatoa, stop; the verification has ended.
5. Otherwise, study the failed attempts, and consider if they are under-specified; if it is the case, go again to step (2).
6. Otherwise, consider the Coq expressions of the still-non-proven statements and transform them to ACL2.
7. If all the statements are automatically proved in ACL2, stop; the verification has ended.
8. Otherwise, by inspecting the failed ACL2 proofs, decide if other specifications are needed (go to item (2)); if it is not the case, decide if the missing proofs should be carried out in Coq or ACL2.

The first step is the most sensitive one, because it is the only point where informal (or, rather, semi-formal) methods are needed. Thus, some unsafe, and manual, code transformation can be required. To minimize this drawback, we apply two strategies:

- First, only well-known transformations are applied; for instance, we eliminate inheritance by “flattening” out the code, but without touching the real behaviour of methods.
- Second, the equivalence between the original code and the transformed one is systematically tested.

Both points together increase the reliability of our approach; a more detailed description of the transformations needed in step (1) are explained in Section 3.2. Step (2) is quite well-understood, and some remarks about this step are provided in Section 5.3. Steps (3)-(6) are mechanized in Krakatoa. The role of ACL2 (steps (6)-(8)) is explained in Section 5.4 and, by means of an example, in Section 5.5.

**Transforming Fiji-Java to Krakatoa-Java** In its current state, the Why/Krakatoa system does not support the complete Java programming language and has some limitations. See Section 3.1. In order to make a Fiji Java program compilable by Krakatoa we have to take several steps.

1. Delete annotations. Krakatoa JML annotations will be placed between `\*` and `@`. Therefore, we need to remove other Java Annotations preceded by `@`.
2. Move the classes that are referenced in the file that we want to compile into the directory *whyInstallationDir/java\_api/*. For example, the class *RankFilters* uses the class *java.awt.Rectangle*; therefore, we need to create the folder *awt* inside the *java* directory that already exists, and put the file *Rectangle.java* into it. Moreover, we can remove the body of the methods because only the headers and the fields of the classes will be taken into consideration. We must iterate this process over the classes that we add. The files that we add into the *java\_api* directory can contain `import`, `extends` and `implements` clauses although the file that we want to compile cannot do it – Krakatoa does not support these mechanisms. This is a tough process: for instance, to make use of the class *Rectangle*, we need to add fifteen classes.
3. Reproduce the behaviour of the class that we want to compile. Considering that we are not able to use `extends` and `implements` clauses, we need to move the code from the upper classes into the one that we want to compile in order to have the same behaviour. For instance, the class *BinaryProcessor* extends from *ByteProcessor* and inside its constructor it calls the constructor of *ByteProcessor*; to solve this problem we need to copy the body of the super constructor at the beginning of the constructor of the class *BinaryProcessor*. If we find the use of interfaces, we can ignore them and remove the `implements` clause because the code will be implemented in the class that makes use of the interface.



4. Remove `import` clauses. We need to delete them from the file that we want to compile and change the places where the corresponding classes appear with the full path codes. If for example we are trying to use the class *Rectangle* as we have explained in Step 2, we need to replace it by *java.awt.Rectangle*.
5. Owing to package declarations are forbidden, we need to remove them with the purpose of halting “*unknown identifier packageName*” errors.
6. Rebuild native methods. The Java programming language allows the use of *native* methods, which are written in C or C++ and might be specific to a hardware and operating system platform. For example, many of the methods in the class *Math* (which perform basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions) simply call the equivalent method included in a different class named *StrictMath* for their implementation, and then the code in *StrictMath* of these methods is just a *native* call. Since native methods are not written in Java, they cannot be specified and verified in Krakatoa. Therefore, if our Fiji program uses some native methods, it will be necessary to rewrite them with our own code. See in Section 5.5 our implementation (and specification) of the native method `sqr` computing the square root of a number of type double, based on Newton’s algorithm.
7. Add a clause in *if-else* structures in order to remove “*Uncaught exception: Invalid\_argument(“equal: abstract value”)*”. We can find an example in the method `filterEdge` of the class *MedianFilter* where we have to replace the last *else...* clause by *else if(true)....*
8. Remove debugging useless references. We have mentioned in a previous step that we can only use certain static methods that we have manually added to the Why core code and therefore we can remove some debugging instructions like `System.out.println(...)`. We can find the usage of standard output printing statement in the method `write` of the class *IJ*.
9. Modify the declaration of some variables to avoid syntax errors. There can be some compilation errors with the definition of some floats and double values that match the pattern `<number>f` or `<number>d`. We can see an example in the line 180 of the file *RankFilters.java*; we have to transform the code: `float f = 50f;` into `float f = 50.`
10. Change the way that Maximum and Minimum float numbers are written. Those two special numbers are located in the file *Float.java* and there are widely used to avoid overflow errors, but they generate an error due to the *eP* exponent. To stop having errors with expressions like `0x1.fffffeP+127d` we need to convert it into `3.4028235e+38f`.

### 3.3 NeuronPersistentJ

NeuronPersistentJ [25] is a plugin for ImageJ and Fiji that implements a method to detect the neuronal structure from an image. See Figure 5

This method can be split into two steps, the first one process the image with filters to dismiss the elements which are not part of the structure of the

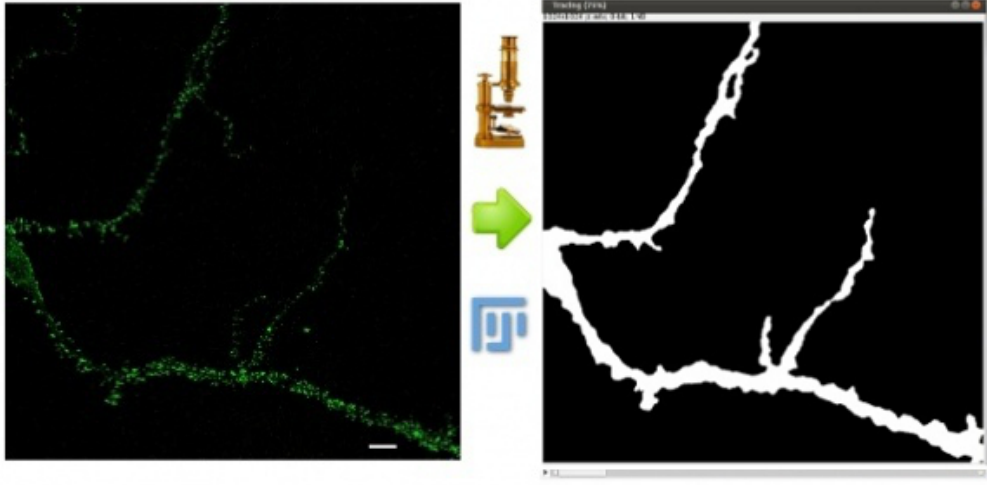


Fig. 5: NeuronPersistentJ in action

main neuron, and the second one is based in the persistent homology. This is a technique which allows us to study the lifetimes of topological attributes.

There exists an unidentified error on NeuronPersistentJ which eventually outputs results that do not correspond with the homological groups of the original image, therefore we need to detect where the problem is and modify the code in order to solve it and convert it into a trustful and robust plugin.

NeuronPersistentJ uses Median Filter (See section `refsec:medianFilter`) and `makeLineRadii` (See Section `refsec:makeRadius`) functions which are described on the following Sections.

**Median Filter** It is impossible to design a filter that removes any noise but keeps all the important image structures intact because no filter can discriminate which information is important for a human being and which is not.

The median filter replaces every value of each pixel by the median of the filters corresponding filter region  $R$ .

$$I'(u,v) \leftarrow \text{median}\{I(u+i,v+j) \mid (i,j) \in R\}.$$

The median of  $2K+1$  pixel is defined as:

$$\text{median}(p_0, p_1, \dots, p_K, \dots, p_{2K}) \triangleq p_K;$$

In order to determine the domain of the filter region a prompt windows is shown (See Figure 6) asking us to manually establish it.

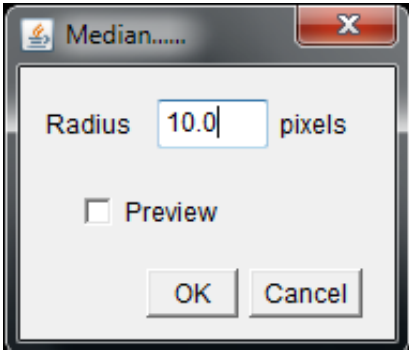


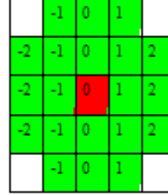
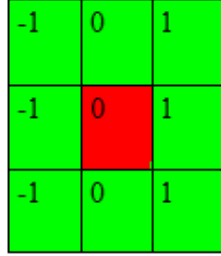
Fig. 6: Prompt window of the Median Filter asking the user to establish a domain.

**MakeLineRadii Function** MakeLineRadii is a very useful function which allows us to determine the pixels that are being taken into account to compute the median value of a pixel; so it creates a circular kernel (structuring element) of a given radius. The function is located in the file *RankFilters.java* which is placed into *ij.plugin.filters* package.

The declaration of the method is:

```
protected int[] makeLineRadii(double radius)
```

The returned array represents a circle of pixels as you can see in Figure 7.

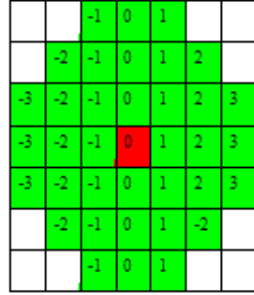


Output: -1 1 -1 1 -1 1 9 1

Output: -1 1 -2 2 -2 2 -2 2 -1 1 21 2

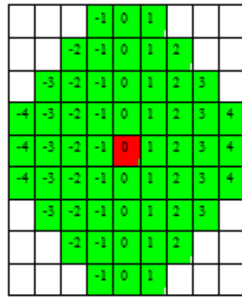
(a) MakeLineRadii Radius 1

(b) MakeLineRadii Radius 2



Output: -1 1 -2 2 -3 3 -3 3 -3 3 -2 2 -1 1 37 3

(c) MakeLineRadii Radius 3



Output: -1 1 -2 2 -3 3 -4 4 -4 4 -4 4 -3 3 -2 2 -1 1 57 4

(d) MakeLineRadii Radius 4

Fig. 7: MakeLineRadii outputs

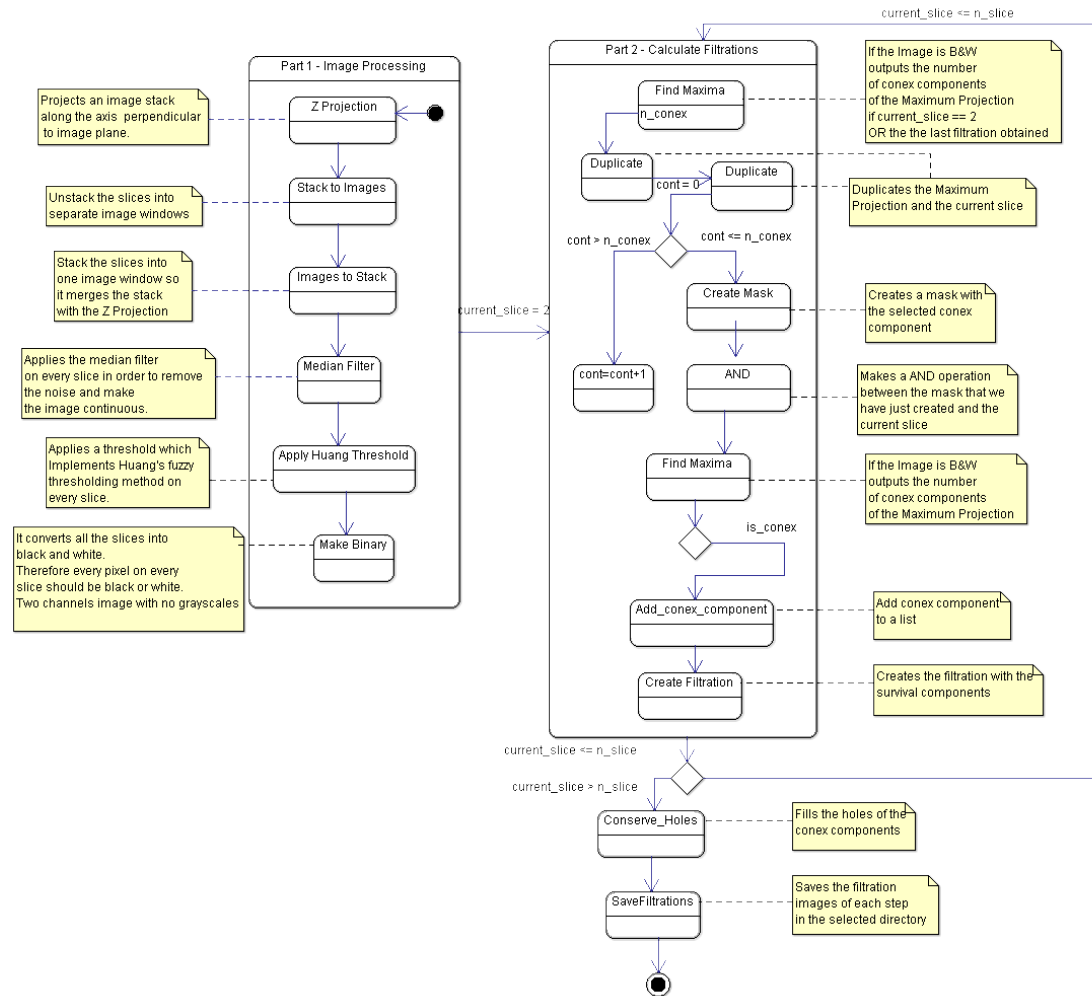


Fig. 8: NeuronPersistentJ State Diagram

The total number of elements that are taken into account in order to set the median value is defined by the formula:

$$\begin{aligned}
 & roint = (int)\sqrt{(int)(radius^2) + 1 + 1e - 10} \\
 & \text{Total Number of Pixels:} \\
 & 2*roint+1+4*(\sum_{i=1}^{roint} (int)\sqrt{(int)(radius^2) + 1 - i^2 + 1e - 10})+2*\sum_{i=1}^{roint} i
 \end{aligned}$$

This formula is obtained because:

1.  $2*roint+1$ : Number of pixels that belong to the row with the red pixel in the center.
2.  $(int)\sqrt{(int)(radius^2) + 1 - i^2 + 1e - 10}$ : For each positive row  $i$  with a given integer  $radius$ , the maximum positive value of the row.
3.  $4 * \text{Step 2}$ : Positive and negative number of pixels of a positive row plus the same number of pixels of the negative row.
4.  $2*\sum_{i=1}^{roint} i$ : For each positive row  $i$ , the central pixel of the positive and negative row.

Therefore, regarding the previous results, the length of the returned array is  $2*roint+2$  because we need to add at the end of the array two elements that are taken into account in order to set the median value, and  $roint$ .

The default behaviour of `makeLineRadii` function follows the rules that we have just described, but according to the source code, there are two exceptions to these rules that we need to consider in order to specify the postcondition of the function:

1. If  $1.5 \leq radius < 1.75 \Rightarrow$  It returns `makeLineRadii(1.75)`.
2. If  $2.5 \leq radius < 2.85 \Rightarrow$  It returns `makeLineRadii(2.85)`.

### 3.4 Java Parser

Javaparser [2] is a Open Source library written in Java created with javacc that aims to generate an easy to use and lightweight source parser.

Using this package is quite easy. There exists two major classes from which we have to extend all the functionality:

- **VoidVisitorAdapter**: It allows us to modify the code without changing the structure.
- **ModifiableVisitorAdapter**: It allows us to modify the code and change the structure of the code. For instance we can change an if-else statement for a switch statement.

## 4 Design

### 4.1 Class Unifier

The classes contained on this section take part of a program that aims at solving the Krakatoa compilation problems commented on Section 3.1. The classes in Figure 9 are divided into three sections:

- **javaParser**: It contains the classes that use Java Parser library.
- **Eclipse Plugin**: It contains the classes that use Eclipse libraries.
- **ClassUnifier**: It is the middleware that aims to communicate the two previous parts.

The classes listed below are displayed in Figure 9.

**RemoveAnnotationsVisitor** This class allows us to remove the Annotations that the source code has. This class fixes Restriction 1.

**FloatsWithNumbersVisitor** This class allows us to change how the numbers are declared and transform them to default floating point notation. This class fixes Restrictions 2 and 6.

**ElseAlwaysTrueVisitor** This class allows us to change the structure of if-else statements by adding an "always true" condition on else statements. This class fixes Restriction 5.

**MethodCallerChangerVisitor** This class allows us to change the caller of every function and variable and replace it by the first of a List of visited Classes. This class is useful to solve part of Restriction 4.

**MethodNameChangerVisitor** This class allows us to change the method name of a function. This class is useful to solve part of Restriction 4.

**SuperChangerVisitor** This class allows us to change the references of a given caller for another one and removes super statements. This class is useful to solve part of Restriction 4.

**PathLocator** This class contains static methods and fields that allow us to outsource the location of the Java Class Files using Eclipse Plugin Resources.

**MainAppHandler** This class is the entrance point of the Application and it launches the method `execute(ExecutionEvent)` when the user requests it.

**ClassUnifier** This class is the middleware that communicates `javaParser` and `eclipsePlugins` parts. It has a `CompilationUnit` field that represents the father of the current class.

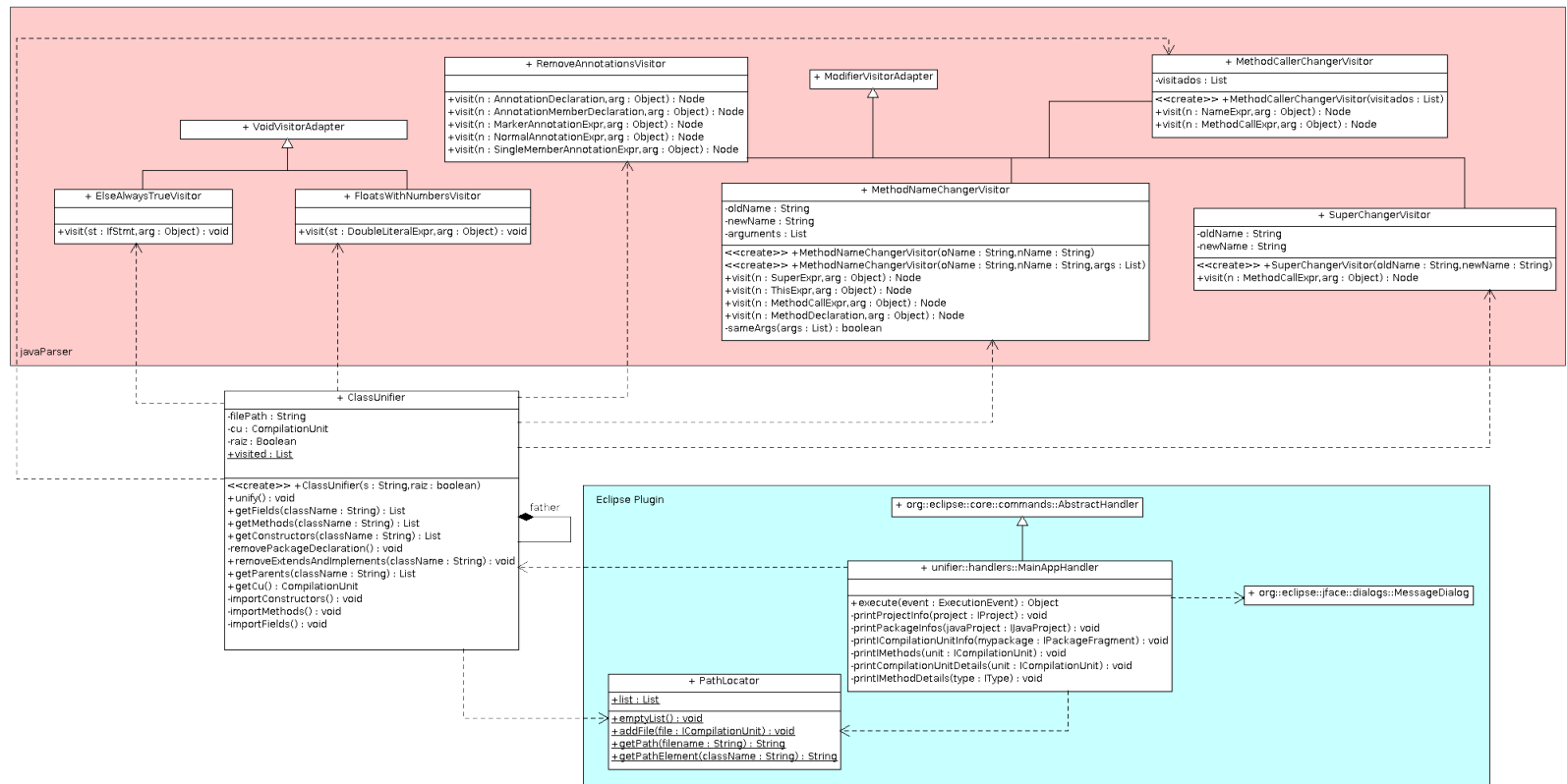


Fig. 9: Class Unifier - Class Diagram



## 4.2 Verification Platform

In order to verify the code, we need to define a Standard Architecture capable of facing a multi-strategy approach. The architecture (See Figure 10) consists on:

- KrakatoaProbe: Represents the file that we want to verify using Krakatoa.
- Utils.java: It is a utility class that we call from KrakatoaProbe file. This file is located under the Krakatoa installation directory and it is loaded through a symbolic link.
- UtilsTest.java: Is the jUnit class file that we use to make Unitary Tests.

The reason why the architecture is defined in this way, is because external classes are referenced in reserved packages and therefore we need to move it to allowed package names in order to make a safe execution environment.

For this reason, the classes are moved using an ant-script<sup>1</sup> from the unsafe to the safe execution environment.

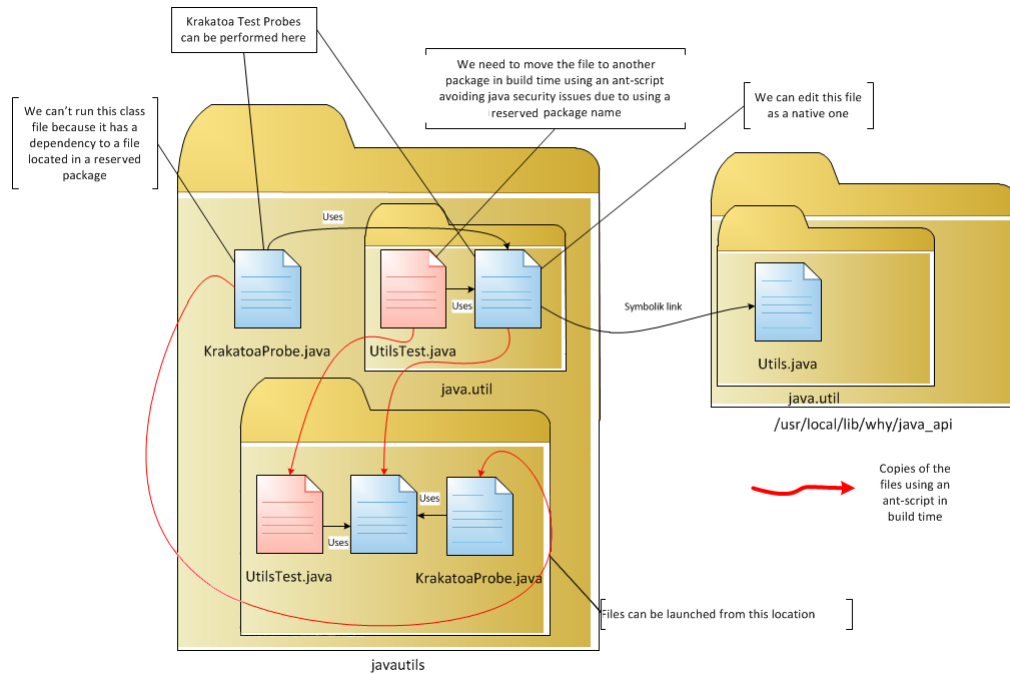


Fig. 10: Verification Platform

<sup>1</sup> An Ant script is an XML build file, containing a single project and a single or multiple targets, each of which consists of a group of tasks that you want Ant to perform. A task is an XML element that Ant can execute to produce a result, for instance, moving class files.

### 4.3 MakeLineRadii

MakeLineRadii function (See Section 3.3) makes use of a variety of functions that need to be formalised; therefore we need to create some basic operations to support more complex operations.

The basic essential operations that we have considered are: Absolute value, Power value, Cast from a Double to an Integer and Integer root of a Double value:

- Abs
  - Axiom1: Abs value of any number  $\geq 0 \Rightarrow$  The number.
  - Axiom2: Abs value of a number  $< 0 \Rightarrow$  - The number.
- Power
  - Axiom1: Any number raised to the power of 0  $\Rightarrow$  1.
  - Axiom2: Any number raised to the power of 1  $\Rightarrow$  The number.
  - Axiom3: Any number raised to the power of 2  $\Rightarrow$  The number x The number.
  - Axiom4:  $\text{Power}(x, i+j) \Rightarrow \text{Power}(x,i) + \text{Power}(x,j)$ .
  - Axiom5:  $\text{Power}(x,i*j) \Rightarrow \text{Power}(\text{Power}(x,i),j)$ .
  - Axiom6:  $\text{Power}(x*y,j) \Rightarrow \text{Power}(x,i) * \text{Power}(y,i)$ .
  - Axiom7:  $\text{Power}(x,i) \Rightarrow x * \text{Power}(x,i-1)$ .
- Cast
  - Axiom1: The cast of a zero double number is zero.
  - Axiom2: The cast of a positive double number is its first integer lower number.
  - Axiom3: The cast of a negative double number is its first integer greater number.
  - Predicate1:  $\text{isGoodCast}(\text{int } x, \text{double } d) \Rightarrow (d==0.0 \Rightarrow x==0) \ \&\& \ (d>0 \Rightarrow x \leq d < x+1) \ \&\& \ (d<0 \Rightarrow x \geq d > x-1)$ .
- Root
  - Axiom1: The root of zero is zero.
  - Axiom2: The root of a positive number is greater or equal than zero.
  - Axiom3: Where  $\text{sqrtd}$  is the integer root of a double positive number  $\Rightarrow (\text{sqrtd}(x) * \text{sqrtd}(x)) \leq x < ((\text{sqrtd}(x)+1) * (\text{sqrtd}(x)+1))$ ;
  - Predicate1:  $\text{isGoodSqrt}(\text{double } x, \text{double } c) \Rightarrow x*x-c < \text{Epsilon}$ ;

## 5 Development

### 5.1 NeuronPersistentJ Plugin

NeuronPersistentJ (as we saw on Section 3.3) had some troubles that made it fail randomly when trying to detect the neuronal structure of the image.

Using Krakatoa to formalise the median value, helped us to know that the problem was not on that part of the code, hence we decided to face the problem using another two different debugging approaches to discover where the problem in NeuronPersistentJ was:

- Launching ImageJ with the argument *-debug* displays a Log Window that helped us to determine what was happening in the program on every moment.
- Downloading the source code of ImageJ gave us the opportunity of setting break points in the code being able to see the values of the program variables and how they were changing while the program was running.

We detected that it was a synchronization problem of the threads that performed the operations on the wrong images and that it could be solved gaining the focus of the image before performing any operation.

Because of this reason, we have defined the following functions that need to be called before performing any operation on the image:

```
private void gainFocus(Thread cu, ImagePlus imp){
    gainFoucs(cu,imp,1);
}

private void gainFoucs(Thread cu, ImagePlus imp, int i) {
    int num = imp.getNSlices();
    if(num >= i){
        imp.setSlice(i);
    }
    WindowManager.setWindow(imp.getWindow());
    WindowManager.setTempCurrentImage(cu,imp);
}
```

With this code, we gain the focus on the selected slice and imagePlus instance in the current Thread. Now we only need to call it before performing any doWand or run action as we can see in the code extracted from NeuronPersistentJ plugin.

```
//First we charge cu variable with the current thread value
Thread cu = Thread.currentThread();

//Now we perform the two operations over the
this.gainFoucs(cu, ImageMax, 1);
IJ.doWand(ImageMax, x[k], y[k], 10, "8-connected");
this.gainFoucs(cu, ImageMax, 1);
IJ.run(ImageMax,"Create Mask", "");
```

Performing this simple change on the code, NeuronPersistentJ is a robust plugin that has exactly the same behaviour each time it is executed.

## 5.2 Verification Platform

**Optimizing Power Axiom** The Axiom7 of the Power Section on 4.3 uses an algorithmic method that requires (power - 1) multiplications to obtain the desired result, but we can replace it for an optimized version that makes use of the power of two axiom:

```

@ /*
@ axiom power_general1:
@   \forall real x , integer i; x > 1 && i%2 == 1
@ ==> lpower(x,i) == lpower(lpower(x,2),i/2)*x;
@
@ axiom power_general2:
@   \forall real x , integer i; x > 1 && i%2 == 0
@ ==> lpower(x,i) == lpower(lpower(x,2),i/2);
@ }
@ */

```

Applying these two axioms to  $3^{45}$  we obtain:

```

3^45 ==> 3^44 * 3 ==> ((3^22)^2) * 3 ==> ((3^11)^2^2) * 3 ==>
((3^10 * 3)^2^2) * 3 ==> ((3^10 * 3)^2^2) * 3 ==> (((3^5)^2 * 3)^2^2) * 3 ==>
((((3^4) * 3)^2 * 3)^2^2) * 3 ==> (((3^2^2) * 3)^2 * 3)^2^2 * 3

```

And instead of 44 multiplications, we only perform 8.

**Setting the Epsilon Value of Sqrt** As you can see on Section 5.3 we replaced the native Math.sqrt function for a custom one that requires an epsilon value that satisfies:

```

/*@
@ predicate is_sqrt(double x, double c) =
@ x*x-c < Epsilon;
@ */

```

We need to find a proper epsilon value similar than the produced by the native Math.sqrt function and to obtain that value we have to obtain random values and save its epsilon value.

After having done that process we get Figure 11

We can see red colored the native Math.sqrt function values and blue colored the custom function values and the epsilon always remains under the value 1,2E-007. The results are displayed in Table 3.

Table 3: Sqrt Epsilon - Values Table

|           | Custom Sqrt Method    | Native Math.sqrt method |
|-----------|-----------------------|-------------------------|
| Max_Error | 1,23E-007             | 1,19E-007               |
| AVG_Error | 1,06447213215943E-008 | 3,65610620705302E-009   |

Analysing the results, we see that the epsilon value in Math.sqrt native function converges to 1.2E-7, hence we have decided to set the epsilon value of our sqrt function to 1.2E-7.

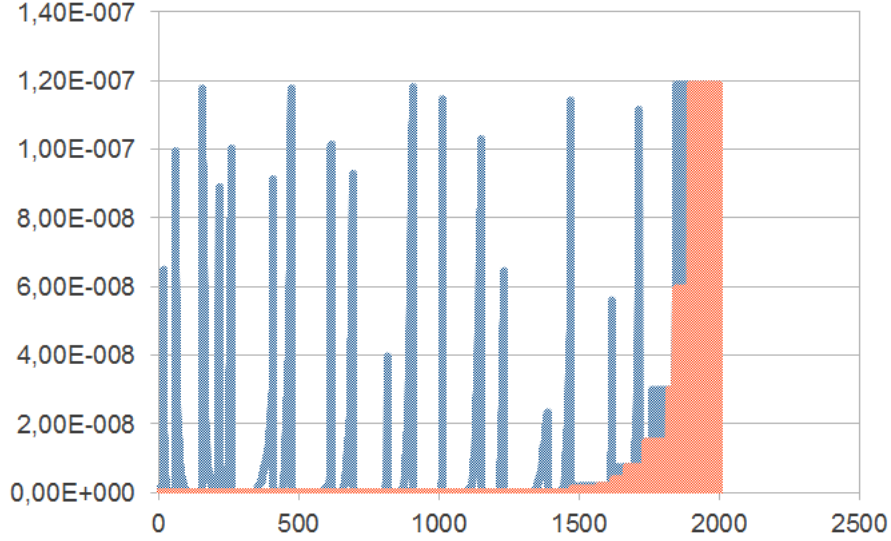


Fig. 11: Sqrt Epsilon value

### 5.3 Specifying programs for digital imaging

As already said in Section 3.1, Fiji and ImageJ are open source projects and many different people from many different teams (some of them not being computer scientists) are involved in the development of the different Fiji Java plug-ins. This implies that the code of these programs is in general not suitable for its formal verification and a deep previous transformation process, following the steps explained in Section 3.2, is necessary before introducing the Java programs into the Why/Krakatoa system. Even after this initial transformation, Fiji programs usually remain complex and their specification in Krakatoa is not a direct process. In this section we present some examples of Fiji methods that we have specified in JML trying to show the difficulties we have faced.

Once that a Fiji Java program has been adapted, following the ideas of Section 3.2, and is accepted by the Why/Krakatoa application, the following step in order to certify its correctness consists in specifying its behaviour (that is, its precondition and its postcondition) by writing annotations in the Java Modelling Language (JML) [8]. The precondition of a method must be a proposition introduced by the keyword **requires** which is supposed to hold in the pre-state, that is, when the method is called. The postcondition is introduced by the keyword **ensures**, and must be satisfied in the post-state, that is, when the method returns normally. The notation **\result** denotes the returned value. To differentiate the value of a variable in the pre- and post- states, we can use the keyword **\old** for the pre-state.

Let us begin by showing a simple example. The following Fiji method, included in the class *Rectangle*, translates an object by given horizontal and vertical increments *dx* and *dy*.

```
/*@ ensures x == \old(x) + dx && y == \old(y) + dy;
   @*/
public void translate(final double dx, final double dy) {
    this.x += dx; this.y += dy;
}
```

The postcondition expresses that the field *x* is modified by incrementing it by *dx*, and the field *y* is increased by *dy*. In this case no precondition is given since all values of *dx* and *dy* are valid, and the keyword *\result* does not appear because the returned type is *void*.

Using this JML specification, Why/Krakatoa generates several lemmas (*Proof Obligations*) which express the correctness of the program. In this simple case, the proof obligations are elementary and they can be easily discharged by the automated theorem provers Alt-Ergo [6] and CVC3 [4], which are included in the environment. The proofs of these lemmas guarantee the correctness of the Fiji method *translate* with respect to the given specification.

Unfortunately, this is not the general situation because, as already said, Fiji code has not been designed for its formal verification and can be very complicated; so, in most cases, Krakatoa is not able to prove the validity of a program from the given precondition and postcondition. In order to formally verify a Fiji method, it is usually necessary to include annotations in the intermediate points of the program. These annotations, introduced by the keyword *assert*, must hold at the corresponding program point. For loop constructs (*while*, *for*, etc), we must give an *inductive invariant*, introduced by the keyword *loop\_invariant*, which is a proposition which must hold at the loop entry and be preserved by any iteration of the loop body. One can also indicate a *loop\_variant*, which must be an expression of type integer, which remains non-negative and decreases at each loop iteration, assuring in this way the termination of the loop. It is also possible to declare new logical functions, lemmas and predicates, and to define *ghost variables* which allow one to monitor the program execution.

Let us consider the following Fiji method included in the class *RankFilters*. It implements Hoare's find algorithm (also known as *quickselect*) for computing the *n*th lowest number in part of an unsorted array, generalizing in this way the computation of the median element. This method appears in the implementation of the "median filter", a process very common in digital imaging which is used in order to achieve greater homogeneity in an image and provide continuity, obtaining in this way a good binarization of the image.

```
/*@ requires buf!=null && 1<= bufLength <= buf.length && 0<=n <bufLength;
   @ ensures Permut{Old,Here}(buf,0,bufLength-1)
   @   && (\forallall integer k; (0<=k<=n-1 ==> buf[k]<=buf[n])
   @   && (n+1<=k<=bufLength-1 ==> buf[k]>=buf[n]))
   @   && \result==buf[n] ;
```

```

    @*/
public final static float findNthLowestNumber
    (float[] buf, int bufLength, int n) {
    int i,j;
    int l=0;
    int m=bufLength-1;
    float med=buf[n];
    float dum ;
    while (l<m) {
        i=l ;
        j=m ;
        do {
            while (buf[i]<med) i++ ;
            while (med<buf[j]) j-- ;
            dum=buf[j];
            buf[j]=buf[i];
            buf[i]=dum;
            i++ ; j-- ;
        } while ((j>=n) && (i<=n)) ;
        if (j<n) l=i ;
        if (n<i) m=j ;
        med=buf[n] ;
    }
    return med ;
}

```

Given an array `buf` and two integers `bufLength` and `n`, the Fiji method `findNthLowestNumber` returns the  $(n + 1)$ -th lowest number in the first `bufLength` components of `buf`. The precondition expresses that `buf` is not null, `bufLength` must be an integer between 1 and the length of `buf`, and `n` is an integer between 0 and `bufLength` – 1. The definition of the postcondition includes the use of the predicate `Permut`, a predefined predicate, which expresses that when the method returns the (modified) `bufLength` first components of the array `buf` must be a permutation of the initial ones. The array has been reordered such that the components  $0, \dots, n - 1$  are smaller than or equal to the component  $n$ , and the elements at positions  $n + 1, \dots, \text{bufLength} - 1$  are greater than or equal to that in  $n$ . The returned value must be equal to `buf[n]`, which is therefore the  $(n + 1)$ -th lowest number in the first `bufLength` components of `buf`.

In order to prove the correctness of this program, we have included different JML annotations in the Java code. First of all, loop invariants must be given for all `while` and `do` structures appearing in the code. Difficulties have been found in order to deduce the adequate properties for invariants which must be strong enough to imply the program (and other loops) postconditions; automated techniques like discovery of loop invariants [20] will be used in the future. We show as an example the loop invariant (and variant) for the exterior while, which is given by the following properties:

```

/*@ loop_invariant

```

```

@ 0<=l<=n+1 && n-1<=m<=bufLength-1 && l<=m+2
@ && (\forall integer k1 k2; (0<=k1<=n && m+1<=k2<=bufLength-1)
@      ==> buf[k1]<=buf[k2])
@ && (\forall integer k1 k2; (0<=k1<=l-1 && n<=k2<=bufLength-1)
@      ==> buf[k1]<=buf[k2])
@ && Permut{Pre,Here}(buf,0,buf.length-1) && med==buf[n]
@ && ((l<m)==> ((l<=n)&&(m>=n)));
@ loop_variant m - l+2;
@*/

```

To help the automated provers to verify the program and prove the generated proof obligations it is also necessary to introduce several assertions in some intermediate points of the program and to use ghost variables which allow the system to deduce that the loop variant decreases.

Our final specification of this method includes 78 lines of JML annotations (for only 24 Java code lines). Krakatoa/Why produces 175 proof obligations expressing the validity of the program. The automated theorem prover Alt-Ergo is able to demonstrate all of them, although in some cases more than a minute (in an ordinary computer) is needed; another prover included in Krakatoa, CVC3, is, on the contrary, only capable of proving 171. The proofs of the lemmas obtained by means of Alt-Ergo certify the correctness of the method with respect to the given specification.

In this particular example, the automated theorem provers integrated in Krakatoa are enough to discharge all the proof obligations. In other cases, some properties are not proven, and then one should try to prove them using *interactive* theorem provers, as Coq. In this architecture, we also introduce the ACL2 theorem prover, as explained in the next section.

#### 5.4 The role of ACL2

In this section, we present the role played by ACL2 in our infrastructure to verify the correctness of Java programs. The Why platform relies on automated provers, such as Alt-Ergo or CVC3, and interactive provers, such as Coq or PVS, to discharge proof obligations; however, it does not consider the ACL2 theorem prover to that aim. We believe that the use of ACL2 can help in the proof verification process. The reason is twofold.

- The scope of automated provers is smaller than the one of ACL2; therefore, ACL2 can prove some of the proof obligations which cannot be discharged by automated provers.
- Moreover, interactive provers lack automation; then, ACL2 can automatically discharge proof obligations which would require user interaction in interactive provers.

We have developed *Coq2ACL2*, a Proof General extension, which integrates ACL2 in our infrastructure to verify Java programs; in particular, we work with



ACL2(r) a variant of ACL2 which supports the real numbers [13] – the formalisation of real analysis in theorem provers is an outstanding topic, see [7]. Coq2ACL2 features three main functions:

- F1.** it transforms Coq statements generated by Why to ACL2;
- F2.** it automatically sends the ACL2 statements to ACL2; and
- F3.** it displays the proof attempt generated by ACL2.

If all the statements are proved in ACL2; then, the verification process is ended. Otherwise, the statements must be manually proved either in Coq or ACL2.

The major challenge in the development of Coq2ACL2 was the transformation of Coq statements to ACL2. There is a considerable number of proposals documented in the literature related to the area of theorem proving interoperability. We have not enough space here to do a thorough review, but we can classify the translations between proof assistants in two groups: *deep* [9, 15, 21] and *shallow* [11, 23, 27].

In our work, we took advantage of a previous shallow development presented in [3], where a framework called *I2EA* to import Isabelle/HOL theories into ACL2 was introduced. The approach followed in [3] can be summarised as follows. Due to the different nature of Isabelle/HOL and ACL2, it is not feasible to replay proofs that have been recorded in Isabelle/HOL within ACL2. Nevertheless, Isabelle/HOL statements dealing with first order expressions can be transformed to ACL2; and then, they can be used as a schema to guide the proof in ACL2.

A key component in the framework presented in [3] was an XML-based specification language called *XLL* (that stands for Xmall Logical Language). XLL was developed to act as an intermediate language to port Isabelle/HOL theories to both ACL2 and an Ecore model (given by UML class definitions and OCL restrictions) – the translation to Ecore serves as a general purpose formal specification of the theory carried out. The transformations among the different languages are done by means of XSLT and some Java programs. We have integrated the Coq system into the I2EA framework as can be seen in Figure 12; in this way, we can reuse both the XLL language and some of the XSLT files developed in [3] to transform (first-order like) Coq statements to ACL2.

In particular, functionality **F1** of Coq2ACL2 can be split into two steps:

1. given a Coq statement, Coq2ACL2 transforms it to an XLL file using a Common Lisp translator program; then,
2. the XLL file is transformed to ACL2 using an XSLT file previously developed in [3].

In this way, ACL2 has been integrated into our environment to verify Java programs. As we will see in the following section, this has meant an improvement to automatically discharge proof obligations.

## 5.5 The method in action: a complete example

In our work, we deal with images acquired by microscopy techniques from biological samples. These samples have volume and the object of interest is not always

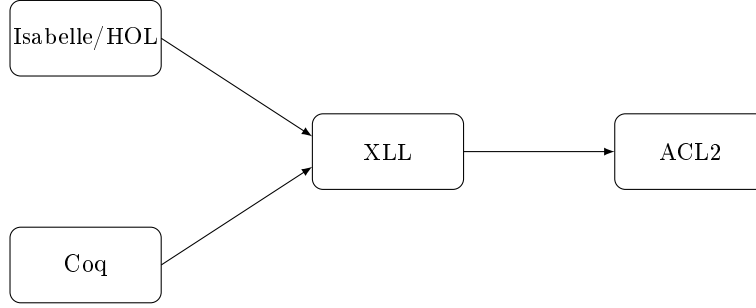


Fig. 12: (Reduced) Architecture of the I2EA framework integrating Coq.

in the same plane. For this reason, it is necessary to obtain different planes from the same sample to get more information. This means that several images are acquired in the same  $XY$  plane at different levels of  $Z$ . To work with this stack of images, it is often necessary to make their *maximum projection*. To this aim, Fiji provides several methods such as maximum intensity or standard deviation to obtain the maximum projection of a set of images.

In this section, we consider the Fiji code for computing the maximum projection of a set of images based on the standard deviation, which uses in particular the method

`calculateStdDev` located in the class *ImageStatistics*.

```

double calculateStdDev(double n, double sum, double sum2) {
    double stdDev = 0.0;
    if (n>0.0) {
        stdDev = (n*sum2-sum*sum)/n;
        if (stdDev>0.0)
            stdDev = Math.sqrt(stdDev/(n-1.0));
        else
            stdDev = 0.0;
    } else
        stdDev = 0.0;
}

```

The inputs are `n` (the number of data to be considered), `sum` (the sum of all considered values; in our case, these values will be obtained from the pixels in an image) and `sum2` (the sum of the squares of the data values). The method `calculateStdDev` computes the standard deviation from these inputs and assigns it to the field `stdDev`. The specification of this method is given by the following JML annotation.

```

/*@ requires ((n==1.0)==> sum2==sum*sum) && ((n<=0.0) || (n>=1.0)) ;
    @ behavior negative_n :
    @   assumes  n<=0.0 || (n>0.0 && (n*sum2-sum*sum)/n <=0.0);

```

```

@ ensures stdDev == 0.0;
@ behavior normal_behavior :
@ assumes n>=1.0 && ((n*sum2-sum*sum)/n > 0.0);
@ ensures is_sqrt(stdDev,(double)((n*sum2-sum*sum)/n/(n-1.0)));
@*/

```

The precondition, introduced by the keyword **requires**, expresses that in the case  $n = 1$  (that is, there is only one element in the data) the inputs **sum** and **sum2** must satisfy  $\text{sum2} = \text{sum} * \text{sum}$ . Moreover we must require that  $n$  is less than or equal to 0 or greater than or equal to 1 to avoid the possible values in the interval  $(0, 1)$ ; for  $n$  in this interval one has  $n - 1 < 0$  and then it is not possible to apply the square root function to the given argument  $\text{stdDev}/(n - 1.0)$ . This fact has not been taken into account by the author of the Fiji program because in all real applications the method will be called with  $n$  being a natural number; however, to formalise the method we must specify this particular situation in the precondition. For the postcondition we distinguish two different behaviours: if  $n$  is non-positive or **sum** and **sum2** are such that  $n * \text{sum2} - \text{sum} * \text{sum} < 0$ , the field **stdDev** is assigned to 0; otherwise, the standard deviation formula is applied and the result is assigned to the field **stdDev**. The predicate **is\_sqrt** is previously defined.

We also consider the code that selects the pixels that are needed to compute the median filter over them and the code that performs the selection is located in the method **makeLineRadii** (See Sections 3.3 and 4.3). The specification of this method in JML annotation is not short enough to be included here, but it can be seen on the attached source code and we are going to explain it anyway.

The precondition has different blocks:

- **firstif1**: It specifies an input argument which is  $1.5 \leq \text{argument} < 1.75$  and an output array equals to  $[0, 0, -1, 1, -2, 2, -1, 1, 0, 0, 13, 2]$  which is the output array of an input equals to 1.75.
- **firstif2**: It specifies an input argument which is  $2.5 \leq \text{radius} \leq 2.85$  and an output array equals to  $[0, 0, -2, 2, -2, 2, -3, 3, -2, 2, -2, 2, 0, 0, 29, 3]$  which is the output array of an input equals to 2.85.
- **normal\_behavior**: It specifies any input argument between 0.5 and 46339 that do not match in the range of **firstif1** or **firstif2** and the corresponding output array that can be obtained using the formulas of the Section 3.3.
- **overflow**: It specifies input arguments that could return Exceptions due to internal operations overflows and it returns a minimum size radius(0.5) output which is equals to:  $[0, 0, -1, 1, 0, 0, 5, 1]$ .

The number 46339 is not an arbitrary number chosen by chance but a number obtained by the maximum radius that can be applied as an input argument without giving an Overflow Exception on the internal program operations. The most important part of the function is the loop which has a JML Annotation:

```

/*@ loop_invariant
@ radius >= 0 &&

```

```

@ 1<=y<=kRadius+1 &&
@ nPoints == suma(y-1,radius)&&
@ kRadius == (sqrt((double) (getIntA((double)
@ ((real)radius*(real)radius+(real)1.0)) + 1e-10))) &&
@ (\forall integer ka; 1<=ka<y ==>
@ kernel[2*(kRadius-ka)] ==
@ 0-sqrt((double) (getIntA((double)
@ ((real)radius*(real)radius+(real)1.0)) - ka*ka + 1e-10))) &&
@ (\forall integer ka; 1<=ka<y ==> kernel[2*(kRadius-ka) + 1 ] ==
@ sqrt((double) (getIntA((double)
@ ((real)radius*(real)radius+(real)1.0)) - ka*ka + 1e-10))) &&
@ (\forall integer ka; 1<=ka<y ==> kernel[2*(kRadius+ka)] ==
@ 0-sqrt((double) (getIntA((double)
@ ((real)radius*(real)radius+(real)1.0)) - ka*ka + 1e-10))) &&
@ (\forall integer ka; 1<=ka<y ==> kernel[2*(kRadius+ka) + 1 ] ==
@ sqrt((double) (getIntA((double)
@ ((real)radius*(real)radius+(real)1.0)) - ka*ka + 1e-10)))
@ );
@ loop_variant (sqrt((double) (getIntA((double)
@ ((real)radius*(real)radius+(real)1.0)) + 1e-10))) -y;
@*/

```

To sum up the behaviour of this code, we ensure for every element between 1 and  $y-1$  which we call  $ka$  that:

$$\begin{aligned}
- \text{kernel}[2*(kRadius - ka)] &== -(int)\sqrt{(int)(radius^2) + 1 - ka^2 + 1e - 10} \\
- \text{kernel}[2*(kRadius - ka) + 1] &== (int)\sqrt{(int)(radius^2) + 1 - ka^2 + 1e - 10} \\
- \text{kernel}[2*(kRadius + ka) + 1] &== (int)\sqrt{(int)(radius^2) + 1 - ka^2 + 1e - 10} \\
- \text{kernel}[2*(kRadius + ka)] &== -(int)\sqrt{(int)(radius^2) + 1 - ka^2 + 1e - 10}
\end{aligned}$$

We can see  $kRadius$  variable which is equals to  $(int)\sqrt{(int)(radius^2) + 1 + 1e - 10}$  and we can also see *suma* function that calculates the pixels that are included on every step of the loop and its code is:

```

/*@
@ logic integer suma(integer paso, double r2);
@ logic integer sumaC(double r2);
@ axiom sum_general:
@ \forall integer paso, double radius;radius>=0 &&
@ 1<=paso<=(sqrt((double) (getIntA((double)
@ ((real)radius*(real)radius+(real)1.0)) + 1e-10)))
@ ==> suma(paso,radius) == @4*sqrt((double) (getIntA((double)
@ ((real)radius*(real)radius+(real)1.0))-paso*paso+1e-10)) +
@ 2 + suma(paso-1,radius);
@ axiom sum_uno:
@ \forall integer paso, double radius;paso == 0 && radius>=0 ==>
@ suma(paso,radius) == 2*(sqrt((double)
@ (getIntA((double) ((real)radius*(real)radius+(real)1.0)) + 1e-10)))+1;
@ axiom sum_total:

```

```

@ \forall double radius; radius >= 0 ==> sumaC(radius) ==
@ suma((sqrtd((double) (getIntA((double)
@ ((real)radius*(real)radius+(real)1.0)) + 1e-10))),radius);
@*/

```

For the proof of correctness of the method `calculateStdDev` in Krakatoa, it is necessary to specify (and verify) the method `sqrt`. The problem here, as already explained in Section 3.2, is that the method `sqrt` of the class *Math* simply calls the equivalent method in the class *StrictMath*, and the code in *StrictMath* of the method `sqrt` is just a native call and might be implemented differently on different Java platforms. In order to give a JML specification of the method `sqrt` is necessary then to rewrite it with our own code. The documentation of *StrictMath* states “*To help ensure portability of Java programs, the definitions of some of the numeric functions in this package require that they produce the same results as certain published algorithms. These algorithms are available from the well-known network library netlib as the package “Freely Distributable Math Library”, fdlibm.*” In the case of the square root, one of these *recommended* algorithms is Newton’s method; based on it, we have implemented and specified in JML the computation of the square root of a given (non-negative) input of type double.

```

/*@ requires c >= 0 && epsi > 0 ;
    @ ensures \result >= 0 && (\result*\result >= c)
    @   && \result*\result - c < epsi ;
    @*/
public double sqrt(double c, double epsi){
    double t;
    if (c > 1) t = c;
    else t = 1.1;
    /*@ loop_invariant
        @ (t >= 0) && (t*t > c) ;
        @*/
    while (t*t - c >= epsi) {
        t = (c/t + t) / 2.0;
    }
    return t;
}

/*@ requires c >= 0 ;
    @ ensures (\result >= 0) && (\result*\result >= c)
    @   && (\result*\result - c < 1.2E-7);
    @*/
public double sqrt(double c){
    double eps = 1.2E-7;
    return sqrt(c, eps);
}

```

The first method computes the square root of a double `x` with a given precision `epsi`; the second one calls the previous method with a precision less than

$1.2E-7$ . Using JUnit, we have run one million tests between  $1E9$  and  $1E-9$  to show that the results of our method `sqrt` have similar precision to those obtained by the *original* method `Math.sqrt`. Here, we applied the “first test, then verify” approach – intensive testing can be really useful to find bugs (and can save us time) before starting the verification process.

For instance, from the given JML specification for the Fiji method `calculateStdDev` and our `sqrt` method, Why/Krakatoa produces 52 proof obligations, 9 of them corresponding to lemmas that we have introduced and which are used in order to prove the correctness of the programs. Alt-Ergo is able to prove 50 of these proof obligations, but two of the lemmas that we have defined remain unsolved. CVC3 on the contrary only proves 44 proof obligations.

The two lemmas that Alt-Ergo (and CVC3) are not able to prove are the following ones:

```
/*@ lemma double_div_pos :
  @ \forallall double x y; x>0 && y > 0 ==> x / y > 0;
  @*/
/*@ lemma double_div_zero :
  @ \forallall double x y; x==0.0 && y > 0 ==> x / y == 0.0;
  @*/
```

In order to discharge these two proof obligations, we can manually prove their associated Coq expressions.

```
Lemma double_div_zero : (forall (x_0_0:R), (forall (y_0:R),
  ((eq x_0_0 (0)%R) /\ (Rgt y_0 (0)%R) -> (eq (Rdiv x_0_0 y_0) (0)%R)))).

Lemma double_div_pos : (forall (x_13:R), (forall (y:R),
  ((Rgt x_13 (0)%R) /\ (Rgt y (0)%R) -> (Rgt (Rdiv x_13 y) (0)%R)))).
```

Both lemmas can be proven in Coq in less than 4 lines, but, of course, it is necessary some experience working with Coq. Therefore, it makes sense to delegate those proofs to ACL2. Coq2ACL2 translates the Coq lemmas to the following ACL2 ones. ACL2 can prove both lemmas without any user interaction (a screenshot of the proof of one of this lemmas in ACL2 is shown in Figure 13).

```
(defthm double_div_zero
  (implies (and (realp x_0_0) (realp y_0) (and (equal x_0_0 0) (> y_0 0)))
    (equal (/ x_0_0 y_0) 0)))

(defthm double_div_pos
  (implies (and (realp x_13) (realp y) (and (> x_13 0) (> y 0)))
    (> (/ x_13 y) 0)))
```

## 5.6 The role of jUnit

In this section, we present the role played by jUnit in our infrastructure to verify the correctness of Java programs. During the Section 5.5 we performed a change

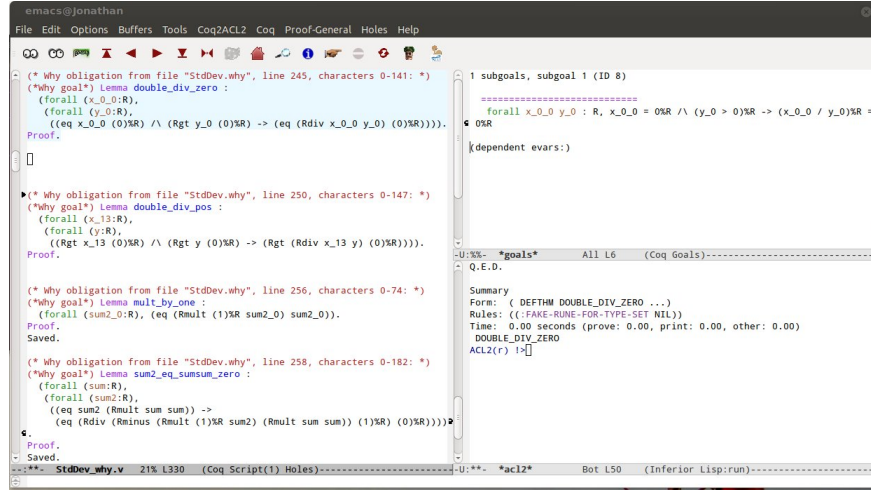


Fig 13: Proof General with Coq2ACL2 extension. The Coq2ACL2 extension consists of the Coq2ACL2 menu and the right-most button of the toolbar. Left: the Coq file generated by the Why tool. Top Right: current state of the Coq proof. Bottom Right: ACL2 proof of the lemma.

in the code and we formalised the methods to ensure the correctness of them, but in addition, we want to ensure that the replaced functions have the same behaviour as the newest ones

For this reason we have developed two tests:

```
@Test
public void comprobarAccion() {
    tester = new Utils();
    for(double i = 0; i < 20000;i = i+0.01){
        assertEquals("Result " + i, (int) i, tester.getInt2(i));
        assertEquals("Result " + (-i), (int) -i, tester.getInt2(-i));
    }
}

@Test
public void comprobarAccion2() {
    double err=1.2E-7;
    //First of all we perform the test with Double_max and Double_min values
    double da1 = Double.MAX_VALUE;
    double ra1= Math.sqrt(da1);
    double da2 = Double.MIN_VALUE;
    double ra= Math.sqrt(da2);
    Assert.assertTrue(ra1*ra1-da1 < err );
    Assert.assertTrue(ra*ra-da2 < err );
    for(int i=0;i<1000000;i++){
        double d = Math.random();
```

```

d = d * tester.power(10, (int)(Math.random()*10));
double ra2= tester.newtonSqrt(d, err);
Assert.assertTrue(ra2*ra2-d < err );
String aux = d + "";
Double da3;
if(aux.indexOf("E-") < 0) {
da3 = Double.parseDouble(aux.replace("E", "E-"));
}
else{
da3 = Double.parseDouble(aux.replace("E-", "E+"));
}
Double ra3= tester.newtonSqrt(da3, err);
Assert.assertTrue(ra3*ra3-da3 < err);
}
}

```

## 6 Final Estimates of Time

This section contains a summary of the estimated and spent hours to conclude the current project (See Table 4)

Table 4: Estimates of time

| Task               | Estimated Hours | Estimated Percentage | Real Hours | Real Percentage |
|--------------------|-----------------|----------------------|------------|-----------------|
| Project Management | 17              | 1.868                | 15         | 1.574           |
| Initiation         | 63              | 6.923                | 71         | 7.45            |
| Training           | 178             | 19.56                | 213        | 22.35           |
| Analysis           | 124             | 13.626               | 104        | 10.913          |
| Design             | 205             | 22.527               | 175        | 18.363          |
| Development        | 198             | 21.758               | 226        | 23.715          |
| Project Report     | 99              | 10.879               | 135        | 14.166          |
| Manuals            | 18              | 1.978                | 4          | 0.42            |
| Defense            | 8               | 0.879                | 10         | 1.049           |
|                    | 910             | 100 %                | 953        | 100 %           |

In the end, the estimates almost reflected reality and we were able to achieve all the goals that we set on Section 2.2. We were only capable of achieving partially the automation of the process that converts Java code into compilable Krakatoa code.



## 7 Conclusions and further work

This dissertation<sup>2</sup> reports an experience to verify actual Java code, as generated by different-skilled programmers, in a multi-programmer tool called Fiji and correct the Fiji Plugin called NeuronPersistentJ.

We have found that the degree of maturity of Krakatoa could be improved although it is a very useful tool to prove isolated and relevant parts of the code using formalisation and several examples in our text show it. In addition to Krakatoa, several theorem provers (Coq and ACL2) have been used to discharge some proof obligations that were not automatically proved by Krakatoa and it has been a teamwork that one person could have never achieved.

In addition we were able to detect and correct an existing problem in Real Java Code (NeuronPersistentJ) used by scientists which is always a highly rewarding task.

Future work includes several improvements in our method. Starting from the beginning, the transformation from real Java code to Krakatoa is partially automated although it needs more efforts to be putted into production. Moreover, a formal study of this transformation could be undertaken to increase the reliability of our method.

We are proud of this project because it was finally accepted in the Conference on Intelligent Computer Mathematics [18] and NeuronPersistentJ plugin has been updated with our suggestions [25].

On the whole, we can say that the formalisation of the code is a very tough process that requires enormous efforts in order to prove a few lines of code, but it should be taken into account in the test of critical software systems.

## References

1. ForMath: Formalisation of Mathematics, European Project. <http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/ForMath>.
2. Java Parser. <https://code.google.com/p/javaparser>.
3. J. Aransay et al. A report on an experiment in porting formal theories from Isabelle/HOL to Ecore and ACL2. Technical report, 2012. [http://wiki.portal.chalmers.se/cse/uploads/ForMath/isabelle\\_acl2\\_report](http://wiki.portal.chalmers.se/cse/uploads/ForMath/isabelle_acl2_report).
4. C. Barrett and C. Tinelli. CVC3. In *19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, pages 298–302, 2007.
5. G. Barthe, D. Pointcheval, and S. Zanella-Béguelin. Verified Security of Redundancy-Free Encryption from Rabin and RSA. In *Proceedings 19th ACM Conference on Computer and Communications Security (CCS'12)*, pages 724–735, 2012.
6. F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The Alt-Ergo automated theorem prover, 2008. <http://alt-ergo.lri.fr/>.
7. S. Boldo, C. Lelay, and G. Melquiond. Formalization of Real Analysis: A Survey of Proof Assistants and Libraries. Technical report, 2013. <http://hal.inria.fr/hal-00806920>.
8. L. Burdy et al. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transf*, 7(3):212–232, 2005.

---

<sup>2</sup> Partially supported by Ministerio de Educación y Ciencia, project MTM2009-13842-C02-01, and by the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

9. M. Codrescu et al. Towards Logical Frameworks in the Heterogeneous Tool Set Hets. In *Post-Proceedings 20th International Workshop on Recent Trends in Algebraic Development Techniques (WADT'10)*, volume 7137 of *LNCS*, pages 139–159, 2012.
10. Coq development team. The Coq Proof Assistant, version 8.4. Technical report, 2012. <http://coq.inria.fr/>.
11. E. Denney. A Prototype Proof Translator from HOL to Coq. In *13th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'00)*, volume 1869 of *LNCS*, pages 108–125, 2000.
12. J. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program verification. In *19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, pages 173–177, 2007.
13. R. Gamboa and M. Kaufmann. Non-Standard Analysis in ACL2. *Journal of Automated Reasoning*, 27(4):323–351, 2001.
14. G. Gonthier et al. A Machine-Checked Proof of the Odd Order Theorem. In *Proceedings 4th Conference on Interactive Theorem Proving (ITP'13)*, *LNCS*, 2013.
15. M. J. C. Gordon, M. Kaufmann, and S. Ray. The Right Tools for the Job: Correctness of Cone of Influence Reduction Proved Using ACL2 and HOL4. *Journal of Automated Reasoning*, 47(1):1–16, 2011.
16. D. Hardin, editor. *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 2010.
17. J. Heras, T. Coquand, A. Mörtberg, and V. Siles. Computing Persistent Homology within Coq/SSReflect. *To appear in ACM Transactions on Computational Logic*, 2013.
18. J. Heras, G. Mata, A. Romero, J. Rubio, and R. Sáenz. Verifying a platform for digital imaging: a multi-tool strategy. In *AISC/MKM/Calculus (CICM'13)*, volume 7691 of *LNCS*, pages 66–81, 2013.
19. J. Heras, M. Poza, and J. Rubio. Verifying an Algorithm Computing Discrete Vector Fields for Digital Imaging. In *AISC/MKM/Calculus (CICM'12)*, volume 7362 of *LNCS*, pages 216–230, 2012.
20. A. Ireland and J. Stark. On the automatic discovery of loop invariants, 1997.
21. M. Jacquél, K. Berkani, D. Delahaye, and C. Dubois. Verifying B Proof Rules Using Deep Embedding and Automated Theorem Proving. In *Proceedings 9th International Conference on Software Engineering and Formal Methods (SEFM'11)*, volume 7041 of *LNCS*, pages 253–268, 2011.
22. M. Kaufmann and J S. Moore. ACL2 version 6.0, 2012. <http://www.cs.utexas.edu/users/moore/acl2/>.
23. C. Keller and B. Werner. Importing HOL Light into Coq. In *Proceedings 1st International Conference on Interactive Theorem Proving (ITP'11)*, volume 6172 of *LNCS*, pages 307–322, 2011.
24. H. Liu and J S. Moore. Java Program Verification via a JVM Deep Embedding in ACL2. In *Proceedings 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'04)*, volume 3223 of *LNCS*, pages 184–200, 2004.
25. G. Mata. NeuronPersistentJ. <http://imagejdocu.tudor.lu/doku.php?id=plugin:utilities:neuronpersistentj:start>.
26. G. Mata. SynapCountJ. <http://imagejdocu.tudor.lu/doku.php?id=plugin:utilities:synapsescountj:start>.
27. S. Obua and S. Skalberg. Importing HOL into Isabelle/HOL. In *3rd International Joint Conference on Automated Reasoning (IJCAR'06)*, volume 4130 of *LNCS*, pages 298–302, 2006.
28. W. S. Rasband. ImageJ: Image Processing and Analysis in Java. Technical report, U. S. National Institutes of Health, Bethesda, Maryland, USA, 1997–2012.

29. J. Schindelin et al. Fiji: an open-source platform for biological-image analysis. *Nature Methods*, 9(7):676–682, 2012.

## A Install and Configure Krakatoa using Eclipse

### Krakatoa Install Manual

Krakatoa is suitable to be installed in Linux or Mac, but during the current document we are describing only Ubuntu Installation.

If you are using another GNU/Linux system distributions, just go to <http://krakatoa.lri.fr/> where you will be able to find different versions of Krakatoa to be compiled. As far as we are concerned, the best version is: <http://why.lri.fr/download/why-2.31.tar.gz>

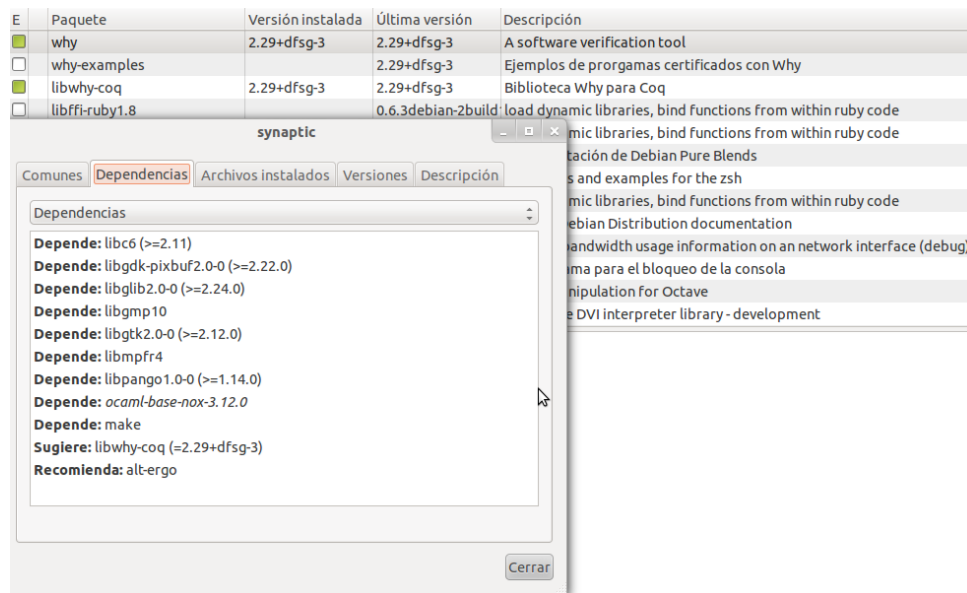
### Krakatoa and Why Installation in Ubuntu

Krakatoa is already included on Ubuntu official Repositories, therefore the easiest way to install it is executing the command:

```
sudo apt-get install why
```

The most easy to install automated-verifiers are alt-ergo y cvc3, In order to install them we execute:

```
sudo apt-get install alt-ergo cvc3
```



If we want to be capable of generating Coq code, we'll need to install the dependency *libwhy-coq*

## Installing and Configuring Eclipse

The first thing we must do is to install Eclipse:

<http://www.eclipse.org/downloads/packages/eclipse-classic-422/junosr2>

We uncompress it wherever we want and we keep it there for a while because we are going to install the JDK.

The process is detailed explained on the following manual:


<http://www.ubuntu-guia.com/2012/04/instalar-oracle-java-7-en-ubuntu-1204.html>

Now we can execute Eclipse, giving it execution permissions:

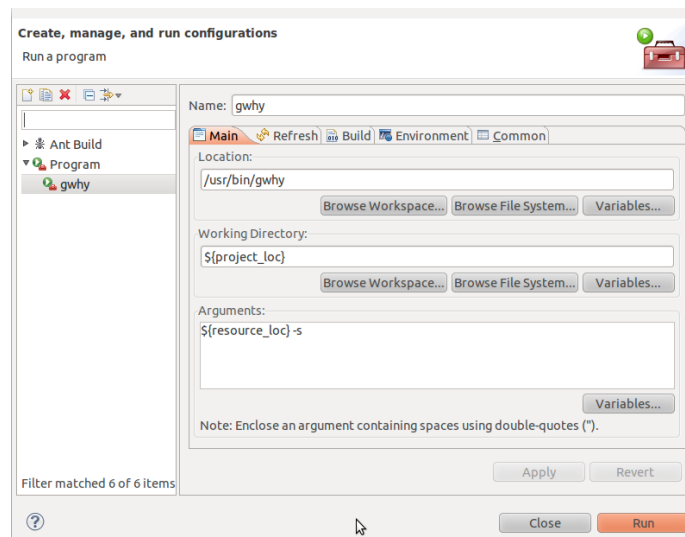
```
chmod +x eclipse
```

And now we can run it by “double-clicking” the corresponding Icon. Once Eclipse is running we must configure it to open Krakatoa, to do that, we perform:

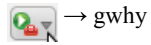
 → External Tools Configuration...

We press the button  to create a new configuration.

And now we fill the fields as follows:



Now we can open a file with Krakatoa just by selecting the file on the package explorer and clicking on:



→ gwhy

A window like this should appear on the screen:

Actividades | gwhy-bin | mar 26 de mar, 12:33 | DMC

gWhy: a verification conditions viewer

File Configuration Proof

| Proof obligations                 | Alt-Ergo<br>0.93 | CVC3<br>2.2<br>(SS) | Statistics |
|-----------------------------------|------------------|---------------------|------------|
| ▼ User goals                      |                  |                     | 2/2        |
| Lemma distinto_siempre            | ✓                | ✓                   |            |
| Lemma raiz_poses                  | ✓                | ✓                   |            |
| ▼ Method abs<br>default behavior  |                  |                     | 2/2        |
| 1. postcondition                  | ✓                | ✓                   |            |
| 2. postcondition                  | ✓                | ✓                   |            |
| ▼ Method absf<br>default behavior |                  |                     | 2/2        |
| 1. postcondition                  | ✓                | ✓                   |            |
| 2. postcondition                  | ✓                | ✓                   |            |

```

rankFilters0 absf ensures default po
num_0: real
H1: true
H3: num_0 < 0.0
return: real
H4: return = -num_0

return = absabstractf(num_0)

```

### ***Configuring Why to generate Coq files and be singleton***

By default we can run more than one instance of Why but it can freeze low performance computers.

If you want we can modify the binary file as follows to kill all the previous generated instances when we run it.

As we can see on the script, we only need to add “-s” argument to the call to be singleton, if we don’t do it, there could be more than one instances of the program.

```
#!/bin/sh
case $1 in
  *.java)
    case "$2" in
      -s)
        curr=`echo $$`
        ant=`echo "$curr" | awk '$0 ~ /^[0-9]/ { print "NOT_NUMBER" }`
        ant=$((curr - 2))
        process_id=`ps aux | grep 'gwhy' | grep -v root | grep -v grep | awk
' {print $2}`
        for line in $process_id; do
          if [ "$line" -le "$ant" ] ; then
            `echo "$line"`
            `kill -9 "$line"`
          fi
        done
      ;;
    esac
    b=`basename $1 .java`
    krakatoa $1 || exit 1
    echo "krakatoa on $b.java done"
    d=`dirname $1`
    echo "cd $d"
    cd $d
    jessie -locs $b.jloc -why-opt -split-user-conj $b.jc || exit 2
    echo "jessie done"
    make -f $b.makefile gui
```

```

    why --lib-file jessie.why --coq-v8 "$d/why/$b.why"
    echo "coq file generated"
    ;;
*.c)
    b=`basename $1 .c`
    caduceus -why-opt -split-user-conj $1 || exit 1
    make -f $b.makefile gui
    ;;
*.jc)
    b=`basename $1 .jc`
    jessie -why-opt -split-user-conj $b.jc || exit 1
    make -f $b.makefile gui
    ;;
*.mlw|*.why)
    gwhy-bin -split-user-conj $1
    ;;
*)
    echo "don't know what to do with $1"
esac

```



## B Krakatoa Operators

| class              | associativity | operators       |
|--------------------|---------------|-----------------|
| selection          | left          | [ ... ] .       |
| unary              | right         | ! ~ + - (cast)  |
| multiplicative     | left          | * / %           |
| additive           | left          | + -             |
| shift              | left          | << >> >>>       |
| comparison         | left          | < <= > >=       |
| comparison         | left          | == !=           |
| bitwise and        | left          | &               |
| bitwise xor        | left          | ^               |
| bitwise or         | left          |                 |
| connective and     | left          | &&              |
| connective xor     | left          | ^^              |
| connective or      | left          |                 |
| connective implies | right         | ==>             |
| connective equiv   | left          | <==>            |
| ternary connective | right         | ... ? ... : ... |
| binding            | left          | \forall \exists |

Krakatoa Operators