View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UPCommons. Portal del coneixement obert de la UPC

Actas de las XIX Jenui. Castellón, 10-12 de julio 2013
ISBN: 978-84-695-8051-6 DOI: 10.6035/e-TIiT.2013.13
Páginas: 201-208

El uso de los demostradores automáticos de teoremas para la enseñanza de la programación

Ana Romero
Departamento de Matemáticas y Computación
Universidad de La Rioja
Edificio Vives, c/Luis de Ulloa s/n
26004 Logroño
ana.romero@unirioja.es

Resumen

La verificación formal de algoritmos, impartida en los estudios de Ingeniería Informática como parte de las asignaturas de programación, se suele explicar de manera "teórica" introduciendo los axiomas de la lógica de Hoare y realizando diversos ejercicios de verificación (a mano) de pequeños programas. Aunque los alumnos han debido adquirir previamente los conocimientos de Lógica necesarios, muchos de ellos presentan serias dificultades para expresar formalmente los distintos pasos de las pruebas de corrección planteadas. En esta experiencia se ha decidido utilizar como herramienta de apoyo para explicar la verificación formal de algoritmos un demostrador automático de teoremas llamado Krakatoa. Esta herramienta permitirá a los estudiantes visualizar de manera interactiva los distintos pasos necesarios para probar la corrección de un programa, reflexionar sobre los razonamientos seguidos y comprender la importancia de la verificación de algoritmos para mejorar la fiabilidad de nuestros programas.

Abstract

Formal verification of algorithms, taught in Computer Engineering studies as part of programming subjects, is usually explained in a "theoretical" way introducing the different axioms of Hoare logic and doing (by hand) various exercises of verificacion of small programs. Although students are supposed to have previously acquired the necessary logic concepts, many of them have serious difficulties to formally express the different steps of the considered correction proofs. In this experience we have decided to explain formal verification of algorithms by using, as a support tool, an automatic theorem prover called Krakatoa. This tool will allow students to interactively visualize the various 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.

Palabras clave

Verificación formal de algoritmos, pruebas de corrección, demostradores automáticos de teoremas, Krakatoa.

1. Introducción

En el campo de la Ingeniería del Software, para comprobar que un programa desarrollado es *correcto* por norma general se suele realizar una fase de testing, en la que se seleccionan un conjunto de datos de entrada para determinar si los resultados producidos por el programa sobre esos datos coinciden o no con los valores esperados. Aunque existen aplicaciones que generan automáticamente de manera aleatoria tantos casos de prueba como se deseen y comprueban si el programa devuelve lo esperado, para asegurar que el programa es correcto esto no es suficiente ya que el programa se debería probar sobre el conjunto (habitualmente infinito) de *todos* los valores posibles de los datos de entrada. Esto quiere decir que el testing sólo puede asegurar que el programa funciona correctamente para un conjunto limitado de entradas, sin asegurar su correcto funcionamiento para el resto.

La verificación formal de algoritmos es una técnica que permite probar la corrección de un algoritmo antes de ser implementado en un lenguaje de programación concreto, asegurando que el programa es correcto sin necesidad de hacer testing. Aunque la verificación formal de algoritmos no aparece recogida en las recomendaciones del Consejo de Universidades para el título oficial del Grado en Ingeniería Informática (BOE número 187 de 4/8/2009, páginas 66699 a 66710), sí que se contempla en la mayoría de los planes de estudios

de nuestras universidades como parte de algunas de las asignaturas de programación, y permite a los estudiantes completar su formación desde una perspectiva más formal.

En la Universidad de La Rioja la verificación formal de algoritmos forma parte de los contenidos de la asignatura "Especificación y Desarrollo de Sistemas de Software", que se imparte en el segundo cuatrimestre del segundo curso. Esta asignatura ya formaba parte (con contenidos muy similares, incluyendo también la verificación formal de algoritmos) del plan de estudios (actualmente a extinguir) de Ingeniería Técnica en Informática de Gestión.

Desde la implantación del plan de estudios anterior hasta el curso 2011/2012, ya dentro del Grado, esta materia se ha venido explicando de una manera "teórica", presentando mediante clase magistral los axiomas de la lógica de Hoare [1] y realizando algunos ejercicios de verificación (a mano) de pequeños programas escritos en un lenguaje imperativo. A pesar de ser el método *habitual* de enseñanza de esta materia en las diferentes universidades, a lo largo de los años se ha observado que este método no resulta totalmente satisfactorio ya que muchos alumnos presentan serias dificultades a la hora de expresar formalmente los distintos pasos que aparecen en las pruebas de corrección y manifiestan que se trata de un tema difícil y de poca utilidad. Aunque los resultados globales de la asignatura no son malos, se intuye que en muchos casos, para superar la prueba de evaluación, los estudiantes optan por mecanizar el proceso de verificación formal de un algoritmo sin llegar a entender correctamente esta materia.

En el curso 2012/2013 se ha decidido completar la enseñanza "tradicional" (teórica) seguida hasta ahora en la asignatura EDSS utilizando como ayuda un demostrador automático de teoremas llamado Krakatoa[1]. Esta herramienta permite especificar formalmente programas Java y probar su corrección. Partiendo de un método Java con su precondición y postcondición especificadas formalmente, Krakatoa muestra los pasos necesarios para verificar formalmente la corrección del programa y permite visualizar cómo se prueba cada uno de estos pasos, en algunos casos automáticamente o por medio de ciertos lemas y aserciones que el usuario puede introducir.

La experiencia pretende utilizar Krakatoa como herramienta de apoyo, mostrando a los estudiantes cómo esta aplicación es capaz de demostrar automáticamente algunos de los ejemplos vistos en clase de manera teórica. No se pretende que los alumnos sean capaces de trabajar de manera autónoma con el demostrador automático de teoremas, ya que se considera que esta tarea requeriría una dificultad excesiva para los objetivos perseguidos en la asignatura. Como herramienta de apoyo, Krakatoa permite a los estudiantes visualizar en un entorno interactivo los distintos pasos que realizan a mano para dar la prueba de la corrección de un algoritmo, comprender mejor los razonamientos seguidos y entender la importancia de la verificación de algoritmos para mejorar la fiabilidad de nuestros programas.

[1]The Krakatoa verification tool for Java programs. http://krakatoa.lri.fr/

2. Contexto de la experiencia

La asignatura "Especificación y Desarrollo de Sistemas de Software"(EDSS), en los estudios del Grado en Ingeniería Informática de la Universidad de La Rioja, es una asignatura obligatoria que se imparte en el segundo curso, segundo cuatrimestre. Consta de 6 créditos ECTS, divididos en 30 horas presenciales de teoría, 28 horas presenciales de prácticas de laboratorio, 2 horas para la prueba de evaluación final y 90 horas de trabajo autónomo del alumno. La asignatura es común al Grado en Matemáticas; los alumnos de ambos grados asisten juntos a clase y tanto los contenidos como los criterios de evaluación son iguales. El número total de matriculados suele estar en torno a 50. La asignatura ya formaba parte (con contenidos muy similares, incluyendo entre ellos la verificación formal de algoritmos) de los planes de estudios (actualmente a extinguir) de Ingeniería Técnica en Informática de Gestión y Licenciatura en Matemáticas, que comenzaron a impartirse en el curso 2002/2003.

EDSS es la cuarta asignatura dentro del módulo "Programación". Además de haber cursado las tres asignaturas predecentes ("Metodología de la programación", "Tecnología de la programación" y "Programación orientada a objetos"), se supone que los estudiantes han adquirido anteriormente los conocimientos fundamentales de la lógica de primer orden impartidos en la asignatura "Lógica", del primer curso, segundo cuatrimestre.

Como se indica en la guía docente de la asignatura[2], EDSS tiene entre sus objetivos aportar una perspectiva formal (mayor nivel de abstracción) sobre diferentes aspectos relacionados con la programación (sintaxis, semántica, corrección y eficiencia), buscando una mejora en los hábitos del alumno a la hora de programar, que mejore la calidad y fiabilidad de su trabajo. Después de abordar diversos temas como la especificación e implementación de Tipos Abstractos de Datos y su relación con la Programación Orientada a Objetos, la especificación de algoritmos y las nociones de sintaxis y semántica de un lenguaje, la parte final de la asignatura (unas 5 semanas de clase, con 10 horas presencia-

[2]https://aps.unirioja.es/GuiasDocentes/servlet/agetdocumentopdf?2012-13,801G,830,2,2

les) está dedicada a la verificación formal de algoritmos.

La evaluación de la asignatura se divide en dos partes: 3 puntos corresponden a la realización de las prácticas (de desarrollo de programas en Java), y los otros 7 al examen final (en papel) que consta de diversos ejercicios de los temas tratados. Dentro del examen final, los ejercicios de verificación formal de algoritmos suponen aproximadamente un $40\,\%$ del total.

3. Verificación formal de algoritmos. Enseñanza tradicional

Tanto dentro del plan de estudios anterior en Ingeniería Técnica en Informática de Gestión (desde el curso 2002/2003 hasta el 2009/2010) como en los cursos 2010/2011 y 2011/2012, ya dentro del Grado en Ingeniería Informática, la verificación formal de algoritmos se ha venido explicando en la asignatura "Especificación y Desarrollo de Sistemas de Software" de una manera "tradicional" o "teórica".

Para desarrollar el tema de verificación formal de algoritmos, en primer lugar se introducen mediante clases magistrales los axiomas de la lógica de Hoare [1], que dan las reglas de inferencia necesarias para probar que un programa cumple una especificación dada mediante una precondición y una postcondición.

Dadas una precondición Q y una postcondición R, un programa "s" (formado por una serie de instrucciones elementales $s \equiv \{s_1, \ldots, s_n\}$) cumple la especificación $\{Q\}s\{R\}$ si siempre que se ejecuta s comenzando en un estado que verifica Q, el programa termina y se llega a un estado que satisface R. Para probar la corrección de $\{Q\}s\{R\}$ se consideran predicados que determinan los estados válidos en los puntos intermedios del programa, denominados *asertos* o *aserciones*, tal que $\{Q\}s_1\{P_1\}s_2\{P_2\}\ldots\{P_{n-1}\}s_n\{R\}$. Si el aserto inicial Q (precondición) se satisface, y cada "programa" elemental s_k, consistente en una sola instrucción, satisface su especificación $\{P_{k-1}\}s_k\{P_k\}$, entonces se satisface finalmente la postcondición R y el programa es correcto.

La lógica de Hoare da reglas para probar la corrección de las instrucciones básicas de un lenguaje de programación (asignación, composición secuencial, condicional, composición iterativa...). Estas reglas permiten calcular de manera mecánica precondiciones válidas a partir de una postcondición dada para asignaciones, composiciones secuenciales y sentencias condicionales. En el caso de la composición iterativa es necesario construir un *invariante* P, que es un predicado que se cumple antes de entrar al bucle y tras cada iteración, y que debe ser lo suficientemente *fuerte* para que a partir de él, a la salida del bucle, podamos deducir la postcondición.

En clase de EDSS se presentan (de manera teórica) cada una de las reglas de Hoare para las instrucciones básicas de un lenguaje imperativo, tal y como viene explicado en [3], y se realizan pequeños ejemplos elementales de aplicación de cada una de ellas. Una vez introducidas todas las reglas, se realizan algunos ejercicios de pruebas de verificación formal de pequeños programas completos con un esquema iterativo.

Las pruebas de corrección realizadas en la asignatura EDSS se *limitan* a programas que responden al siguiente esquema:

```
{Q}
<inicializar>
mientras que B hacer
     <cuerpo>
fmq
devuelve (<var>)
{R}
```

donde los bloques `<inicializar>` y `<cuerpo>` están formados por una secuencia de instrucciones elementales, normalmente asignaciones y/o estructuras condicionales. En realidad esto no supone ninguna restricción, ya que si hay varios bucles "hermanos" puede pensarse que todos menos el último están en `<inicializar>`, y si hay bucles anidados puede pensarse que los bucles internos están en `<cuerpo>`.

Según la axiomática de Hoare, para verificar un programa que tenga el esquema anterior hace falta:

1. Encontrar un invariante P para el bucle.
2. Probar que se cumple $\{Q\}$`<inicializar>`$\{P\}$.
3. Probar que P es invariante, es decir, se cumple $\{P$ and $B\}$ `<cuerpo>` $\{P\}$.
4. Probar que $\{P$ and not$(B)\}$ es más fuerte que la postcondición $\{R\}$.
5. Asegurar que el bucle termina.

Siguiendo estos pasos se consideran en la asignatura las pruebas de corrección de algunos algoritmos sencillos como por ejemplo el cálculo (iterativo) de la potencia de un número real elevado a un entero positivo, el cálculo del factorial de un número entero, el cálculo de la raíz cuadrada entera, la búsqueda (secuencial) de un elemento en un vector o la suma de las componentes de un vector. Tras presentar algunos de estos ejercicios en la pizarra, se realizan también algunas clases de problemas en las que los alumnos deben poner en práctica lo aprendido y realizar ellos solos algunas pruebas de corrección propuestas.

Este método "tradicional", utilizado también en otras muchas universidades, presenta algunos problemas, debido sobre todo a que los estudiantes del Grado en Ingeniería Informática poseen una formación de carácter técnico adecuada pero tienen más dificultad en adquirir los conocimientos y las competencias en los aspectos más matemáticos, lógicos y formales.

A pesar de que los ejemplos y ejercicios de pruebas de corrección realizados en la asignatura EDSS son programas sencillos, a lo largo de los años se ha observado que los alumnos presentan serias dificultades a la hora de expresar formalmente los distintos pasos necesarios para aplicar las reglas de inferencia de la lógica de Hoare y realizar las pruebas de corrección planteadas. Muchos estudiantes manifiestan que el tema de verificación de algoritmos es el que más difícil les resulta de la asignatura y así lo reflejan también las calificaciones obtenidas en la parte correspondiente a esta materia dentro del examen final realizado. Además, aunque los resultados globales de la asignatura no son malos, se observa el caso de algunos alumnos que consiguen superar la evaluación porque terminan por mecanizar el proceso de las pruebas de corrección, muchas veces sin llegar a entenderlo totalmente.

4. Demostradores automáticos de teoremas y verificación formal de algoritmos

En el curso 2012/2013 se ha decidido completar el método "teórico" seguido hasta ahora para explicar la verificación formal de algoritmos utilizando como herramienta de apoyo una aplicación de verificación automática de programas escritos en Java llamada Krakatoa. Para probar la corrección de un programa Java, Krakatoa recibe su especificación (precondición y postcondición) escrita en el lenguaje Java Modeling Language [2] (JML) y por medio de una herramienta llamada Why[3] genera una serie de lemas (llamados "proof obligations", obligaciones) que corresponden a los pasos necesarios, según la lógica de Hoare, para probar la corrección del programa. Estos lemas tratarán de ser probados mediante algunos demostradores automáticos incorporados dentro de Krakatoa, y si no se consigue, podrán ser enviados a Coq[4], un demostrador de teoremas interactivo en el que el usuario puede construir sus propias pruebas.

Por ejemplo, el siguiente método Java calcula el máximo de dos enteros:

```
/*@ ensures \result >= x && \result >= y
  @ && \forall integer z; z >= x && z >= y
  @ ==> z >= \result;
  @*/
public static int max(int x, int y) {
    if (x>y) return x; else return y;
}
```

La especificación del método, escrita en lenguaje JML, se indica con comentarios entre `/*@` y `@*/`. La cláusula `ensures` indica la postcondición, que es un predicado que deberá verificarse a la salida del método para cualquier valor de sus argumentos. Dentro de la postcondición, `result` denota el valor devuelto por el método. La postcondición en este caso significa:

- el resultado es mayor o igual que x,
- el resultado es mayor o igual que y,
- el resultado es el menor de todos los posibles enteros que sean mayores que x e y.

El objetivo es probar que el método `max` está implementado correctamente, es decir, que satisface la especificación dada. Como muestra la Figura 1, Krakatoa genera para ello 6 lemas (*obligaciones*) que expresan la validez del programa. Estas 6 obligaciones prueban cada una de las 3 componentes de la postcondición, que deben verificarse para cada una de las dos ramas de la clásula condicional, y corresponden exactamente a los pasos que los alumnos deberían realizar para probar formalmente (de modo teórico) la corrección del programa. El primer lema, que aparece detallado en la imagen, indica que, en el caso $x > y$, se verifica que el resultado es mayor o igual que x. En este caso los lemas son sencillos y los demostradores automáticos Alt-Ergo[5] y CVC3 [6] incorporados dentro Krakatoa son capaces de probarlos directamente. La prueba de las 6 obligaciones demuestra la corrección del programa respecto a la especificación dada, lo que garantiza que en cualquier situación, es decir, para cualquiera de los (infinitos) posibles datos de entrada, el programa devuelve el resultado esperado.

La prueba de corrección de un programa se complica cuando se utilizan estructuras iterativas. Consideremos ahora el siguiente método para calcular la raíz cuadrada de un número entero:

```
/*@ requires x >= 0;
  @ ensures
  @   \result >= 0 && \result * \result <= x
  @   && x < (\result + 1) * (\result + 1);
  @*/
public static int sqrt(int x) {
    int count = 0, sum = 1;
    while (sum <= x) {
        count++;
        sum = sum + 2*count+1;
    }
    return count;
}
```

La claúsula `requires` introduce la precondición, que es un predicado que se debe cumplir al llamar al método. En este caso se pide que el parámetro x sea no negativo, ya que en otro caso no sería posible calcular su raíz cuadrada. Krakatoa genera ahora 5 obligaciones (3 para probar la postcondición y dos para probar

[3]The Why verification tool. http://why.lri.fr

[4]The Coq Proof Assistant. http://coq.inria.fr/

[5]Alt-Ergo: An automatic theorem prover dedicated to program verification. http://alt-ergo.lri.fr/

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

Figura 1: Obligaciones generadas por Krakatoa para el método max.

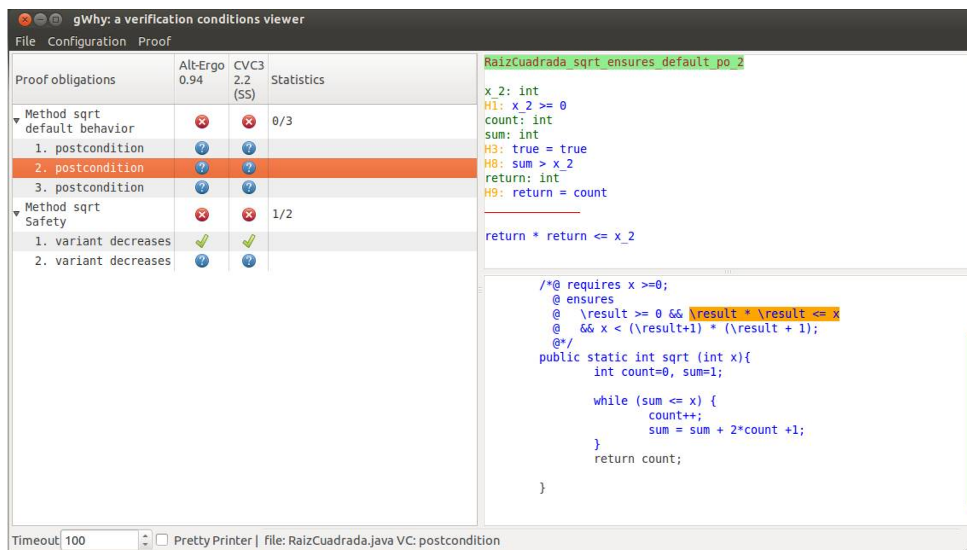

Figura 2: Obligaciones generadas por Krakatoa para el método sqrt sin especificar el invariante.

que el método es seguro) pero, como se observa en la Figura 2, sólo es capaz de probar una de ellas.

Para probar la corrección de un programa con una estructura iterativa siguiendo los axiomas de la lógica de Hoare, como ya hemos explicado en la Sección 4, hace falta definir un invariante P que es un predicado que se cumple a la entrada del bucle y tras cada iteración. Este invariante tiene que ser lo suficientemente fuerte para que de él, a la salida del bucle, se deduzca la postcondición, por lo que no siempre es sencillo deducir cuál es el invariante correcto.

Para introducir el invariante en la especificación JML en Krakatoa utilizamos la claúsula `loop invariant`. Además, para poder comprobar que el bucle termina (y por tanto el método es seguro), en muchas ocasiones hay que indicarle a Krakatoa una expresión entera y no negativa que decrezca en cada iteración, llamada `loop variant`, asegurando así que siempre llegará un momento en que se deje de cumplir la condición de entrada. Para la estructura iterativa dentro del método `sqrt` podemos utilizar la siguiente especificación:

```
public static int sqrt(int x) {
    int count = 0, sum = 1;
    /*@ loop_invariant
      @   count >= 0 && x >= count*count &&
      @   sum == (count+1)*(count+1);
      @ loop_variant x - sum;
      @*/
    while (sum <= x) {
        count++;
        sum = sum + 2*count+1;
    }
    return count;
}
```

Ahora Krakatoa genera 11 obligaciones; algunas de ellas han aparecido al incluir el invariante, y aseguran que éste verifica las propiedades necesarias. Podemos observar que las obligaciones generadas corresponden

a los pasos indicados en la Sección 4 para la prueba de corrección “teórica”. Con la *ayuda* del invariante el demostrador CVC3 sí que es capaz de probar la corrección del programa, como vemos en la Figura 3.

En otros casos más complicados, además de definir el invariante y el *variante* para las estructuras iterativas, puede ser necesario incluir predicados o definiciones axiomáticas o añadir “pistas”, llamadas *aserciones*, que ayuden a probar los lemas generados por Krakatoa. Además, si los demostradores automáticos incorporados dentro de Krakatoa no son capaces de probar las obligaciones generadas, también existe la posibilidad de *enviar* estos lemas (a través de un fichero que se genera automáticamente) al demostrador interativo Coq y realizar la prueba dentro de él.

La experiencia planteada en la asignatura EDSS pretende utilizar Krakatoa como herramienta de apoyo para la enseñanza de la verificación formal de algoritmos. Una vez que los alumnos conocen los axiomas de la lógica de Hoare y han visto cuáles son los pasos teóricos necesarios para dar la prueba de corrección de un programa sencillo, se mostrará a los estudiantes cómo la herramienta es capaz de demostrar automáticamente algunos de los ejemplos vistos en clase de manera teórica. Se observará que los lemas generados por Krakatoa corresponden a los distintos pasos teóricos de la lógica de Hoare y se verá la necesidad de definir un invariante que sea lo suficientemente fuerte para que de él, a la salida del bucle, se pueda deducir la postcondición.

No se pretende que los estudiantes sean capaces de trabajar de manera autónoma con el demostrador automático de teoremas, lo que requeriría una dificultad excesiva para los objetivos perseguidos. Sin embargo, como herramienta de apoyo, Krakatoa permitirá a los estudiantes visualizar en un entorno interactivo los distintos pasos que realizan a mano para dar la prueba de la corrección de un algoritmo, comprender mejor los razonamientos seguidos y entender la importancia de la verificación de algoritmos para mejorar la fiabilidad de nuestros programas.

Para evaluar la comprensión de la herramienta en su primer año de utilización se incorporará un ejercicio dentro del examen teórico con algunas cuestiones sencillas sobre Krakatoa como la identificación de las distintas partes de la especificación JML de un método Java o la detección de elementos incorrectos en un código dado.

5. Algunos ejemplos de pruebas de corrección con Krakatoa

5.1. Búsqueda secuencial

Consideramos uno de los ejemplos (sencillos) de prueba de corrección que se han explicado los últimos años en EDSS: un algoritmo para la búsqueda secuencial de un dato en un vector de enteros.

La implementación de este algoritmo en Java vendría dada por el siguiente método:

```
public static boolean busqueda_secuencial
                         (int v[], int x) {
    int i=0;
    boolean esta=false;
    while (i<v.length ) {
        if (v[i] == x) esta=true;
        i=i+1;
    }
    return esta;
}
```

Tras presentar la idea del método, los estudiantes deberían pensar (en lenguaje lo más formal posible) cuál sería su especificación (precondición y postcondición). La postcondición debe expresar que el método devuelve cierto si el elemento x está en el vector v, y falso en otro caso.

Para dar la especificación en JML, se mostraría a los estudiantes que Krakatoa permite definir predicados. En este caso es conveniente definir el siguiente predicado, que indica si un dato x aparece entre las n primeras componentes de un vector v:

```
/*@ predicate contiene{L}(int[] v,
  @                 integer n, integer x) =
  @   0 <= n && n < v.length &&
  @   (\exists integer i;
  @     0 <= i && i <= n && v[i]==x );
  @*/
```

Utilizando este predicado, los alumnos podrían tratar de escribir, utilizando la sintaxis de JML, la especificación del método `busqueda secuencial`, que sería:

```
/*@ requires v != null && 1 <= v.length ;
  @ ensures \result <==>
  @     contiene(v,v.length-1,x);
  @*/
```

Con esta especificación, se mostraría que Krakatoa genera 8 obligaciones de las cuales sólo es capaz de probar 3. Como ya se ha comentado anteriormente, es necesario definir el invariante P para la estructura condicional. Los estudiantes pensarían un invariante adecuado y se discutirían en clase las posibles opciones planteadas, valorando si son o no correctas, e implementando algunas de las sugerencias en JML para ir

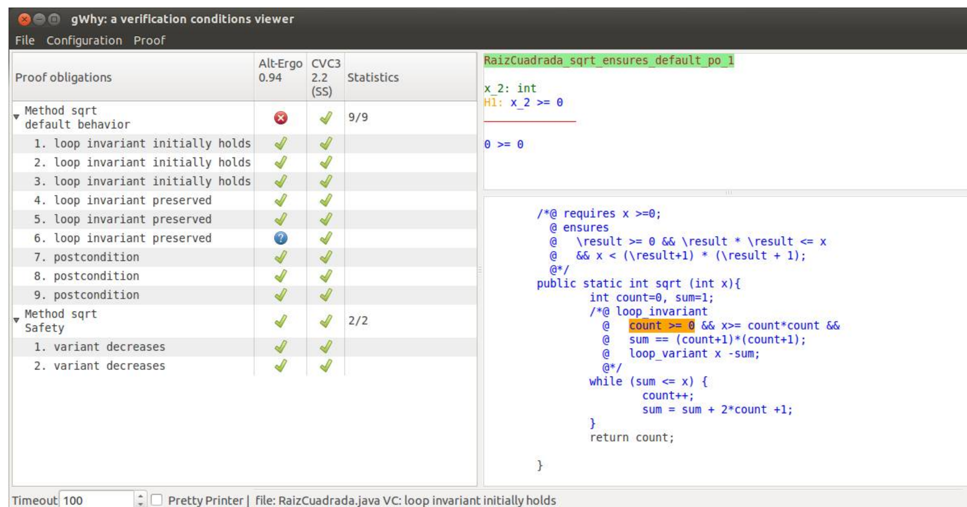

Figura 3: Obligaciones generadas por Krakatoa para el método sqrt con el invariante definido.

viendo con cada una de ellas si Krakatoa consigue o no probar la corrección del programa.

Una solución para el invariante (y el *variante*) en JML sería:

```
/*@ loop_invariant
  @   0<=i && i<=v.length &&
  @     (esta==true <==> contiene(v,i-1,x));
  @ loop_variant
  @   v.length-i ;
  @*/
```

Finalmente, observaríamos que con esta ayuda Krakatoa sí que es capaz de verificar el programa, asegurando así su corrección.

5.2. Factorial y potencia de exponente natural

Otros dos ejemplos sencillos de pruebas de corrección de algoritmos que se han realizado en años anteriores en la asignatura de EDSS de manera teórica y que ahora se pueden presentar con la ayuda de Krakatoa son el cálculo de la potencia de exponente natural y el factorial de un número natural.

Como la versión de JML soportada por Krakatoa no incorpora la posibilidad de utilizar el factorial ni la potencia, para poder especificar los métodos en Krakatoa hacen falta algunas definiciones axiomáticas. En el caso del factorial la definición en JML sería:

```
/*@ axiomatic Factorial {
  @ logic integer fact(integer n);
  @ axiom fact_zero :
  @    fact(0) ==1;
  @ axiom fact_sum:
  @    \forall integer n;
  @        fact(n+1)==fact(n)*(n+1);
  @}
  @*/
```

La especificación del método, que los demostradores incorporados en Krakatoa prueban automáticamente, sería:

```
/*@ requires n >=0;
  @ ensures \result==fact(n);
  @*/
public static int factorial(int n){
    int i=0;
    int f=1;
    /*@ loop_invariant
      @ 0 <= i && i <= n && f==fact(i) ;
      @ loop_variant n-i ;
      @*/
    while (i<n) {
        i++;
        f=f*i;
    }
    return f;
}
```

5.3. Raíz cuadrada real utilizando el método de Newton

Podemos considerar también el siguiente método para el cálculo de la raíz cuadrada de un número real utlizando el método de aproximación de Newton (con error menor que un dato `epsi` dado).

```
public static double newtonSqrt
                      (double c, double epsi){
    double t;
    if (c>1) t= c;
        else t=1.1;
    while (t* t - c  >= epsi) {
        t = (c/t + t) / 2.0;
    }
    return t;
}
```

En este caso, además de dar el invariante, para probar que la expresión $t * t - c$ decrece en cada iteración

y por tanto el bucle termina, hace falta introducir en Krakatoa el siguiente lema:

```
/*@ lemma newton_decreases :
  @ \forall double c t;
  @   t>0 && t*t > c ==>
  @   (c/t + t) / 2.0 < t;
  @
*/
```

Aunque los demostradores automáticos incorporados dentro de Krakatoa no son capaces de probar este lema, sí que se puede observar en clase que a partir de él la aplicación es capaz de demostrar la corrección del método. Para poder demostrar formalmente el lema anterior habría que hacer uso de Coq, pero su utilización resulta mucho más compleja y no se plantea dentro de los objetivos de la experiencia.

6. Conclusiones

La verificación formal de algoritmos forma parte de las asignaturas de programación en los estudios en Ingeniería Informática y se suele enseñar de manera teórica, presentando los axiomas de la lógica de Hoare y realizando (a mano) algunos ejemplos de pruebas de corrección de programas sencillos. Este método, seguido durante varios años en la asignatura de "Especificación y Desarrollo de Sistemas de Software" de la Universidad de La Rioja, presenta algunas carencias, debido sobre todo a que muchos estudiantes tienen dificultades para expresar formalmente los diferentes pasos de las pruebas.

En la experiencia presentada se ha utilizado una herramienta de apoyo llamada Krakatoa para intentar que los alumnos consigan entender mejor las pruebas de corrección realizadas en clase. Krakatoa es una aplicación que permite verificar automáticamente programas escritos en lenguaje Java, que deben ser especificados (indicando, entre otras cosas, su precondición y postcondición), en lenguaje JML. Para probar la corrección de un programa, Krakatoa genera una serie de lemas que pueden ser probados por algunos demostradores automáticos de teoremas incorporados o usando el demostrador interactivo Coq. La prueba de estos lemas certifica la corrección del programa, y asegura por tanto que su comportamiento será el esperado sin necesidad de hacer testing.

En la experiencia se muestra cómo Krakatoa es capaz de realizar automáticamente algunas de las pruebas de corrección que los estudiantes han hecho anteriormente a mano, siguiendo los mismos pasos que los realizados en el método teórico. Aunque no se trata de que los alumnos aprendan a manejar la aplicación, el uso de este entorno interactivo como herramienta de apoyo permite que los estudiantes comprendan mejor las pruebas planteadas y los pasos a realizar necesarios en cada una de ellas.

En este primer año de aplicación no se conocen aún los resultados en las pruebas de evaluación, pero se espera que las calificaciones del examen final en la parte correspondiente a la verificación formal sean mayores que en años anteriores, al conseguir profundizar más en el tema mediante la ayuda de Krakatoa. Los resultados reales estarán disponibles al final del cuatrimestre actual y se mostrarán en la presentación del trabajo en las XIX Jornadas sobre la Enseñanza Universitaria de la Informática.

Referencias

[1] Charles Antony Richard Hoare. *An axiomatic basis for computer programming*. Communications of the ACM, 12 (1969) 576-580.

[2] Gary T. Leavens, Albert L. Baker y Clyde Ruby. *Preliminary design of JML: A behavioral interface specification languaje for Java*. Technical report 98-06, Iowa State University, 2000.

[3] Ricardo Peña Marí. *Diseño de programas. Formalismo y abstracción*. Prentice Hall, 1993.