# Using Krakatoa for teaching formal verification of Java programs

Jose Divasón and Ana Romero

University of La Rioja, Spain
{jose.divason,ana.romero}@unirioja.es

**Abstract.** In this work, we present a study of different support tools to teach formal verification of Java programs and show our experience with Krakatoa, an automatic theorem prover based on Hoare logic which allows students to interactively visualize the different steps required to prove the correctness of a program, to think about the used reasoning and to understand the importance of verification of algorithms to improve the reliability of our programs.

**Keywords:** Formal methods, Hoare logic, automated theorem provers, JML, Krakatoa.

## 1 Introduction

Formal verification of algorithms is a technique to ensure that a program is *correct* even before being implemented in some programming language, verifying that the program *does what it is supposed to do* for all the possible input data without the necessity of applying testing. Although formal verification is not (yet) a mandatory subject in most computer science studies, it is included in the *Common Criteria for Information Technology Security Evaluation* [1].

In the University of La Rioja, formal verification is taught as part of a compulsory course called "Specification and Development of Software Systems" (SDSS). In the first years of existence of this course (the degree in computer science is taught in our University since 2002), formal verification was considered only in a "theoretical" way, explaining the Hoare logic axioms [12] and presenting the inference rules that make it possible to prove that a program satisfies a specification (given by means of a precondition and a postcondition). Six courses ago we decided to complement this teaching by means of some support tool to formally verify Java programs in a semi-automatic way. To this aim, we did a study of the available software for this task (some of them used only in companies or research) and the chosen support tool was Krakatoa [8]. In the first year of use of this theorem prover, we only used it as support tool during theoretical lessons, showing to students some basic examples. Then, we tried to improve the experience and since 2014 we decided to include in the course some practical lectures in a computer classroom where students could use the tool themselves.

The paper is organized as follows. In Section 2 we present the context of the course SDSS and the teaching of formal verification in our university. Next, in

Section 3 we study several alternatives for teaching formal verification of Java programs, considering different criteria which are interesting for our lectures, and we explain why Krakatoa is the most adequate for our purposes. Some examples of formal verification with this tool are shown in Section 4 as well as the set of exercises that the students must solve and the question on verification in the final exam. We present the results of our experience in Section 5. Finally, conclusions and further work are detailed in Section 6.

## 2 Context of our experience

At the computer science studies at the University of La Rioja, formal verification is taught as part of the course "Specification and Development of Software Systems" (SDSS). SDSS is a compulsory course that is taught in the fourth semester of the degree in computer science, and corresponds to the fourth course of Programming. The course has 6 ECTS, divided into 30 hours of theoretical lectures, 28 hours of practical exercises in a computer laboratory, 2 hours for the final exam and 90 hours of the student individual work. As it is claimed in the guide of the course, one of the aims of SDSS is to provide a formal perspective about different aspects of programming (syntax, semantics, correctness and efficiency), trying to improve the programming skills of students. After considering some subjects such as specification and implementation of abstract data types and their relation with object-oriented programming and specification of algorithms, the final part of the course (about 15 hours) is devoted to formal verification of algorithms. The course SDSS is also part of the degree in mathematics; the students of both degrees attend together the lectures and the contents and the evaluation system for all of them are the same. The students are supposed to have followed the three previous courses on Programming. Moreover, they are supposed to have acquired the fundamental concepts of first order logic which are taught in the second semester of the degrees (as part of the course "Logic"). Each year there are around 70 students, of which about two thirds are students of computer science the rest of mathematics. With respect to the evaluation, 70% of the mark corresponds to the final exam and 30% to laboratory exercises.

One of the goals of SDSS is to consolidate the acquired knowledge in the third semester of both degrees, where it is introduced the concept of object-oriented programming in Java. For this reason, Java is the chosen programming language for the SDSS course. It is worth noting that at the beginning of the course, the students lack a strong experience in programming languages. This, in conjunction with the few hours available to the formal verification part, cause that the introduction from scratch of another different programming language could be a counter-productive decision.

Until 2013, formal verification was taught in SDSS only in a "theoretical" way by means of the Hoare logic axioms [12]. Given a precondition $Q$ and a postcondition $R$, a program "s" (consisting of a sequence of elementary instructions $s \equiv \{s_1, \ldots, s_n\}$) satisfies the specification $\{Q\}s\{R\}$ if: whenever the program $s$ is executed starting in a state which satisfies $Q$, the program

terminates and the final state satisfies $R$. In order to verify the correctness of $\{Q\}s\{R\}$, one must consider predicates which determine the states which are satisfied at the intermediate points of the program, called *assertions*, such that $\{Q\}s_1\{P_1\}s_2\{P_2\}\ldots\{P_{n-1}\}s_n\{R\}$. If the initial assertion $Q$ (precondition) is satisfied, and each elementary "program" $s_k$, consisting of one simple instruction, satisfies the specification $\{P_{k-1}\}s_k\{P_k\}$, then when the program stops the postcondition $R$ is satisfied and therefore the program is correct. In our course, we do not deal with *partial correctness* but with *total correctness*, i.e., a program is correct when it returns the expected result and the algorithm terminates.

Hoare logic provides rules to verify the correctness of the elementary instructions of a programming language (assignments, sequential composition, conditional clauses and iterative composition). These rules allow one to compute in a straightforward way correct preconditions, from a given postcondition, for the cases of assignments, sequential compositions and conditional clauses. However, in the case of the iterative composition the process is not direct and it is necessary to construct first an *invariant* predicate $P$ and a *variant* $V$. Then, Hoare logic requires that the loop body decreases the variant (to ensure termination) while maintaining the invariant. In addition, the invariant must be strong enough so that at the end of the loop we could deduce the postcondition. Usually, the students find it difficult to figure out the invariant.

In SDSS we present (in a theoretical way) the Hoare rules for the basic instructions of an iterative language and we do small examples of application of each one of the rules. Once all the rules have been introduced, we do some exercises of formal verification proofs of some small programs with an iterative scheme. The proofs of correctness considered in the course SDSS are *restricted* to programs corresponding to the following sketch:

```
{Q}
<init>
while B do {
     <body>
}
<end>
return <var>
{R}
```

where the blocks `<init>`, `<body>` and `<end>` consist of a sequence of elementary instructions, usually assignments and conditional structures. In fact this is not a restriction, because if there are several "sibling" loops it can be thought that all but the last one are inside `<init>`, and if there are nested loops one can think that the internal loops are inside `<body>`.

Taking into account Hoare's axioms, in order to verify the correctness of a program with the previous sketch it is necessary to:

1. Find an invariant $P$ for the loop.
2. Verify the specification $\{Q\}$`<init>`$\{P\}$.
3. Verify that $P$ is an invariant, that is to say, the specification $\{P \text{ and } B\}$ `<body>` $\{P\}$ is satisfied.

4. Verify the specification $\{P$ and not$(B)\}$<end>$\{R\}$.
5. Find a variant.

Following these steps, in SDSS we consider proofs of correctness of some easy algorithms such as the (iterative) computation of the power of a real number raised to a natural number, the computation of the factorial of an integer, the integer square root, the sequential search of an element in an array and the sum of all components of an array. After explaining some of these exercises on the blackboard, we do also some exercise classes where the students must apply their knowledge in a practical way and make some correctness proofs on their own. The difficult part of the exercises is the determination of the invariant $P$, and it is very frequent that students propose invariants that are not strong enough and they have to make different attempts (and repeat steps 2, 3 and 4 for all of them) in order to find the correct one.

This "traditional" way of teaching formal verification was the chosen one at our University until 2013. At that moment, we decided to complement the theoretical lectures with the help of some support automatic tool for formal verification of Java programs based on Hoare logic. A study of the different alternatives was done, and the chosen tool was Krakatoa, an automatic theorem prover which allows students to interactively visualize the various steps required to prove the correctness of a Java program, to think about the used reasoning and to understand the importance of verification of algorithms to improve the reliability of our programs. In the first year of use of this theorem prover, we only used it as support tool during theoretical lessons, showing to students some basic examples of formal proofs with Krakatoa. After the positive results of that initial attempt (the marks in the final exam of the formal verification part were higher than previous years and students showed interest in Krakatoa), we tried to improve the experience and since 2014 we decided to include in the course some practical lectures in a computer classroom where students could use the tool themselves, providing the correct specification and the necessary code to verify the proposed programs as explained in Section 4.

## 3 Study of different alternatives

Our study of different alternatives for teaching formal verification of Java programs started in 2013, when we decided to complement our theoretical lessons with some support tool. At that moment we found some documentation of universities where formal verification was taught in a practical way (see for example [7, 14, 15]), but most of them did not correspond to Java programs or did not seem to be based on Hoare logic.

There are several approaches and levels to carry out formal verification of programs. Essentially, tools for this task are classified in three groups. Interactive theorem provers, such as Isabelle [18] and Coq [6], belong to this kind of tools. They allow a mathematical modeling and verification of programs at the highest level of confidence (Common Criteria certification at level EAL7). They have

been used in industrial applications, such as the verification of seL4, an operating system kernel [13]. Nevertheless, these tools need a steep learning curve to gain enough expertise to be able to prove formally specifications of programs, so they seemed not to be a good choice for a first introduction course on formal verification. Secondly, there are tools based on model checking, such as Java Pathfinder [11]. This kind of tools are supposed to be a rigorous method to find a violation of a given specification, not only by means of tests but with abstract interpretations. Finally, there exist tools based on Hoare logic or similar logics, which are the ones we were mainly interested in due to their relation with the theoretical part of the course. In addition, some such tools are indeed focused on teaching (but not in Java), such as Dafny [14] and HAHA [17]. Due to the context of SDSS, we aim to use a tool devoted to verify Java programs. Then, we selected the following tools for evaluation.

– Krakatoa [8] (version 2.41, May 2018)
– KeY [2] (version 2.6.3, October 2017)
– OpenJML [5] (version 0.8.40, October 2018)

Those ones seem to be the most important ones, although there exist more alternatives that could also have been considered for this study such as Jahob [4] and Jack [3]. However, most of them are no longer developed.

In the concrete context of our SDSS laboratory lessons, we evaluated the following features of the tools: ease of use (taking into account that the program will be used by students with no previous knowledge on it), feedback (the information about the proof attempts and proof failures should be understandable for students with no expertise in the tool), documentation (the evaluated tool should have enough examples of different levels of difficulty), relation between the tool and the contents that are taught in the theoretical lessons (this is the most important feature for us: we want to check if the tool clearly follows the steps from Hoare logic), ease of installation and if there exist plugins for Eclipse (the IDE used in our laboratories) or an online tool. Table 1 shows a summary of this evaluation. Apart from that, we also checked the tools against seven exercises that we teach in SDSS in a theoretical way:

1. Minimum of two integers
2. Swap two elements of an integer array
3. Square root (linear version)
4. Square root (binary version)
5. Check if an integer array is sorted in ascending order
6. Exponentiation
7. Linear search of an element in an integer array

We checked if the language of each tool is expressive enough to specify the algorithms, and also whether the evaluated tools are able to prove them strictly by means of just the specification (precondition and postcondition), together with the corresponding invariants and a measure that decreases in each step if necessary. The result is shown in Table 2. It is worth remarking that most of

the programs are able to verify automatically the exercises once the user has provided some hints (or working a bit with the corresponding goals), but we wanted to test them exactly with the same reasoning that we would make in the theoretical lessons: that is, just making use of the specification, invariants and variants (something that decreases in each step).

KeY is the most powerful tool, from the ones that we have studied, and it is also the most used one. It is worth noting that there are several universities where KeY is used as a tool for teaching formal verification of Java programs such as Chalmers University[1] or the Karlsruhe Institute of Technology [2], but within the computer science master's programme. In our opinion, KeY requires a longer learning step than Krakatoa and it is to be used by experts, or at least, it is not designed to be used by degree students in the forth semester. In addition, KeY was not able to automatically prove the correctness of our examples (just from the specifications, invariants and variants), but Krakatoa had a higher success rate. To sum up, despite of the fact that Krakatoa is not the most powerful one, it fits our requirements. Thus, we decided to put it into practice in our laboratory lessons. The Krakatoa program was then available in the computer laboratories of our university as an Eclipse plugin. Indeed, it is a virtualized application, i.e., the students can use it at home easily. This solves the two main drawbacks which Krakatoa presents in our study: the lack of an online tool and the difficulty of its installation. The confusing feedback provided by Krakatoa is solved with the help of the teachers in the laboratory lessons.

It is worth noting that the study of these tools was repeated every year from 2013 (we present here the one that we did in January 2019). The performance of KeY with some exercises has improved in the last years but the results of all studies were similar.

Table 1: Main features of the evaluated tools

| | **Tool** | | |
|---|---|---|---|
| | **Krakatoa** | **KeY** | **OpenJML** |
| Ease of use | ✓ | ✗ | ✓ |
| Feedback | Lack of information | Need a deep knowledge | ✓ |
| Related to theory | ✓ | ✓ | ✗ |
| Documentation | Few examples | ✓ | Under development |
| Ease installation | ✗ | ✓ | ✓ |
| Plugin Eclipse | ✓ | ✓ | ✓ |
| Online tool | ✗ | ✓ | ✓ |

[1] http://www.cse.chalmers.se/edu/year/2018/course/TDA294_Formal_Methods_for_Software_Development/

[2] https://formal.iti.kit.edu/teaching/FormSys2SoSe2017/

Table 2: Expressiveness and solvable problems by the tools

| | Tool | | | | | |
|---|---|---|---|---|---|---|
| | **Krakatoa** | | **KeY** | | **OpenJML** | |
| | Specif. | Solv. | Specif. | Solv. | Specif. | Solv. |
| Minimum | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Swap two elements | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Linear sqrt | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |
| Binary sqrt | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ |
| Sorted array | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Exponential | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |
| Linear search | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ |

# 4 Some examples of formal verification with Krakatoa

Trying to complement the theoretical teaching of formal verification by means of some software, and after the study of alternatives explained in Section 3, the chosen support tool has been Krakatoa. As already said in Section 2, in an initial experience we used it only as a support tool during theoretical lessons but since 2014 we decided to include in the course some practical lectures in a computer classroom where students could use the tool. More concretely, we have now 3 practical lectures (each of them of 2 hours) for 3 different levels of exercises. The first lecture is devoted to the specification and verification of Java methods where only assignments and conditional clauses are used; in this case, if the specification given by the student is valid, Krakatoa should be able to verify directly that the program is correct. On the contrary, if iterative structures are included, Krakatoa needs some *help* and the student must write the invariant predicate for the loops; the second lecture is devoted to this kind of exercises. Finally, it is also sometimes necessary to introduce auxiliary *predicates*, *axiomatic definitions* or *assertions*, which are explained in the third lecture. Since formal verification is only part of the course contents, we teach it in an introductory way and we do not have time to teach formal verification of object oriented aspects such as classes, inheritance or dynamic types.

In this section, we present some examples that we show in the lectures, the mandatory exercises that the students must solve in the computer laboratory and the verification exercises of the final exam.

## 4.1 Lectures in the computer classroom

As we have already said, we have now 3 practical lectures for 3 different levels of exercises. We present here one example of the exercises explained in each one of the lectures. Other examples of exercises of formal verification of Java programs explained in SDSS can be found in [16].

In order to verify the correctness of a Java program, Krakatoa inputs the specification (precondition and postcondition) written in the Java Modeling Language [10] (JML). Then, making use of a tool called Why [9], it generates a series

Fig. 1: Obligations generated by Krakatoa for the method min.

of lemmas (called *proof obligations*) that correspond to the different steps, following Hoare logic, to verify the correctness of the program. These lemmas must be verified by some automatic theorem provers which are included in the Krakatoa tool; if these theorem provers do not reach some of the proofs, it is also possible to send the lemmas to Isabelle and Coq, two interactive theorem provers where the user can help the prover to construct the proofs.

**Minimum of 2 elements** One of the simplest examples explained in the first Krakatoa practical lesson is the following Java method for computing the minimum of two integers $x$ and $y$:

```
/*@ ensures \result <= x && \result <= y &&
  @ ((\result == x) || (\result == y));
  @*/
public static int min(int x, int y) {
    if (x<y) return x; else return y;
}
```

The specification of the method, in JML, is written as a comment between `/*@` and `@*/`. The clause `ensures` is used to introduce the postcondition, which is a logical predicate which must be satisfied when the method stops for any possible value of the inputs. Inside the postcondition, `result` is used to denote the returned value. In this case, the postcondition means: the result is smaller than or equal to $x$, the result is smaller than or equal to $y$, and the result is equal to $x$ or equal to $y$.

The goal of Krakatoa consists of verifying that the method `min` is implemented in a correct way, that is to say, it satisfies the given specification. As shown in Figure 1, Krakatoa generates 6 lemmas (proof obligations) that express the correctness of the program. The 6 obligations correspond to each one of the 3 components of the postcondition, which must be satisfied by each one of the two branches of the conditional clause. These obligations are the steps that the students should do to formally verify (in a theoretical way) that the program is correct. The first lemma, which is detailed on the right side of Figure 1, says that, the result is less than or equal to $x$. In this example the lemmas are

very easy and the automatic theorem provers Alt-Ergo[3] and CVC3[4], which are integrated in Krakatoa, are able to verify them in a direct way. The proof of the 6 obligations imply that the program is correct with respect to the given specification, which ensures that in any possible situation, that is to say, for any of the infinite possible input data, the method returns the desired result.

**Deciding if an array is sorted** The correctness proof of a program is more complicated when it includes iterative instructions. Let us consider now the following method to decide if the elements of an array of integers are sorted (in ascending order):

```
/*@ requires v != null && 1 <= v.length ;
  @ ensures \result <==> (\forall integer j; (0 <=j < v.length-1) ==>
  @ v[j]<=v[j+1]);
  @*/
static boolean isSorted(int v[]) {
    int i=0; boolean b=true;
    while (i<v.length-1 && b) {
        if (v[i] > v[i+1]) b=false;
        i=i+1;
    }
    return b;
}
```

The clause `requires` introduces the precondition, which is a logic predicate that must be satisfied when the method is called. In this case, the argument `v` must be a non-null array with positive length. Krakatoa generates now 2 obligations corresponding to the postcondition but, as one can observe in Figure 2, it is not able to prove them. From the 8 obligations which ensure that the method is safe, it only proves 5 of them.

In order to verify the correctness of a Java program with iterative instructions following the axioms of Hoare logic, as we have explained in Section 2, it is necessary to define an invariant $P$ which is a predicate that is satisfied at the beginning and end of each execution of the loop. This invariant must be strong enough so that when the loop finalizes the postcondition is satisfied. In general it is a difficult problem to find the adequate invariant.

In order to introduce the invariant predicate in the JML specification of a program in Krakatoa one uses the clause `loop invariant`. Moreover, to be able to verify that the loop stops (and therefore the method is safe), very frequently we must define in Krakatoa the variant, which must be an integer expression such that it is non negative and it decreases after each execution, denoted by `loop variant`. For the iterative structure inside the method `isSorted` we can use the following specification:

```
static boolean isSorted(int v[]) {
    int i=0; boolean b=true;
```

[3] Alt-Ergo. http://alt-ergo.lri.fr/

[4] CVC3. http://www.cs.nyu.edu/acsys/cvc3/

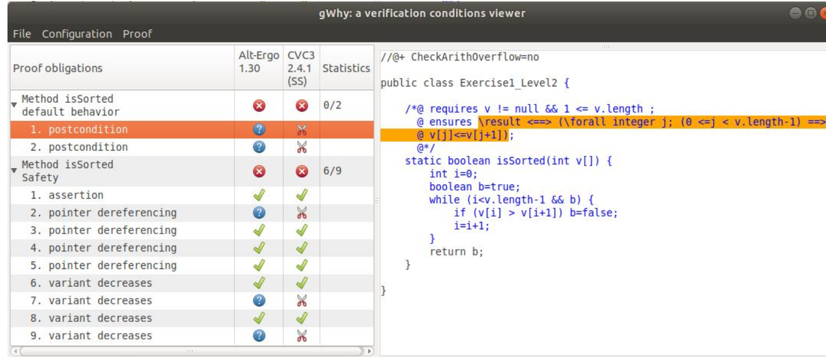


Fig. 2: Obligations generated by Krakatoa for the method isSorted without specifying the invariant predicate.

```
    /*@ loop_invariant 0<=i && i<v.length && (b == true <==>
      @ (\forall integer j; (0 <=j < i) ==> v[j]<=v[j+1]));
      @ loop_variant v.length-i;
      @*/
    while (i<v.length-1 && b) {
        if (v[i] > v[i+1]) b=false;
        i=i+1;
    }
    return b;
}
```

Krakatoa generates now 21 obligations, some of them have appeared when the invariant has been introduced. The proof of such obligations will show the soundness of the algorithm. We can also observe than the generated obligations correspond to steps of the *theoretical* proof of the correctness of the program explained in Section 2. With the help of this invariant the Alt-Ergo and CVC3 theorem provers are able to verify the correctness of the program, as shown in Figure 3.

**Exponential function** In some situations, the definition of the invariant predicate and the *variant* is not enough to prove the correctness of a program with iterative structures and it is also necessary to include auxiliary predicates, axiomatic definitions and *assertions* which help the theorem provers to verify the lemmas generated by Krakatoa.

The following method implements the exponential function raising a float to an integer:

```
public static float exponential (float x, int n) {
    int i=0; float r=1;
    while (i<n) {
        i++;
        r=r*x;
```

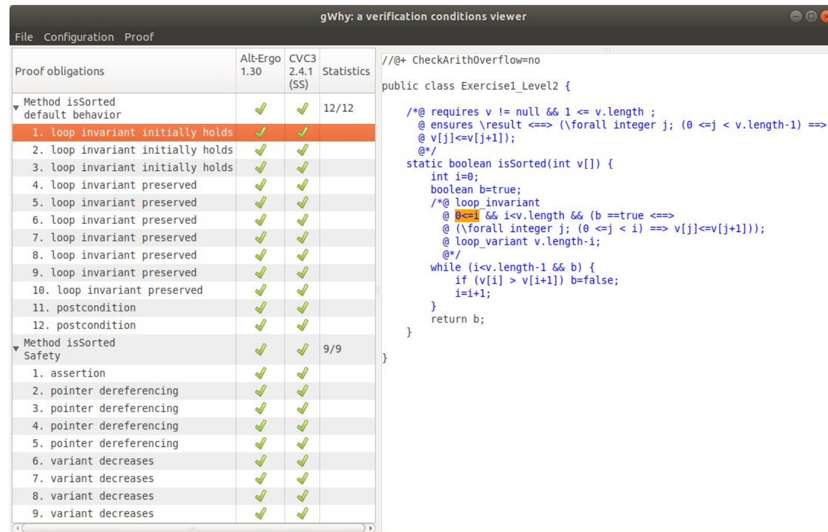


Fig. 3: Obligations generated by Krakatoa for the method isSorted after specifying the invariant predicate.

```
    }
    return r;
}
```

After introducing the idea of the method, the students must think of a possible specification written in JML. Since the JML version supported by Krakatoa does not allow to use the exponential function, in order to specify the method it is necessary to include the following axiomatic definition:

```
/*@ axiomatic Exponential {
  @ logic float exp(float x, integer n);
  @ axiom exp_zero : \forall float x; exp(x,0) ==1;
  @ axiom exp_sum: \forall integer n; \forall float x;
  @ exp(x,n+1) == exp(x,n)*x;
  @}
@*/
```

Using this axiomatic definition, the students should write, using the JML syntax, the specification of the method `exponential`:

```
/*@ requires n >=0;
  @ ensures \result == exp(x,n);
  @*/
```

With this specification, Krakatoa generates 3 obligations but it is able to prove only 1 of them. As we have already said, it is necessary to define the invariant predicate $P$ for the iterative structure. The students should propose an invariant, run Krakatoa and see if the obligations are proved. A possible solution for the invariant (and variant) of the program is:

```
/*@ loop_invariant 0 <= i && i <= n && r == exp(x,i);
  @ loop_variant n-i;
  @*/
```

### 4.2 Exercises in the computer classroom

Once Hoare logic, the steps for verifying programs and the previous examples are explained in the lectures, the students must practice and complete by themselves some exercises with Krakatoa. To this aim, the students work in pairs. They have 6 hours at the computer laboratories with the help of the teacher and 2 days of work at home before the deadline to send their solutions via GitHub classroom. This set of exercises consists of two parts and corresponds to a 5% in the final mark of the course. The first part is mandatory and has 8 exercises devoted to design and verify the following Java programs:

1. A method to compute the absolute value of an integer
2. Check if the arithmetic mean of three non-negative real numbers is higher or equal to 5
3. A method to compute the maximum of three integer numbers
4. Given an array with 4 real numbers, modify it by dividing each component by the sum of all components (with no loops)
5. Decide if a number is prime
6. Check if all elements in an integer array are non-negative
7. Compute the highest factor of a positive integer number (excluding itself)
8. A method to compute the factorial of a non-negative integer number

The second part comprises three voluntary exercises: modification of each component of an integer array by its absolute value, find the frequency of a number in an array and finally design and verify other algorithms with loops or that use some of the previous exercises.

In the course 2019, 61 students (from 68) did the set of Krakatoa exercises. As in previous years, they had very good marks: the mean was 0.457 (over 0.5). Concretely, the students of computer science had a mean of 0.447 and the ones of mathematics 0.467. Table 3 shows the number of students with wrong answers in each mandatory exercise (all students did all of them). As it can be seen, the students have problems with exercise 7, which corresponds to the one with the most difficult invariant. With respect to the optional exercises, 80% of the students did the first one (from which, 90% did it well). Only 26.9% of the students sent the second voluntary exercise (all of the received answers were correct). One student did the third one.

### 4.3 The exam

The final exam consists of three written exercises of the different aspects covered by the course. The exercises are solved without the help of the computer. One of them is about formal verification. It is the most important one: approximately 45% of the mark in the final exam corresponds to this exercise.

In the last course (2019), this exercise consists of two parts:

Table 3: Results of mandatory exercises in the computer classroom. Number of students who did the exercises N = 61.

| **Exercise** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| **Wrong answers** | 2 | 1 | 4 | 3 | 2 | 2 | 10 | 4 |

1. Prove that the predicate $P = (1 \leq i \leq n)$ and $(n \,\%\, m = 0)$ and $(\forall \alpha \in \{m+1, \ldots, i-1\}.(n \,\%\, \alpha \neq 0))$ is an invariant for the following loop:

```
while (i < n) {
    if (n % i == 0) m=i;
    i++;
}
```

2. Verify the correctness of an algorithm computing the mean of the values of the elements of an array.

In general, the marks in this exercise were high (7.35 over 10 last year, the higher the better). During these years, we noticed a better understanding of the concepts among the students of the degree in mathematics, since they are more used to abstract reasoning and formal proofs. Indeed, the students of that degree outperform the students of the degree in computer science. This can be shown, for instance, in the marks of the verification exercise of the last exam: the mean of the marks of the students of computer science was 6.29, whereas the mean increased to 8.19 for the students of the degree in mathematics.

## 5 Results of the experience

The results of using Krakatoa as a support tool for teaching formal verification of Java programs have been very positive. First of all, we have observed that after using Krakatoa the students understand the different steps of the (theoretical) formal proofs in a better way; more concretely, when Krakatoa was not used as a support tool many of the students *memorized* the exercises of formal verification (and very frequently they did not really understand them). This better understanding of students has been shown in the marks on average of the formal verification exercises in the final exam that have increased significantly (see Table 4, exercises are marked with a number between 0 and 10, the higher the better). The first year of use of Krakatoa just as a support tool (2013), the average of the marks in the final exam of the formal verification part was higher than previous years. The difficulty was very similar. During the following courses the marks remained higher than in 2012, although deeper contents and higher difficulty of the exercises were demanded in the exams. Moreover, many students claim now that this is the most interesting part of the course. Indeed, two students decided to carry out their final-degree project on this subject. Our experience as teachers is also positive and we plan to continue using Krakatoa in the following courses.

Table 4: Marks on average of formal verification exercises in the final exam.

| Year | Students | Marks |
|---|---|---|
| 2012 | 46 | 6.14 |
| 2013 | 50 | 7.50 |
| 2014 | 37 | 7.06 |
| 2015 | 38 | 7.17 |
| 2016 | 54 | 6.81 |
| 2017 | 54 | 7.23 |
| 2018 | 73 | 7.86 |
| 2019 | 66 | 7.35 |

## 6 Conclusions and further work

In this work we have presented our experience with the tool Krakatoa to teach formal verification of Java programs, improving in this way the theoretical lessons on Hoare logic and helping students to understand the different steps of formal verification. With this experience, the average marks of formal verification exercises in the final exam has been increased; moreover, students show interest in this part of the course.

After these positive results, we plan to continue using Krakatoa in the following courses in laboratory sessions, considering other exercises with similar difficulty to the ones presented in this work. We will also repeat the study of other possible tools presented in Section 3, and we think that KeY could be also a good candidate in the future.

## Acknowledgments

Partially supported by the Spanish Ministry of Science, Innovation and Universities, project MTM2017-88804-P.

## References

1. Common Criteria for Information Technology Security Evaluation. Tech. rep. (2012)
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016)
3. Barthe, G., Burdy, L., Charles, J., Grégoire, B., Huisman, M., Lanet, J.L., Pavlova, M., Requet, A.: JACK—A Tool for Validation of Security and Behaviour of Java Applications. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects. pp. 152–174. Springer (2007)
4. Bouillaguet, C., Kuncak, V., Wies, T., Zee, K., Rinard, M.: Using First-Order Theorem Provers in the Jahob Data Structure Verification System. In: Cook, B., Podelski, A. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 74–88. Springer (2007)

5. Cok, D.R.: OpenJML: JML for Java 7 by Extending OpenJDK. In: Proceedings of the Third International Conference on NASA Formal Methods. pp. 472–479. NFM'11 (2011)
6. Coq development team: The Coq Proof Assistant, version 8.9.1. Tech. rep. (2019), `https://coq.inria.fr/`
7. Feinerer, I., Salzer, G.: Automated tools for teaching formal software verification. In: Proceedings of the 2006 Conference on Teaching Formal Methods: Practice and Experience. pp. 4–4 (2006)
8. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: Proceedings of the 19th International Conference on Computer Aided Verification. pp. 173–177. CAV'07 (2007)
9. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Proceedings of the 22nd European Symposium on Programming. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013)
10. Gary T. Leavens, A.L.B., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. Tech. rep. (2000), iowa State University
11. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. International Journal on Software Tools for Technology Transfer **2**(4), 366–381 (2000)
12. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM pp. 576–580 (1969)
13. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al.: seL4: Formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. pp. 207–220. ACM (2009)
14. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. pp. 348–370. LPAR'10, Springer (2010)
15. Poll, E.: Teaching Program Specification and Verification Using JML and ESC/Java2, pp. 92–104. Springer (2009)
16. Romero, A.: El uso de los demostradores automáticos de teoremas para la enseñanza de la programación. In: Proceedings of Jornadas de Enseñanza Universitaria de la Informática (JENUI 2013) (2013)
17. Sznuk, T., Schubert, A.: Tool support for teaching Hoare logic. In: International Conference on Software Engineering and Formal Methods. pp. 332–346. Springer (2014)
18. T. Nipkow, M. Wenzel, L.C.P.: Isabelle 2019 (2019), `https://isabelle.in.tum.de/`