Verifying a plaftorm for digital imaging: a multi-tool strategy[⋆]

Jónathan Heras[1], Gadea Mata[2], Ana Romero[2], Julio Rubio[2], and Rubén Sáenz[2]

[1] School of Computing, University of Dundee, UK
[2] Department of Mathematics and Computer Science, University of La Rioja, Spain
jonathanheras@computing.dundee.ac.uk, gadea.mata@unirioja.es, ana.romero@unirioja.es, julio.rubio@unirioja.es,ruben.saenz@unirioja.es

Abstract. Fiji is a Java platform widely used by biologists and other experimental scientists to process digital images. In particular, 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, we have 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. In this paper, we present a *multi-tool* approach filling this gap, based on the combination of Why/Krakatoa, Coq and ACL2.

1 Introduction

Fiji [25] is a Java platform widely used by biologists and other experimental scientists to process digital images. In particular, 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.

The reliability of results is instrumental in biomedical research; therefore, we are working towards the certification of the programs that we use to analyse biomedical images. In a previous work, see [17,16], 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 multi-authors plug-ins, Fiji is messy, very flexible (program pieces are used in some occasions with an objective different from the one they were designed), contains many redundancies and death code, and so on. In summary, it is a big bunch of programs which has not been devised to be formally verified. So, this endeavour is challenging.

In this paper, Krakatoa [11] is used to specify and prove the correctness of Fiji/Java programs. This experience allows us to evaluate both the verification of "real-life" Fiji/Java code, and the Krakatoa tool itself in an unprepared scenario.

⋆ 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).

Krakatoa uses some automated theorem provers (as Alt-Ergo [5] or CVC3 [4]) to discharge the proof obligations generated by means of the Why tool [11]. When some proof obligation cannot be solved by means of the automated provers, the corresponding statements are generated in Coq [7]. Then, the user can try to prove the missing properties interacting with this proof assistant.

In this picture, we add the ACL2 theorem prover [18]. 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 [18]. Instead of integrating ACL2 in the architecture of Why/Krakatoa, we have followed another path. We have reused a proposal to interoperate Isabelle and ACL2 [3]; in particular, we have enhanced our previous tools to translate also from Coq to ACL2, through an XML specification language called XLL [3]. 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 organization of the paper is as follows. The used tools together with our general way of working are briefly presented in Section 2. Section 3 deals with a methodology to "tame" real Fiji code in such a way that it is acceptable for Why/Krakatoa. Section 4 describes an example of the kind of specification we faced. The role of ACL2, and the tools to interoperate between Coq and ACL2, are explained in Section 5. The exposition style along the paper tries to be clear (without much emphasis on formal aspects), driven by real examples extracted from our programming experience in Fiji; in the same vein, Section 6 contains a complete example illustrating the actual role of ACL2 in our setting. The paper ends with a conclusions section and the bibliography.

All the programs and examples presented throughout this paper are available at `http://www.computing.dundee.ac.uk/staff/jheras/vpdims/`.

2 Context, tools, method

2.1 Context

Kenzo [10] is a computer algebra system devoted to Algebraic Topology. Since it is a complex system which has been capable of finding results unknown up to the date [24], a research project was launched to apply formal methods (and, more specifically, verification by means of theorem provers) in the analysis of correctness of Kenzo, and also in the formalization of the algorithms involved. Our team has made contributions using Isabelle/HOL [2], Coq [9] and ACL2 [19].

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 task is to formalize 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 were developed (SynapCountJ [21] and NeuronPersistentJ [20]); 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 [17,16]. 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 Tools

Fiji/ImageJ: Java programming for digital imaging. Fiji [25] is a Java program which can be described as as distribution of ImageJ [23]. 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.

Why/Krakatoa: Specifying and verifying Java code. The Why/Krakatoa tools [11] are an environment for proving the correctness of Java programs annotated with JML [6] specifications. It 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 [7], 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`.

Coq and ACL2: Interactive theorem proving. Coq [7] 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 formalization of relevant mathematical results such as the Four Colour Theorem [13] or the Fundamental Theorem of Algebra [12].

ACL2 [18] 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 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 [15] and for implementing large proofs in mathematics.

2.3 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 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 provide us a sensible trust in our approach; a more detailed description of the transformations needed in step (1) are explained in Section 3. Step (2) is quite well-understood, and some remarks about this step are provided in Section 4. Steps (3)-(6) are mechanized in Krakatoa. The role of ACL2 (steps (6)-(8)) is explained in Section 5 and, by means of an example, in Section 6.

3 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. 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. This is a tough process since we need to add fifteen more classes to make use of the class *Rectangle*.
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. Native methods might be implemented differently on different platforms by Java and 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. For example, the method `sqrt` of the class *Math* which computes the square root of a number of type double simply calls the equivalent method in the class *StrictMath*, and the code in *StrictMath* of the method `sqrt` is just a native call. See in Section 6 our implementation (and specification) of this program based on Newton's algorithm.
7. Add a clause in *if-else* structures for the sake of removing "*Uncaught exception: Invalid_argument("equal: abstract value")*". We can locate 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 overlow 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`.

4 Specifying programs for digital imaging

As already said in Section 2.2, 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, 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, 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) [6] . 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 [5] 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 nth 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)
  @    && (\forall 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`, previously defined, 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 $0, \ldots, n-1$ are smaller than or equal to the component n, and the elements at positions $n+1, \ldots,$ `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. 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 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 integrated ACL2 in our infrastructure to verify Java programs by means of a new Proof General extension called *Coq2ACL2*. 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. Related work on communicating different Interactive Theorem Provers mainly concerned the integration of HOL with other theorem provers, see [14,8,22]. In our work, we took advantage of a previous development presented in [3], where a framework to import Isabelle/HOL theories into ACL2 was introduced. The approach followed in [3] can be summarized 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. Therefore, we can reuse both the XLL language and some of the XSLT files in our work 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;
2. the XLL file is transformed to ACL2 using an XSLT file.

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.

6 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 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 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)) ;
  @ behaviour negative_n :
  @   assumes  n<=0.0 || (n>0.0 && (n*sum2-sum*sum)/n <=0.0);
  @   ensures stdDev == 0.0;
  @ behaviour normal_behaviour :
  @   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 $\texttt{n} = 1$ (that is, there is only one element in the data) the inputs `sum` and `sum2` must satisfy $\texttt{sum2} = \texttt{sum} * \texttt{sum}$. 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 $\texttt{n} - 1 < 0$ and then it is not possible to apply the square root function to the given argument $\texttt{stdDev}/(\texttt{n} - 1.0)$. 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 formalize 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 $\texttt{n} * \texttt{sum2} - \texttt{sum} * \texttt{sum} < 0$, 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.

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, 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`.

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 :
  @ \forall double x y; x>0 && y > 0 ==> x / y > 0;
  @*/
/*@ lemma double_div_zero :
  @ \forall 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 1).

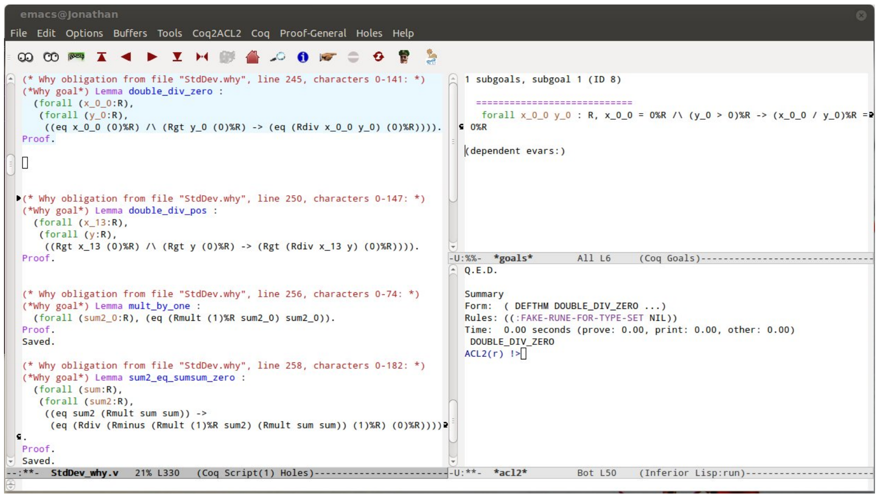
```
(defthm double_div_zero
 (implies (and (realp x_0_0) (realp y_0) (and (equal x_0_0 0) (> y_0 0)))
          (equal (/ x_0_0 y_0) 0)))

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

7 Conclusions and further work

This paper reports an experience to verify actual Java code, as generated by different-skilled programmers, in a multi-programmer tool called Fiji. As one could suspect, the task is challenging and, in some sense, the objectives are impossible of being accomplished, at least in their full extent.

Nevertheless, we defend the interest of this kind of experimental work. It is useful to evaluate the degree of maturity of the verification tools (Krakatoa, in our case). In addition, by a careful examination of the code really needed for a concrete application, it is possible to isolate the relevant parts of the code, and then it is possible to achieve a complete formalization. Several examples in our text showed this feature.

As a further interest of our work, we have reused a previous proposal to interoperability [3], between Isabelle and ACL2, to get an integration of ACL2


emacs@Jonathan
File Edit Options Buffers Tools Coq2ACL2 Coq Proof-General Holes Help

```
(* Why obligation from file "StdDev.why", line 245, characters 0-141: *)
(*Why goal*) 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)))).
Proof.



(* Why obligation from file "StdDev.why", line 250, characters 0-147: *)
(*Why goal*) 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)))).
Proof.


(* Why obligation from file "StdDev.why", line 256, characters 0-74: *)
(*Why goal*) Lemma mult_by_one :
  (forall (sum2_0:R), (eq (Rmult (1)%R sum2_0) sum2_0)).
Proof.
Saved.

(* Why obligation from file "StdDev.why", line 258, characters 0-182: *)
(*Why goal*) Lemma sum2_eq_sumsum_zero :
  (forall (sum:R),
   (forall (sum2:R),
    ((eq sum2 (Rmult sum sum)) ->
     (eq (Rdiv (Rminus (Rmult (1)%R sum2) (Rmult sum sum)) (1)%R) (0)%R))))
.
Proof.
Saved.
--:**-  StdDev_why.v   21% L330   (Coq Script(1) Holes)-----------------------
```

```
1 subgoals, subgoal 1 (ID 8)

  ============================
   forall x_0_0 y_0 : R, x_0_0 = 0%R /\ (y_0 > 0)%R -> (x_0_0 / y_0)%R =
 0%R

(dependent evars:)
-U:%%-  *goals*       All L6     (Coq Goals)-------------------------------
```

```
Q.E.D.

Summary
Form:  ( DEFTHM DOUBLE_DIV_ZERO ...)
Rules: ((:FAKE-RUNE-FOR-TYPE-SET NIL))
Time:  0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
 DOUBLE_DIV_ZERO
ACL2(r) !>
-U:**-  *acl2*       Bot L50    (Inferior Lisp:run)----------------------
```

Fig. 1. 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.

(through a partial mapping from Coq to ACL2), without touching the Krakatoa kernel.

Future work includes several improvements in our method. Starting from the beginning, the transformation from real Java code to Krakatoa one could be automated (Section 3 can be understood as a list of requirements to this aim). Then, a formal study of this transformation could be undertaken to increase the reliability of our method.

As for applications, more verification is needed to obtain a certified version of, for instance, the SynapCountJ plug-in [21]. The preliminary results presented in this paper allow us to be reasonably optimistic with respect to the feasibility of this objective.

References

1. ForMath: Formalisation of Mathematics, European Project. `http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/ForMath`.
2. J. Aransay, C. Ballarin, and J. Rubio. A Mechanized Proof of the Basic Perturbation Lemma. *Journal of Automated Reasoning*, 40(4):271–292, 2008.
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`.
4. C. Barrett and C. Tinelli. CVC3. In *19th International Conference on Computer Aided Verification (CAV 2007)*, volume 4590 of *LNCS*, pages 298–302, 2007.
5. 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/`.

6. L. Burdy et al. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transf*, 7(3):212–232, 2005.
7. Coq development team. The Coq Proof Assistant, version 8.4. Technical report, 2012. `http://coq.inria.fr/`.
8. 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.
9. C. Domínguez and J. Rubio. Effective homology of bicomplexes, formalized in Coq. *Theoretical Computer Science*, 412(11):962–970, 2011.
10. X. Dousson, F. Sergeraert, and Y. Siret. The Kenzo program, 1999. `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/`.
11. J. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program verification. In *19th International Conference on Computer Aided Verification (CAV 2007)*, volume 4590 of *LNCS*, pages 173–177, 2007.
12. H. Geuvers, F. Wiedijk, J. Zwanenburg, R. Pollack, and H. Barendregt. The "Fundamental Theorem of Algebra" Project, 2000. `http://www.cs.ru.nl/~freek/fta/`.
13. G. Gonthier. Formal proof - The Four-Color Theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008.
14. 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*, pages 1–16, 2011.
15. D. Hardin, editor. *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 2010.
16. J. Heras, T. Coquand, A. Mörtberg, and V. Siles. Computing Persistent Homology within Coq/SSReflect. *To appear in ACM Transactions on Computational Logic*, 2013.
17. J. Heras, M. Poza, and J. Rubio. Verifying an Algorithm Computing Discrete Vector Fields for Digital Imaging. In *AISC/MKM/Calculemus (CICM 2012)*, volume 7362 of *LNCS*, pages 216–230, 2012.
18. M. Kaufmann and J S. Moore. ACL2 version 6.0, 2012. `http://www.cs.utexas.edu/users/moore/acl2/`.
19. L. Lambán, F.J. Martín-Mateos, J. Rubio, and J.-L. Ruiz-Reina. Formalization of a normalization theorem in simplicial topology. *Annals of Mathematics and Artificial Intelligence*, 64(11):1–37, 2012.
20. G. Mata. NeuronPersistentJ. `http://imagejdocu.tudor.lu/doku.php?id=plugin:utilities:neuronpersistentj:start`.
21. G. Mata. SynapCountJ. `http://imagejdocu.tudor.lu/doku.php?id=plugin:utilities:synapsescountj:start`.
22. 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.
23. W. S. Rasband. ImageJ: Image Processing and Analysis in Java. Technical report, U. S. National Institutes of Health, Bethesda, Maryland, USA, 1997–2012. `http://imagej.nih.gov/ij/`.
24. A. Romero and J. Rubio. Homotopy groups of suspended classifying spaces: an experimental approach. *To appear in Mathematics of Computation*, 2013. DOI 10.1090/S0025-5718-2013-02680-4.
25. J. Schindelin et al. Fiji: an open-source platform for biological-image analysis. *Nature Methods*, 9(7):676–682, 2012.